TinyBlog: Créer votre Première Application Web avec Pharo

Olivier Auverlot, Stéphane Ducasse et Luc Fabresse

April 26, 2019

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

Contents

II Eléments optionnels

Illustrations

CHAPTER 1

A propos de ce livre

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

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

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

1.1 Structure

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

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

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

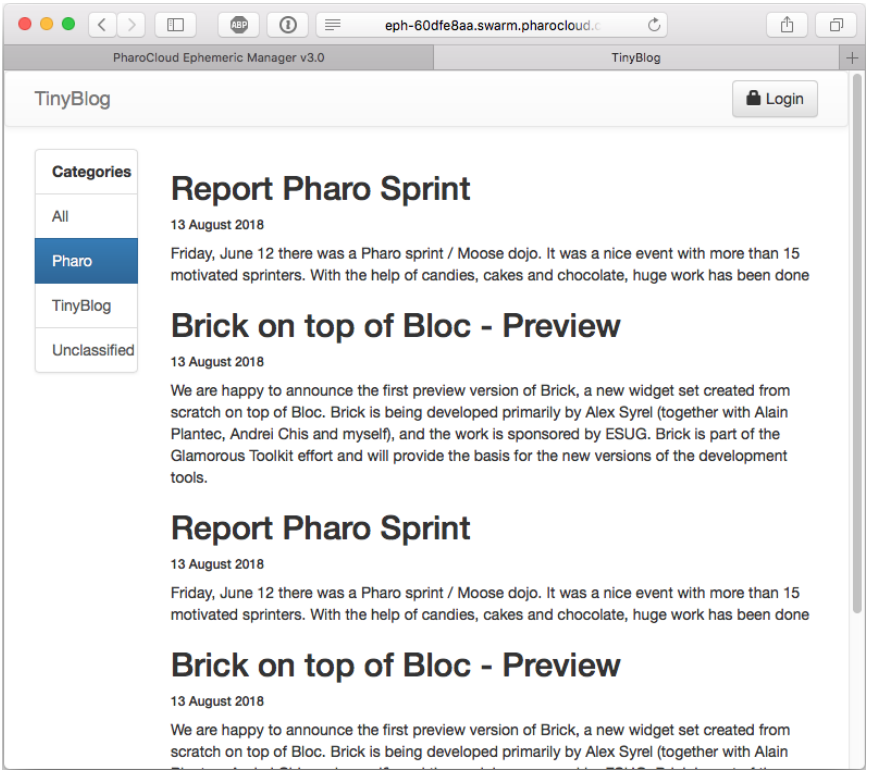


Figure 1-1 L'application TinyBlog.

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

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

1.2 Installation de Pharo

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

Pour développer votre application TinyBlog et suivre ce tutoriel, nous vous suggérons de toujours utiliser l'image disponible à l'adresse suivante: http://mooc.pharo.org/image/PharoWeb-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 (http://pharo.org/download):

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

1.3 Règles de nommage

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

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

1.4 Ressources

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

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

- Sur la page Web http://pharo.org/community vous trouverez le lien vers le channel discord où nombre de Pharoers échangent et s'entraident.

1.5 Remerciements

Les auteurs remercient chaleureusement René Paul Mages pour sa relecture attentive de ce livre.

Part I

Tutoriel de base

CHAPTER 2

Modèle de l'application TinyBlog

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

2.1 La classe TBPost

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

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

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

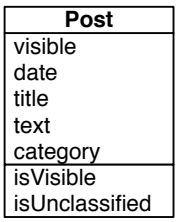


Figure 2-1 TBPost une classe très simple gérant principalement des données.

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

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

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

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

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

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

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

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

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

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

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

```
TBPost >> 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.

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

```
TBPost >> notVisible
   self visible: false
```

2.3 Initialisation

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

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

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

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

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

2.4 Méthodes de création

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

Nous définissons deux méthodes.

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

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

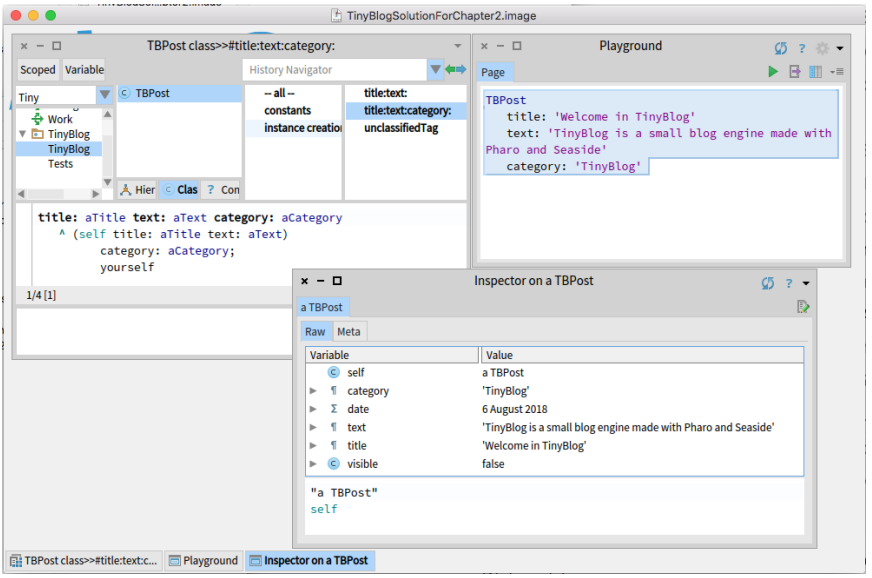


Figure 2-2 Inspecteur sur une instance de TBPost.

2.5 Création de posts

Créons des posts pour s'assurer que tout fonctionne. Ouvrez l'outil Playground et executez 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 Ajout de quelques tests unitaires

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

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

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

Nous définissons deux tests.

```
TBPostTest >> testWithoutCategoryIsUnclassified

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

```
TBPostTest >> testPostIsCreatedCorrectly

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

Vos tests doivent passer.

2.7 Interrogation d'un post

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

```
TBPost >> isVisible
   ^ self visible
```

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

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

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

```
TBPostTest >> testWithoutCategoryIsUnclassified

  | post |
  post := TBPost
    title: 'Welcome to TinyBlog'
```

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

2.8 Conclusion

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

CHAPTER 3

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 débuggeur pour être encore plus productif. Nous ne l'avons pas fait car le modèle est simpliste et expliquer comment coder dans le débuggeur demande plus de description textuelle. Vous pouvez voir cette pratique dans la vidéo du Mooc intitulée *Coding a Counter in the Debugger* (http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_26.html) et lire le livre *Learning Object-Oriented Programming, Design with TDD in Pharo* (http://books.pharo.org).

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

3.1 La classe TBBlog

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

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

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

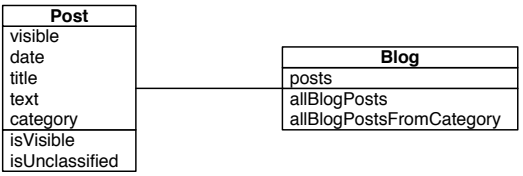


Figure 3-1 TBBlog une classe très simple.

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

3.2 Un seul blog

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

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

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

Nous définissons deux méthodes pour gérer le singleton.

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

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

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

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

3.3 Tester les règles métiers

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

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

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

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

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

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

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

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

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

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

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

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

3.4 Un premier test

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

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

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

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

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

```
TBBlog >> size
   ^ posts size
```

Le test précédent doit maintenant passer.

3.5 Améliorons la couverture de test

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

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

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

3.6 Autres fonctionnalités

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

Obtenir l'ensemble des posts (visibles et invisibles)

Ajoutons un nouveau test qui échoue :

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

Et le code métier qui permet de le faire passer:

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

Votre nouveau test doit passer.

Obtenir tous les posts visibles

Ajoutons un nouveau test qui échoue :

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

Voici le nouveau code métier ajouté :

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

Votre nouveau test doit passer.

Obtenir tous les posts d'une catégorie

Ajoutons un nouveau test qui échoue :

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

Voici le nouveau code métier ajouté :

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

Votre nouveau test doit passer.

Obtenir tous les posts visibles d'une catégorie

Ajoutons un nouveau test qui échoue :

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

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

Voici le nouveau code métier ajouté :

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

Votre nouveau test doit passer.

Vérifier la gestion des posts non classés

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

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

Obtenir la liste des catégories

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

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

Voici le code métier :

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

Votre nouveau test doit passer.

3.7 Données de test

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

```
TBBlog class >> createDemoPosts
   "TBBlog createDemoPosts"
   self current
      writeBlogPost: ((TBPost title: 'Welcome in TinyBlog' text:
    'TinyBlog is a small blog engine made with Pharo.' category:
    'TinyBlog') visible: true);
      writeBlogPost: ((TBPost title: 'Report Pharo Sprint' text:
    'Friday, June 12 there was a Pharo sprint / Moose dojo. It was a
    nice event with more than 15 motivated sprinters. With the help
    of candies, cakes and chocolate, huge work has been done'
    category: 'Pharo') visible: true);
```

```
  writeBlogPost: ((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.

3.8 Futures évolutions

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

3.9 Conclusion

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

CHAPTER 4

Persistance des données de TinyBlog avec Voyage et Mongo

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

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

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

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

4.1 Configurer Voyage pour sauvegarder des objets 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.

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

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

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

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

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

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

Le cas de la méthode `current` est plus délicat. Avant l'utilisation de Mongo, nous avions un singleton tout simple. Cependant utiliser un Singleton ne fonctionne plus car imaginons que nous ayons sauvé notre blog et que le serveur s'éteigne par accident ou que nous 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 sauvée.

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

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

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

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

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

4.2 Sauvegarde d'un blog

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

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

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

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

4.3 Revision des tests

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

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

Nous ajoutons la variable d'instance `previousRepository` à la classe `TBBLogTest`.

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

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

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

Dans la méthode `tearDown`, à la fin de l'exécution d'un test nous réinstallons l'objet base données que nous avons sauvé avant l'exécution.

```
TBBlogTest >> tearDown
  VORepository setRepository: previousRepository
```

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

4.4 Utilisation de la base

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

Nous créons des posts ainsi :

```
TBBlog createDemoPosts.
```

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

```
TBBlog count
>1
```

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

```
TBBlog selectAll
```

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

Vous pouvez voir l'API de Voyage en parcourant

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

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

4.5 Si nous devions sauvegarder les posts [Discussion]

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

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

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

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

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

Post comme racine = Unicité

Si vous désirez qu'un bulletin 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 bulletins 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 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 bulletin puis nous remettons à jour la collection.

4.6 Déployer avec une base Mongo [Optionnel]

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

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

Installation de Mongo

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

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

> **Note** Le serveur Mongo ne doit pas utiliser d'authentification (ce n'est pas le cas avec une installation locale par défaut) car la nouvelle méthode de chiffrement SCRAM utilisée par MongoDB 3.0 n'est actuellement pas supportée par Voyage.

Quelques commandes utiles pour la suite :

```
# pour stopper votre conteneur
docker stop mongo

# pour re-démarrer votre conteneur
docker start mongo

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

Connexion à un serveur local

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

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

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

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

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

```
TBBlog reset.
TBBlog createDemoPosts
```

En cas de problème

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

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

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

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

ou dans le conteneur docker :

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

Attention : Changements de TBBlog

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

```
VORepository current reset
```

4.7 Conclusion

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

CHAPTER 5

Commencer avec Seaside

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

Le travail présenté dans la suite est indépendant de celui sur Voyage et sur la base de données MongoDB. Vous pouvez charger le code des chapitres précédents en suivant les instructions indiquées dans le dernier chapitre (Chapitre 13).

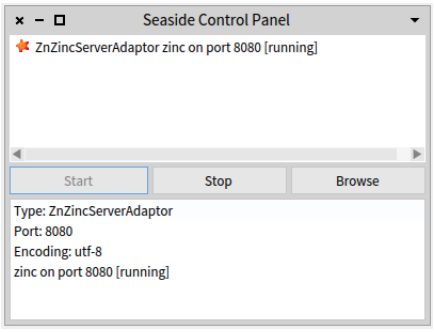


Figure 5-1 Lancer le serveur.

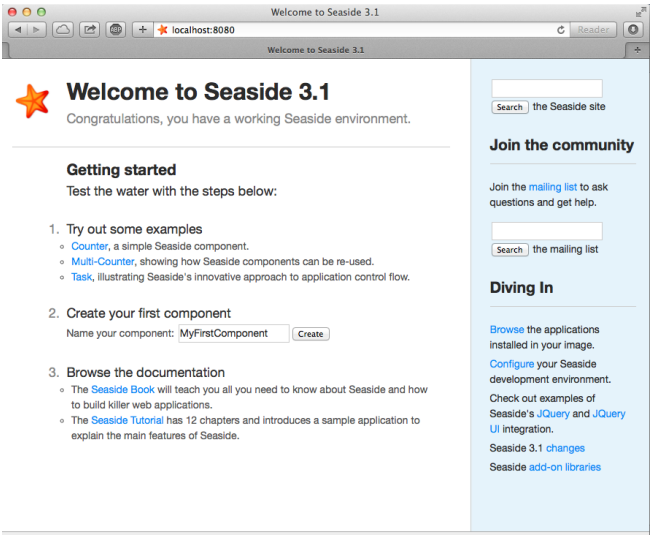


Figure 5-2 Vérification que Seaside fonctionne.

5.1 Démarrer Seaside

Seaside est déjà chargé dans l'image Pharo, sinon relisez le premier chapitre (Chapitre 13). 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 ajoutez 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.

5.2 Bootstrap for Seaside

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

En cliquant sur le lien **Examples** au bas de la page, vous pouvez voir les éléments graphiques et le code pour les obtenir (figure 5-4).

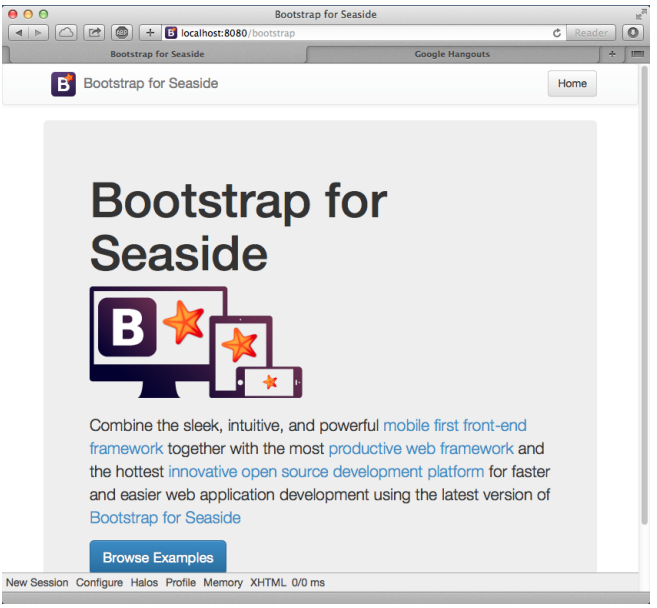


Figure 5-3 Accès à la bibliothèque Bootstrap.

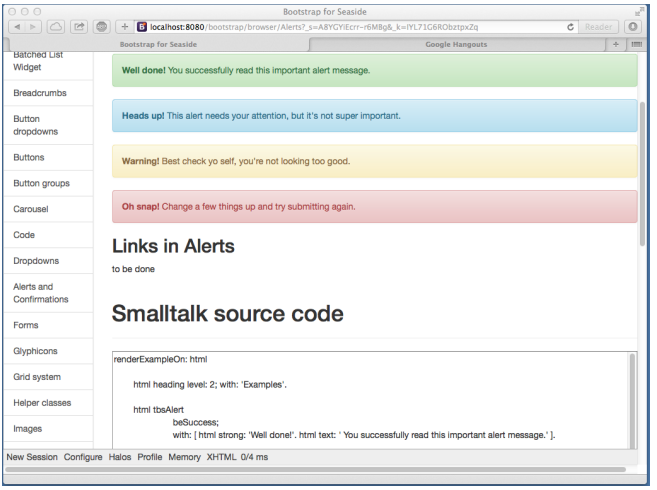


Figure 5-4 Comprendre un élément et son code en Bootstrap.

Bootstrap

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

5.3 Point d'entrée de l'application

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

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

Nous déclarons l'application au serveur Seaside, en définissant coté classe, dans le protocole `'initialization'` 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 de classe `initialize` ne sont automatiquement exécutées que lors du chargement en mémoire de la classe. Ici nous venons juste de la définir et donc il est nécessaire de l'exécuter pour en voir les bénéfices.

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

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

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

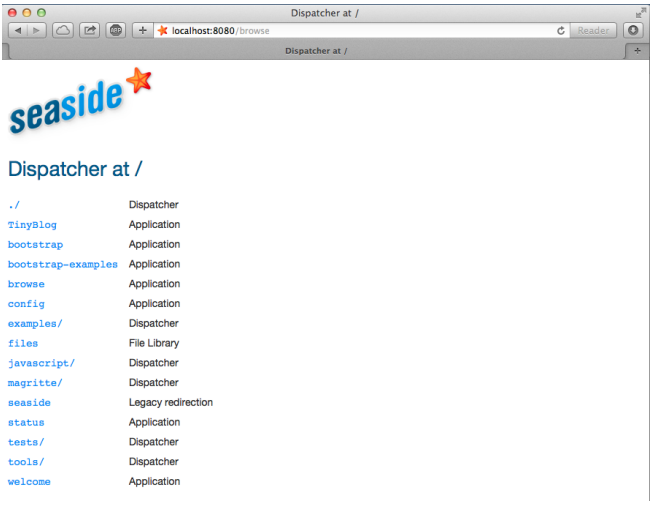


Figure 5-5 TinyBlog est bien enregistrée.

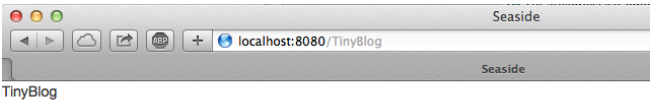


Figure 5-6 Une page quasi vide mais servie par Seaside.

5.4 Premier rendu simple

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

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

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

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

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

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

5.5 Architecture

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

Premier apercu des composants

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

Eléments

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

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

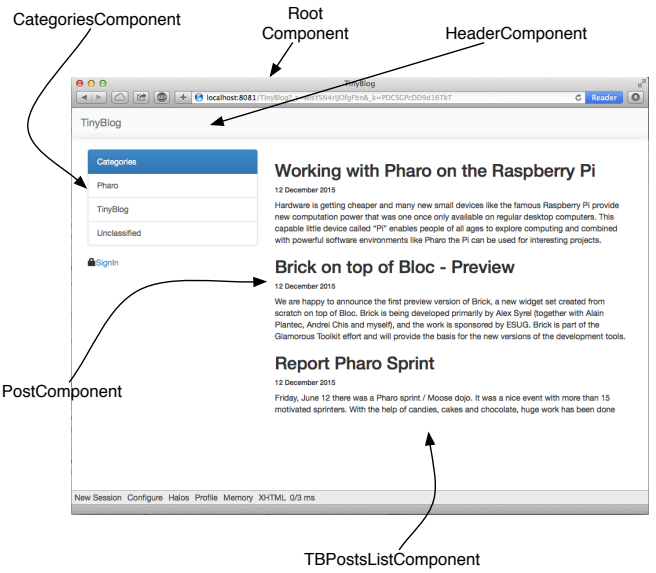


Figure 5-7 Les composants composant l'application TinyBlog (en mode non admin).

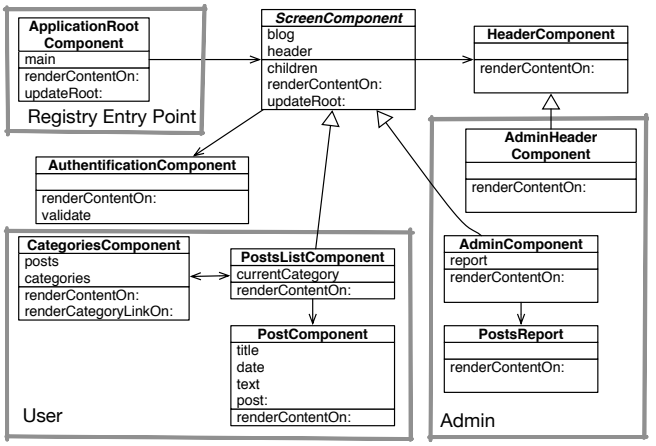


Figure 5-8 Architecture de TinyBlog.

5.6 Conclusion

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

CHAPTER 6

Des composants web pour TinyBlog

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

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

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

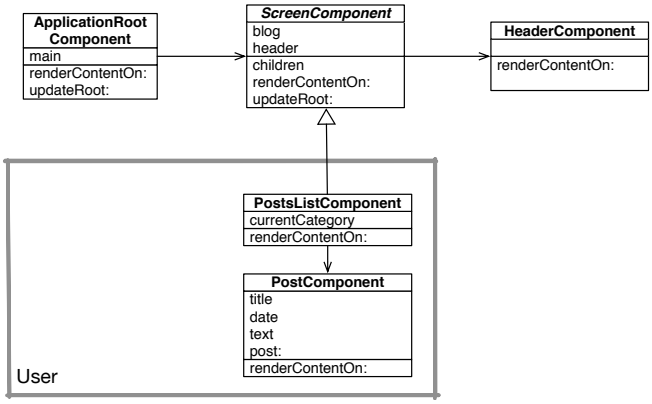


Figure 6-1 L'architecture des composants utilisateurs (par opposition à administration).

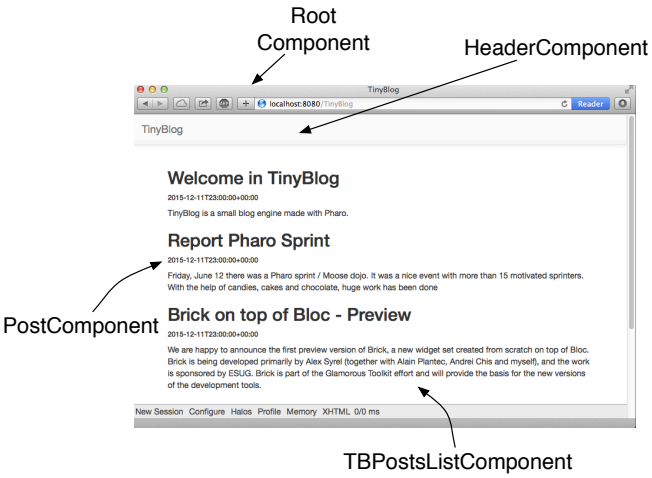


Figure 6-2 Les composants visuels de l'application TinyBlog.

précédents sont disponibles dans le chapitre 13.

6.1 Composants visuels

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

Le composant TBScreenComponent

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

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

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

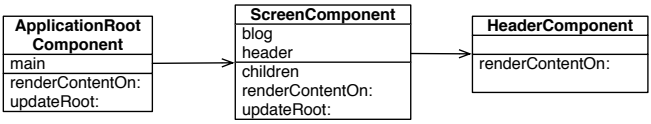


Figure 6-3 Le composant `ApplicationRootComponent` utilise de manière temporaire le composant `ScreenComponent` qui a un `HeaderComponent`.

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

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

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

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

6.2 Utilisation du composant Screen

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

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

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

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

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

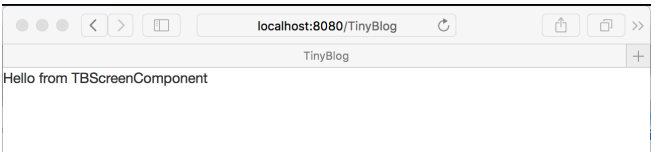


Figure 6-4 Premier visuel du composant `TBScreenComponent`.

```
TBApplicationRootComponent >> renderContentOn: html
   html render: main
```

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

```
TBApplicationRootComponent >> children
   ^ { main }
```

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

6.3 Pattern de définition de composants

Nous allons souvent utiliser la même façon de procéder:

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

6.4 Ajouter quelques bulletins au blog

Vérifiez que votre blog a quelques bulletins :

```
TBBlog current allBlogPosts size
```

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

```
TBBlog createDemoPosts
```

6.5 Définition du composant TBHeaderComponent

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

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

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

6.6 Utilisation du composant header

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

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

Créons une méthode `initialize` dans le protocole 'initialization' :

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

```
TBScreenComponent >> createHeaderComponent
   ^ TBHeaderComponent new
```

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

6.7 Relation composite-composant

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

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

6.8 Rendu visuel de la barre de navigation

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

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

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

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

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

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

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

Améliorations possibles

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

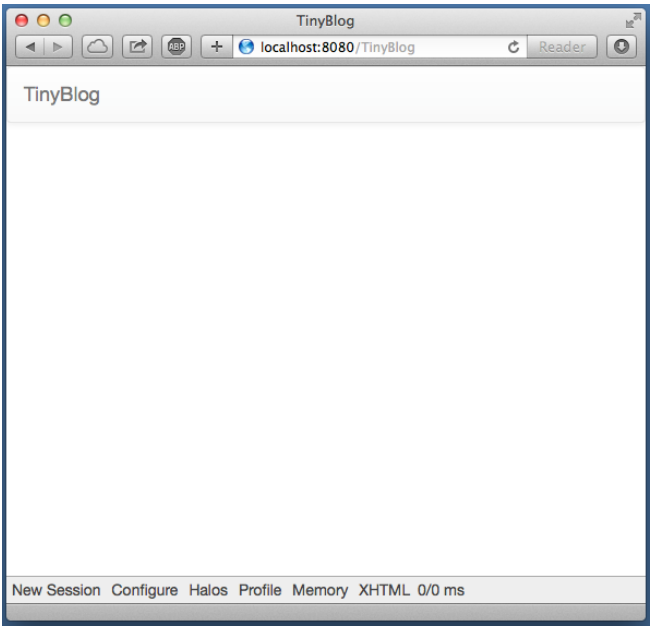


Figure 6-5 TinyBlog avec une barre de navigation.

6.9 Liste des posts

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

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

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

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

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

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

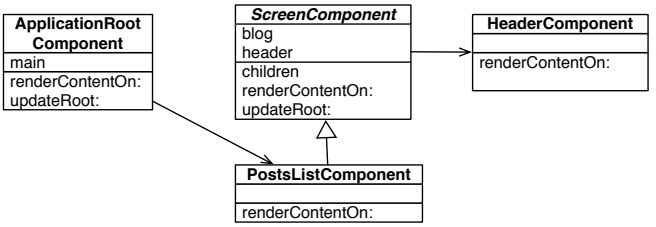


Figure 6-6 Le composant `ApplicationRootComponent` utilise le composant `PostsListComponent`.

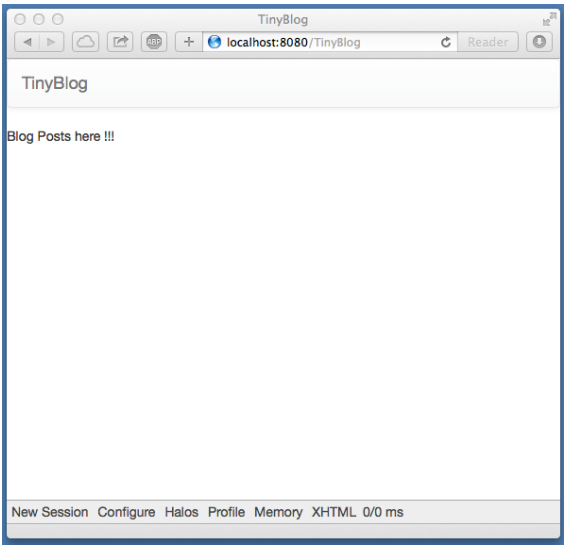


Figure 6-7 TinyBlog avec une liste de bulletins plutot élémentaire.

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

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

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

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

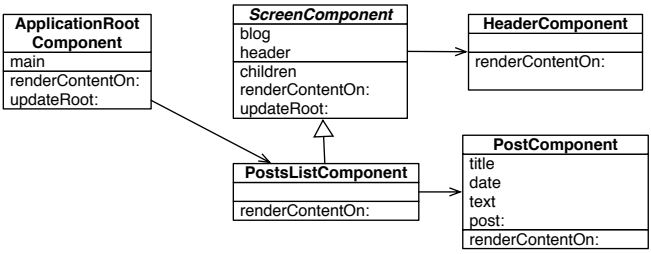


Figure 6-8 Ajout du composant Post.

6.10 Le composant Post

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

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

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

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

```
TBPostComponent >> title
   ^ post title
```

```
TBPostComponent >> text
   ^ post text
```

```
TBPostComponent >> date
   ^ post date
```

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

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

A propos des formulaires

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

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

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

6.11 Afficher les bulletins (posts)

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

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

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

6.12 Débugger les erreurs

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

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

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

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

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

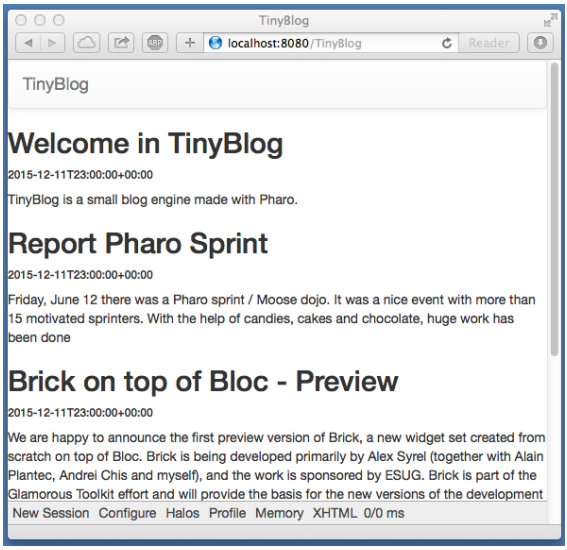


Figure 6-9 TinyBlog avec une liste de posts.

6.13 Affichage de la liste des posts avec Bootstrap

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

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

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

6.14 Cas d'instanciation de composants dans renderContentOn:

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

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

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

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

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

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

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

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

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

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

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

6.15 Conclusion

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

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

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

CHAPTER 7

Gestion des catégories

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

Les instructions pour charger le code des chapitres précédents sont disponibles dans le chapitre 13.

7.1 Affichage des bulletins par catégorie

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

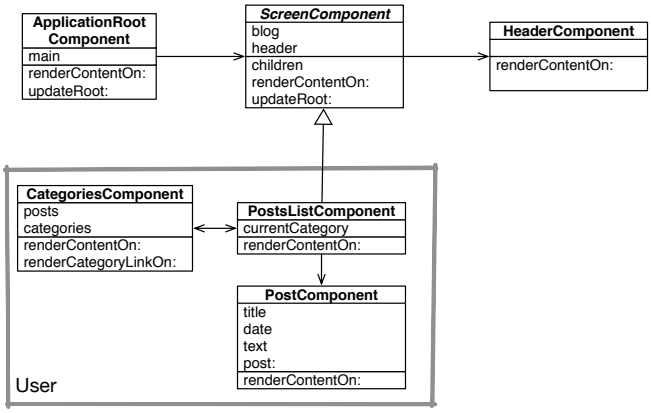


Figure 7-1 L'architecture des composants de la partie publique avec catégories.

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

Pour afficher les catégories

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

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

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

Definition du composant

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

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

Nous définissons les accesseurs associés.

```
TBCategoriesComponent >> categories
   ^ categories
```

```
TBCategoriesComponent >> categories: aCollection
   categories := aCollection asSortedCollection
```

```
TBCategoriesComponent >> postsList: aComponent
      postsList := aComponent
```

```
TBCategoriesComponent >> postsList
   ^ postsList
```

Nous définissons aussi une méthode de création au niveau classe.

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

Liaison depuis la liste de bulletins

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

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

Nous définissons les accesseurs associés.

```
TBPostsListComponent >> currentCategory
   ^ currentCategory
```

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

La méthode selectCategory:

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

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

7.2 Rendu des catégories

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

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

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

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

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

7.3 Mise à jour de la liste des bulletins

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

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

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

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

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

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

Améliorations possibles

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

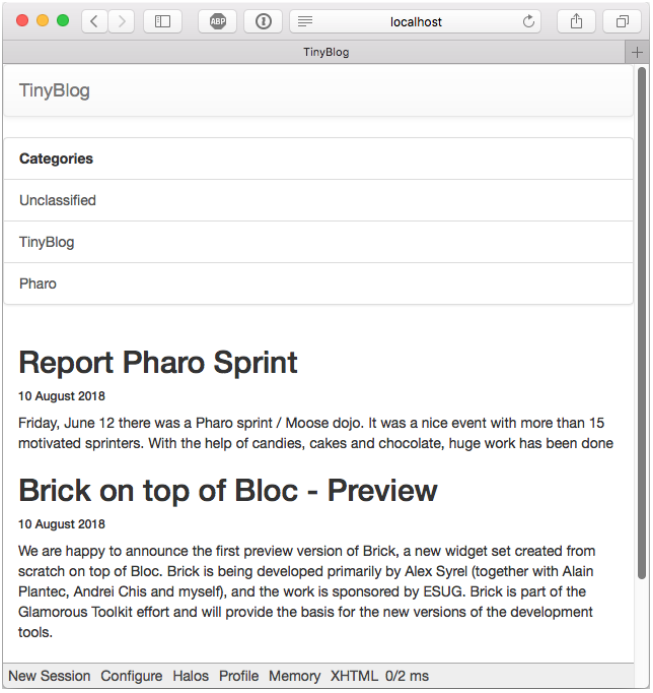


Figure 7-2 Catégories afin de sélectionner les posts.

7.4 Look et agencement

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

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

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

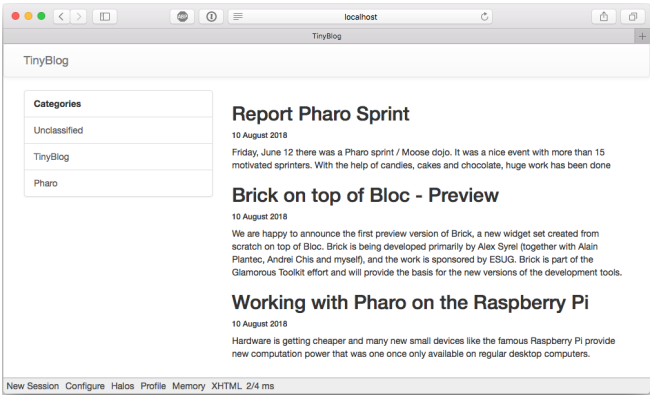


Figure 7-3 Liste des catégories avec un meilleur agencement.

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

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

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

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

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

7.5 Modulariser son code avec des petites méthodes

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

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

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

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

Autre passe de découpage

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

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

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

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

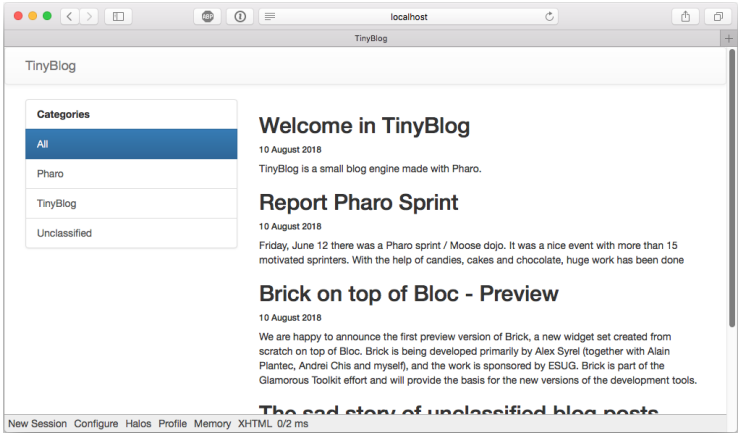


Figure 7-4 TinyBlog UI version finale.

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

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

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

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

7.6 Conclusion

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

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

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

Améliorations possibles

A titre d'exercice, vous pouvez :

- trier les catégories par ordre alphabétique ou
- ajouter un lien nommé 'All' dans la liste des catégories permettant d'afficher tous les bulletins visibles quelque que soit leur catégorie.

CHAPTER 8

Authentification et Session

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

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

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

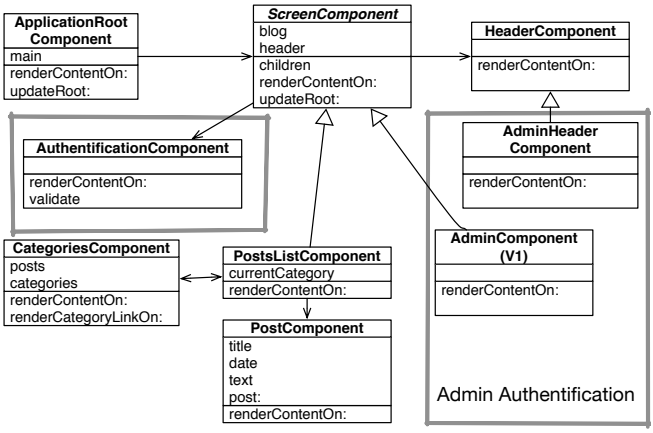


Figure 8-1 Gérant l'authentification pour accéder à l'administration.

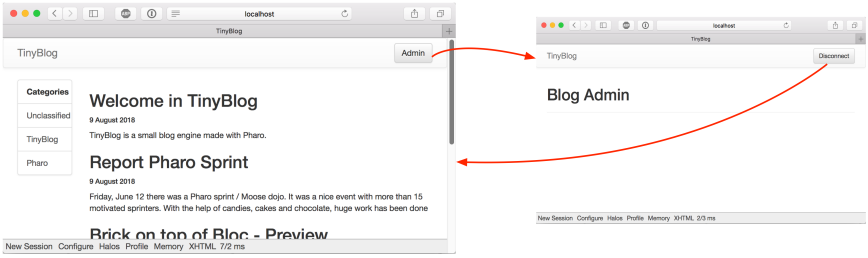


Figure 8-2 Lien simple vers la partie administration.

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

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

8.1 Composant d'administration simple (v1)

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

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

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

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

8.2 Ajout d'un bouton 'admin'

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

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

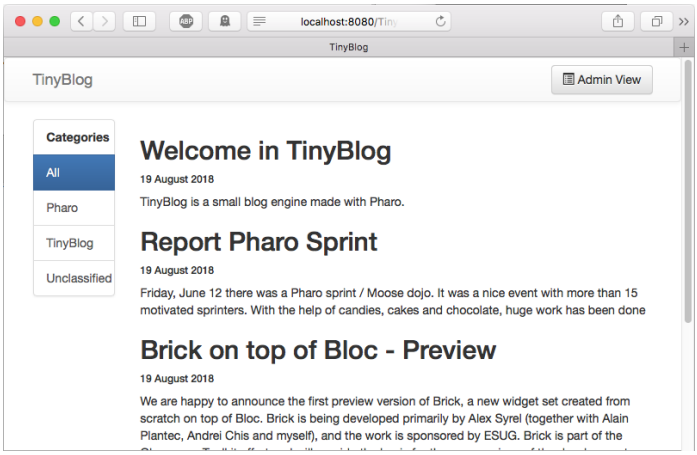


Figure 8-3 Barre de navigation avec un button admin.

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

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

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

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

8.3 Revisons la barre de navigation

Commençons par réviser la définition de `TBHeaderComponent` en lui ajoutant une variable d'instance `component` pour stocker et accéder au composant courant (qui sera soit la liste de bulletins, soit le composant d'administration). Ceci va nous permettre de pouvoir accéder au composant depuis la

barre de navigation :

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

```
TBHeaderComponent >> component: anObject
   component := anObject

TBHeaderComponent >> component
   ^ component
```

Nous ajoutons une méthode de classe.

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

8.4 Activation du bouton d'admin

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

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

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

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

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

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

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

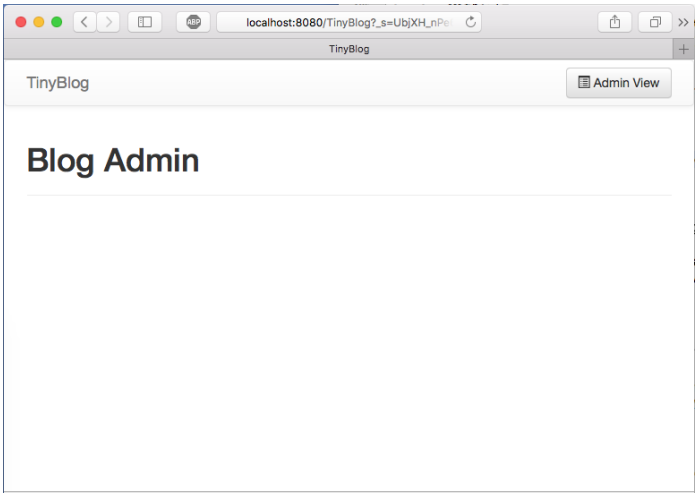


Figure 8-4 Affichage du composant admin en cours de définition.

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

8.5 Ajout d'un bouton 'disconnect'

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

Définissons un nouveau composant en-tête:

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

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

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

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

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

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

```
TBAdminComponent >> goToPostListView
  self answer
```

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

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

Notion call:/answer:

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

8.6 Composant fenêtre modale d'identification

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

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

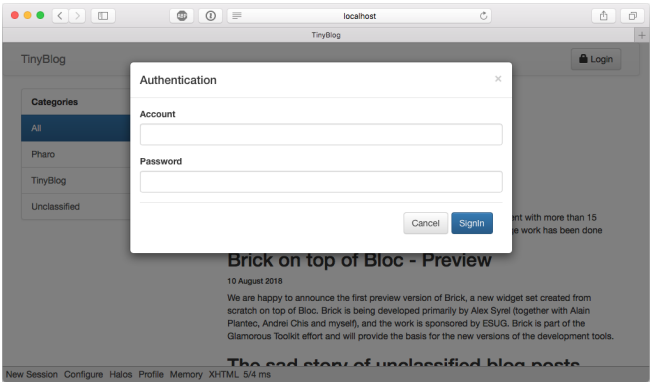


Figure 8-5 Aperçu du composant d'identification.

Définition d'un composant d'identification

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

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

```
TBAuthentificationComponent >> account
   ^ account
```

```
TBAuthentificationComponent >> account: anObject
   account := anObject
```

```
TBAuthentificationComponent >> password
   ^ password
```

```
TBAuthentificationComponent >> password: anObject
   password := anObject
```

```
TBAuthentificationComponent >> component
   ^ component
```

```
TBAuthentificationComponent >> component: anObject
   component := anObject
```

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

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

8.7 Rendu du composant d'identification

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

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

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

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

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

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

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

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

Le même procédé est utilisé pour le mot de passe.

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

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

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

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

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

8.8 Intégration du composant d'identification

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

```
TBHeaderComponent >> renderButtonsOn: html
   self renderModalLoginButtonOn: html
```

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

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

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

8.9 Gestion naive des logins

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

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

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

8.10 Gestion des erreurs

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

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

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

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

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

Nous ajoutons une méthode pour tester cet état.

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

Nous définissons aussi un message d'erreur.

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

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

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

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

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

8.11 Modélisation des administrateurs

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

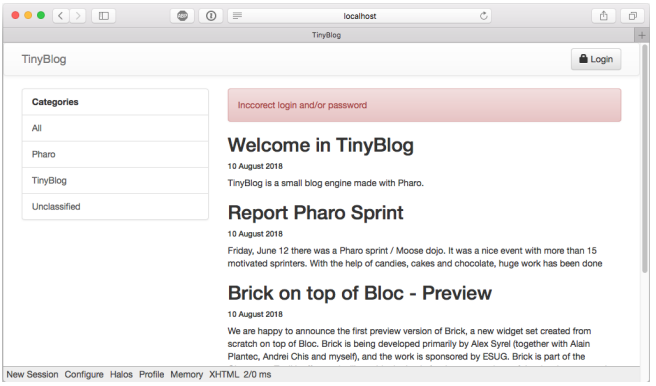


Figure 8-6 Message d'erreur en cas d'identifiants erronnés.

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

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

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

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

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

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

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

Nous définissons aussi une méthode de création.

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

Vous pouvez vérifier cela en inspectant l'expression suivante :

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

8.12 Administrateur pour un blog

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

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

```
TBBlog >> administrator
   ^ adminUser
```

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

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

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

Maintenant nous pouvons créer un administrateur par défaut.

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

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

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

8.13 Définir un administrateur

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

```
  TBBlog reset; createDemoPosts
```

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

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

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

Améliorations possibles

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

8.14 Intégration du compte administrateur

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

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

8.15 Stocker l'administrateur courant en session

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

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

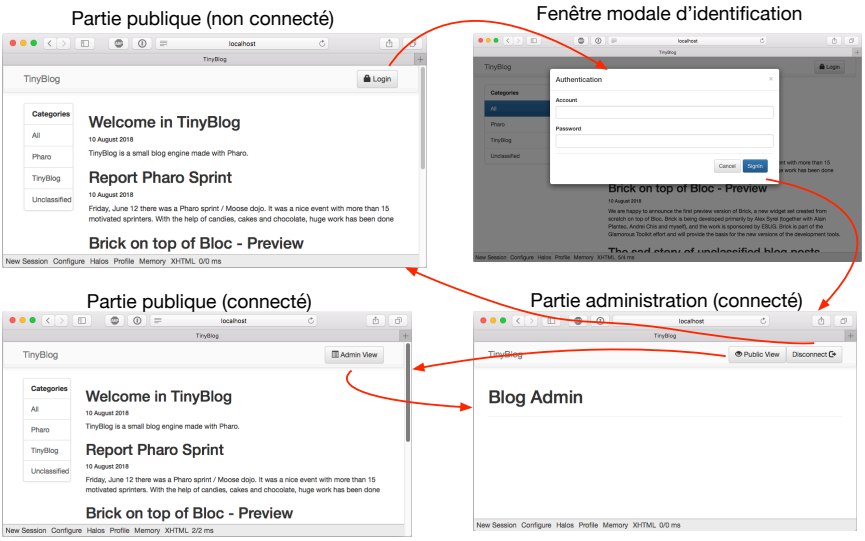


Figure 8-7 Navigation et identification dans TinyBlog.

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

8.16 Définition et utilisation d'une classe session spécifique

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

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

```
TBSession >> currentAdmin
    ^ currentAdmin
```

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

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

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

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

fectue dans la méthode `initialize` de la classe `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: JQUiDeploymentLibrary;
         addLibrary: TBSDeploymentLibrary
```

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

8.17 Stockage de l'administrateur courant en session

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

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

8.18 Navigation simplifiée vers la partie administration

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

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

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

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

8.19 Déconnexion

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

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

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

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

Le bouton 'Disconnect' fonctionne à nouveau correctement.

8.20 Navigation simplifiée vers la partie publique

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

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

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

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

8.21 Conclusion

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

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

Améliorations possibles

A titre d'exercice, vous pouvez :

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

CHAPTER 9

Interface web d'administration et génération automatique

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

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

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

9.1 Décrire les données métiers avec Magritte

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

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

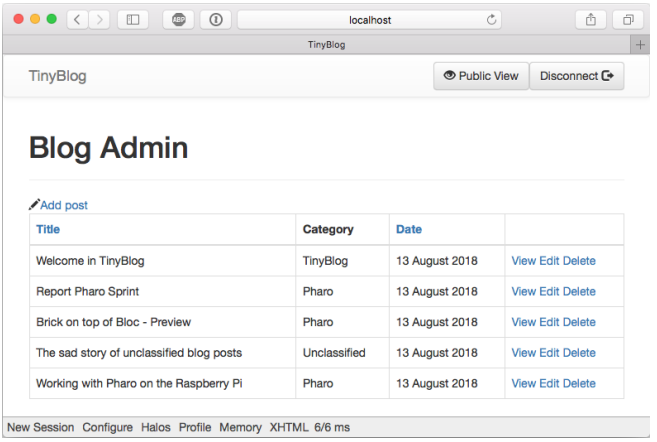


Figure 9-1 Gestion des bulletins.

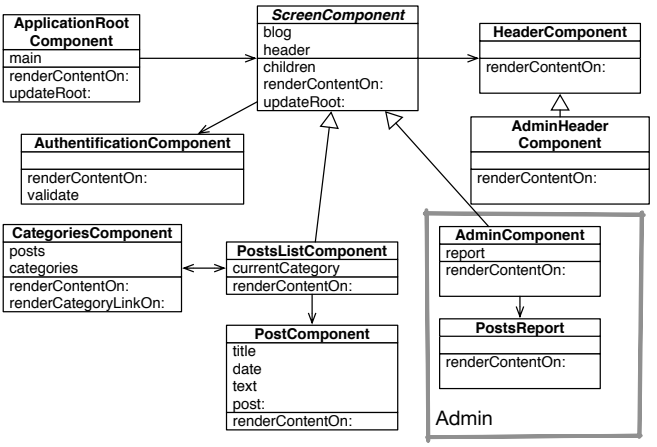


Figure 9-2 Composants pour l'administration.

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

9.2 Description d'un bulletin

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

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

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

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

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

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

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

```
TBPost >> descriptionCategory
   <magritteDescription>
   ^ MAStringDescription 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>
   ^ MADateDescription new
      accessor: #date;
      beRequired;
      yourself
```

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

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

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

9.3 Création automatique de composant

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

```
aTBPost asComponent
```

Nous allons voir comment utiliser cela dans la suite.

9.4 Mise en place d'un rapport des bulletins

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

Le composant PostsReport

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

Pour rester modulaire, nous allons créer un composant Seaside pour cette tâche. Le composant `TBPostsReport` étend la classe `TBSMagritteReport` qui gére les rapports avec Bootstrap.

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

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

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

9.5 Intégration de PostsReport dans AdminComponent

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

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

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

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

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

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

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

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

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

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

Vous pouvez déjà tester dans votre navigateur.

9.6 Filtrer les colonnes

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

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

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

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

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

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

9.7 Amélioration du rapport

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

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

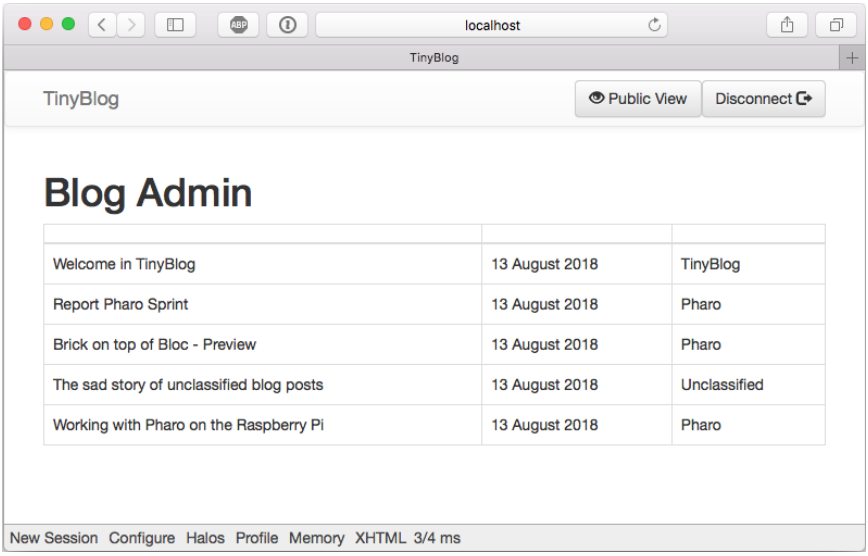


Figure 9-3 Rapport Magritte contenant les bulletins du blog.

```
      yourself
```

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

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

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

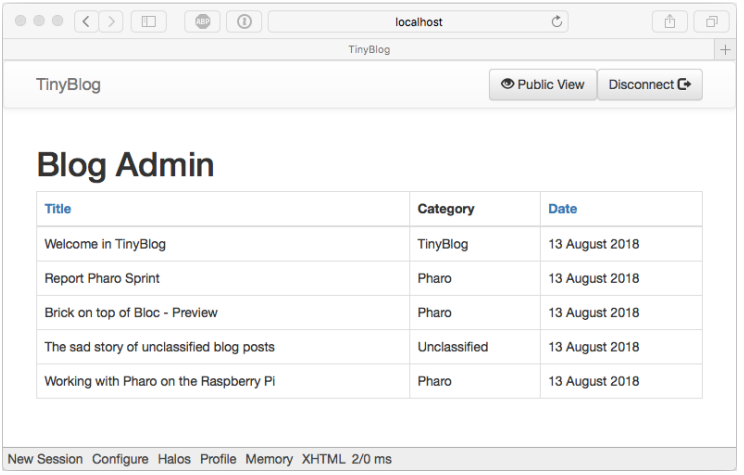


Figure 9-4 Administration avec un rapport.

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

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

9.8 Administration des bulletins

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

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

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

```
TBSMagritteReport >> blog
   ^ blog
```

```
TBSMagritteReport >> blog: aTBBlog
   blog := aTBBlog
```

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

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

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

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

9.9 Gérer l'ajout d'un bulletin

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

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

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

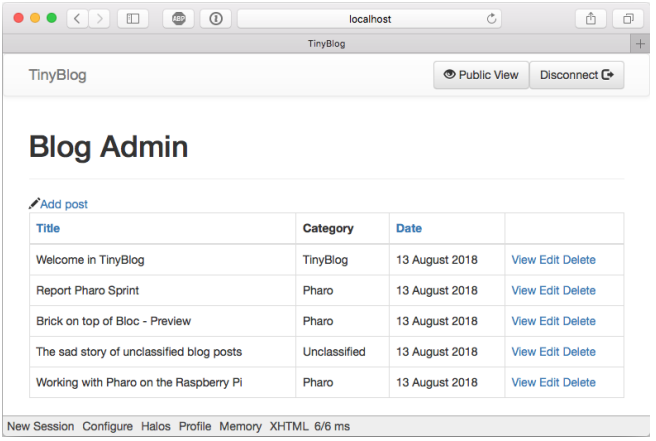


Figure 9-5 Rapport des bulletins avec des liens d'édition.

9.10 Implémentation des actions CRUD

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

9.11 Ajouter un bulletin

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

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

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

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

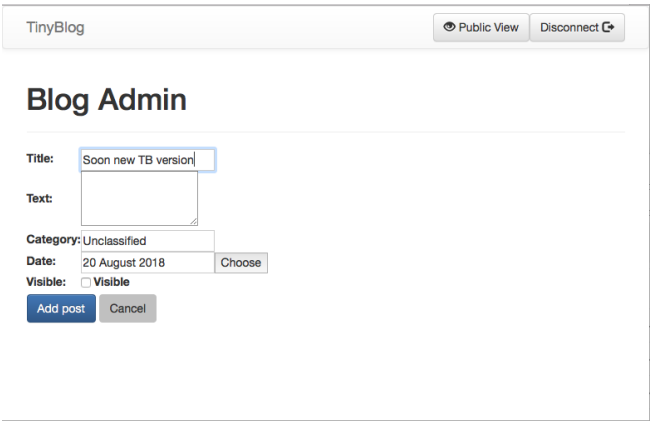


Figure 9-6 Affichage rudimentaire d'un bulletin.

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

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

Afficher un bulletin

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

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

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

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

Editer un bulletin

Pour éditer un bulletin nous utilisons la même approche.

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

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

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

Supprimer un bulletin

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

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

ainsi qu'un test unitaire :

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

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

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

9.12 Gérer le rafraîchissement des données

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

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

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

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

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

9.13 Amélioration de l'apparence des formulaires

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

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

```
TBPost >> descriptionContainer
    <magritteContainer>
    ^ super descriptionContainer
        componentRenderer: TBSMagritteFormRenderer;
        yourself
```

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

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

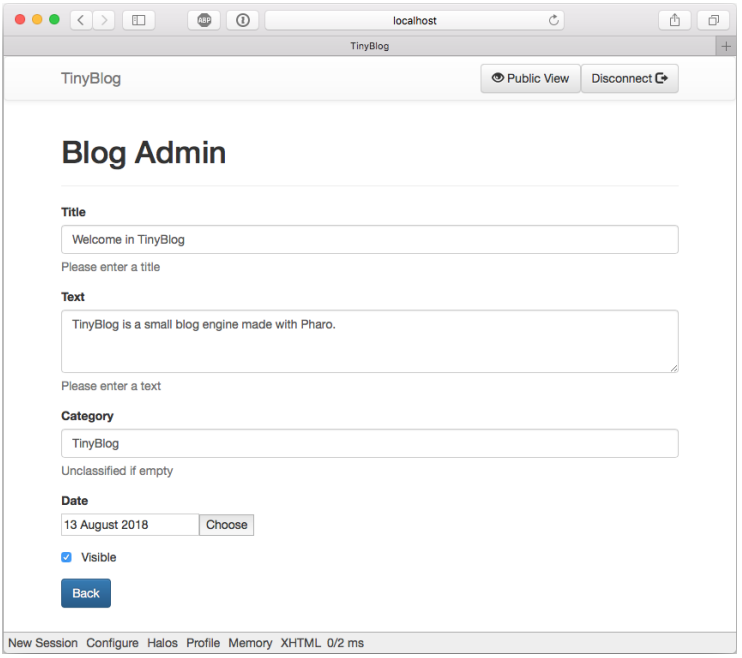


Figure 9-7 Formulaire d'ajout d'un post avec Bootstrap.

```
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>
    ^ MAStringDescription new
        label: 'Category';
        priority: 300;
        accessor: #category;
        comment: 'Unclassified if empty';
        componentClass: TBSMagritteTextInputComponent;
        yourself
```

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

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

9.14 Conclusion

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

CHAPTER 10

Déploiement de TinyBlog

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

10.1 Déployer dans le cloud

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

10.2 Hébergement sur PharoCloud

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

Préparer son compte PharoCloud :

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

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

10.3 Préparation de l’image Pharo pour PharoCloud

Repartir d’une image vierge

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

Configuration de Seaside

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

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

Chargement de TinyBlog

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

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

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

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

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

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

TinyBlog comme application par défaut

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

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

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

Lancer Seaside :

```
  "Start HTTP server"
  ZnZincServerAdaptor startOn: 8080.
```

Save your image

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

10.4 Déploiement manuel sur Ephemeric Cloud de PharoCloud

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

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

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

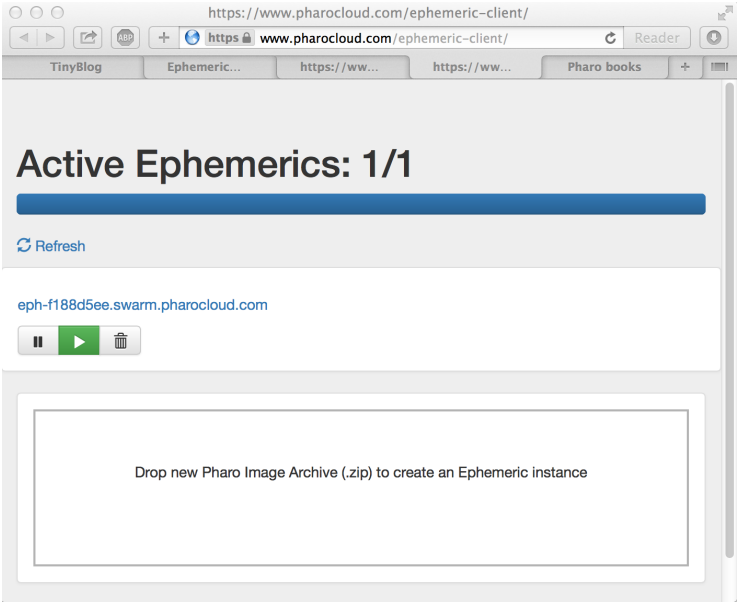


Figure 10-1 Administration des images Pharo sur Ephemeric Cloud.

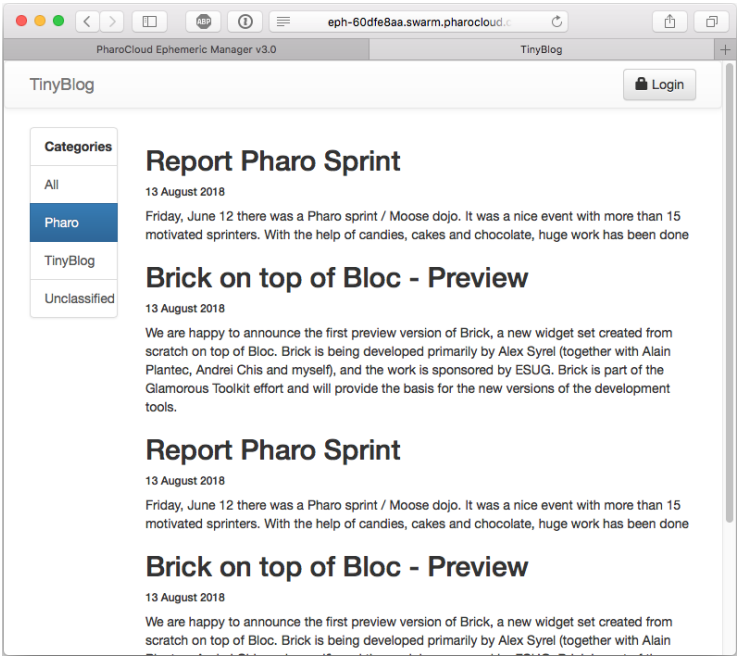


Figure 10-2 Votre application TinyBlog sur PharoCloud.

10.5 Déploiement automatique sur Ephemeric Cloud de PharoCloud

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

```
|client EPHUSER EPHTOKEN|

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

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

10.6 A propos de dépendances

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

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

Part II

Eléments optionnels

CHAPTER 11

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.

11.1 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.

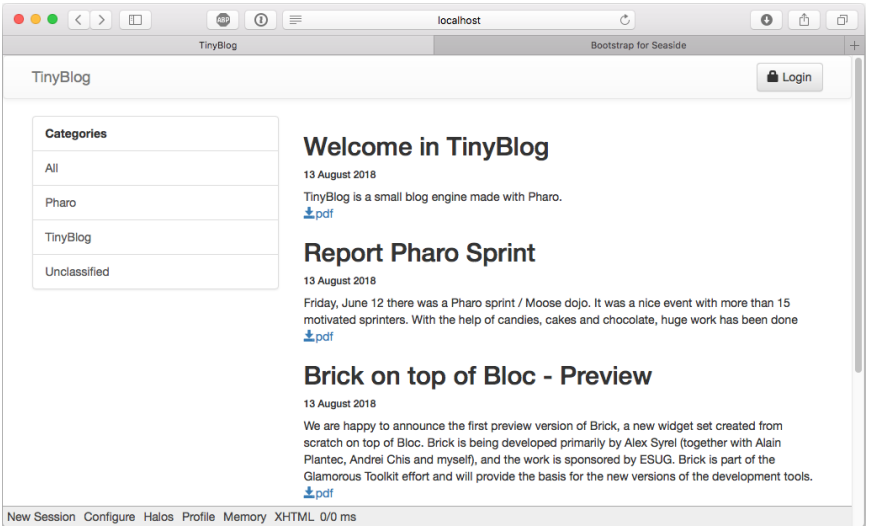


Figure 11-1 Chaque post peut être exporté en PDF

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:`.

```
TBPostComponent >> renderContentOn: html
   html paragraph class: 'title'; with: self title.
   html paragraph class: 'subtitle'; with: self date.
   html paragraph class: 'content'; with: self text.
   html div
      with: [
         html anchor
         callback: [ self exportPostAsPdf ];
         with: [
         html tbsGlyphIcon iconSave.
         html text: 'pdf' ] ].
```

La figure 11-1 montre le lien d'export en PDF ajouté pour chacun des posts.

Ajoutons maintenant la méthode de callback:

```
TBPostComponent >> exportPostAsPdf
   | pdfStream |
   pdfStream := TBPostPDFExport post: post.
   self requestContext respond: [:response |
      response
       contentType: 'application/pdf; charset=UTF-8';
       attachmentWithFileName: post title, '.pdf';
       binary;
       nextPutAll: pdfStream contents ]
```

Lorsque l'utilisateur clique sur le lien, une instance de la classe `TBPostPDF-Export` est créée. Cette classe aura la responsabilité de construire le document PDF à partir du bulletin courant. Ce document est ensuite envoyé à l'utilisateur grâce au contexte HTTP.

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 `pdfdoc` 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. Les méthodes `exportPdf` et `renderPostAsPdfInto:` produisent ensuite le document PDF .

```
TBPostPDFExport class >> post: aPost
  ^ self new
    post: aPost;
    exportPdf
```

```
TBPostPDFExport >> exportPdf
  | pdfStream |
  pdfStream := MultiByteBinaryOrTextStream on: String new.
  self renderPostAsPdfInto: pdfStream.
  ^ pdfStream reset
```

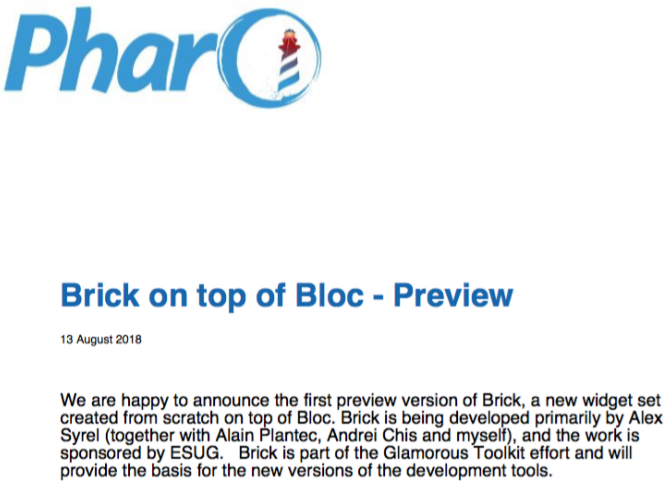
```
TBPostPDFExport >> renderPostAsPdfInto: aStream
   | aPage titleFont titleColor layout pharoLogo metaDataColor
    defaultFont |
   pharoLogo := Morph new
```

```
    extent: PolymorphSystemSettings pharoLogo extent;
    color: Color white;
    addMorph: PolymorphSystemSettings pharoLogo.

pdfdoc := PDFDocument new.

titleColor := PDFColor r: 13 g: 100 b: 175.
titleFont := PDFHelveticaFont new
    fontSize: 22 pt;
    bold: true.

metaDataColor := PDFColor greyLevel: 0.3.

defaultFont := PDFHelveticaFont new
    fontSize: 12 pt ; yourself.

aPage := PDFPage new.
aPage add: ((PDFPngElement fromMorph: pharoLogo)
    from: 10 mm @ 20 mm;
    dimension: 80mm @ 27mm).

layout := PDFVerticalLayout on: {
    (PDFFormattedTextElement new
        font: titleFont;
        textColor: titleColor;
        text: post title).

    (PDFFormattedTextElement new
        textColor: metaDataColor;
        text: post date asString).

    (PDFParagraphElement new
        dimension: 150 mm @ 35 mm;
        font: defaultFont;
        text: post text ) }.

layout from: 25 mm @ 80 mm.
layout spacing: 1 cm.
aPage add: layout.
pdfdoc add: aPage.
pdfdoc exportTo: aStream
```

La figure 11-2 montre le résultat d'un export en PDF d'un bulletin.

11.2 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



Brick on top of Bloc - Preview

13 August 2018

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.

Figure 11-2 Résultat du rendu PDF d'un bulletin

(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 formatés 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.

11.3 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
```

11.4 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 necessaire (titre et contenu), NeoCSV utilise des guillemets pour indiquer que la donnée est une chaine 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)
```

```
  ]
```

11.5 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 installer 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'un 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
```

11.6 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 `TBPost` 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 `enabledPrettyPrinting` 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 une en-tête 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 yyyymmdd).
            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.

11.7 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.

CHAPTER 12

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'`.

12.1 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'en-tête 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.

12.2 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 de 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. C'est en 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 WARestfulFilter.

```
WARestfulFilter 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: JQUiDeploymentLibrary;
        addLibrary: TBSDeploymentLibrary.

    app addFilter: TBRestfulFilter new.
```

N'oublier 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.

12.3 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:>`.

```
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 object 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
```

12.4 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'éxecution 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 capables 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 réponses 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 chaine 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`.

12.5 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 comme nous l'avons vu dans le chapitre précédent. Nous vous laissons le soin d'ajouter ces formats aux services REST de TinyBlog.

12.6 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.

12.7 Utiliser un service REST

Il existe plusieurs façons d'utiliser ce service REST.

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 tout système 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.

12.8 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 chaine de caractères comme unique argument et recherche un post ayant un titre identique à la chaine de caractères. Si plusieurs posts sont trouvés, la méthode retourne le premier sélectionné.

```
TBBlog >> 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 `TBRestService-Search` qui hérite de `TBRestService`. Cet objet a besoin de connaître le titre du post recherché et fait appel à la méthode TBBlog » postWithTitle: définie précedemment.

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

```
TBRestServiceSearch >> title
  ^ title
```

```
TBRestServiceSearch >> title: anObject
  title := anObject
```

```
TBRestServiceSearch >> execute
  | post |

  post := TBBlog 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 appliquée à la chaine 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 caractères "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'en-tête 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
```

12.9 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` convertie les deux chaines 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 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).

12.10 Ajouter un post

Voyons maintenant comment ajouter un nouveau post à notre blog à l'aide de REST. Etant 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 servie 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'en-tête 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 local `post`. Il suffit ensuite d'instancier un `TBPost` et de le sauver 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 guise 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'expliciter la cause du problème.

12.11 Améliorations 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.

CHAPTER 13

Charger le code des chapitres

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

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

13.1 Chapitre 3 : Extension du modèle et tests unitaires

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

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

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

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

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

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

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

13.3 Chapitre 5 : Commencer avec Seaside

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

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

Exécutez les tests.

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

```
ZnZincServerAdaptor startOn: 8080.
```

Ouvrez votre browser sur `http://localhost:8080/TinyBlog`

Si vous avez besoin de créer quelques posts initiaux:

```
TBBlog reset ; createDemoPosts
```

13.4 Chapitre 6 : Des composants web pour TinyBlog

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

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

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

```
ZnZincServerAdaptor startOn: 8080.
```

Ouvrez votre browser sur `http://localhost:8080/TinyBlog`

Si vous avez besoin de créer quelques posts initiaux:

```
TBBlog reset ; createDemoPosts
```

13.5 Chapitre 7 : Gestion des catégories

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

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

Même process que le précédent.

13.6 Chapitre 8 : Authentification et Session

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

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

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

```
ZnZincServerAdaptor startOn: 8080.
```

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

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

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

13.8 Chapitre 10 : Déploiement de TinyBlog

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

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

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

```

ephUser :='<your PharoCloud Ephemeric login>'.
ephToken :='<your PharoCloud Ephemeric passwod>'.

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

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

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

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

"Register TinyBlog on Seaside"
TBApplicationRootComponent initialize.

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

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

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

CHAPTER 14

Sauver votre code

Lorsque vous sauvez l'image Pharo avec le menu 'Save', celle-ci contient tous les objets du système et donc les classes elles-mêmes. Cette solution est pratique mais peu pérenne. Jusqu'à Pharo 60, Pharo nous propose d'utiliser Monticello et SmalltalkHub. A partir de Pharo 70, vous pouvez sauver votre code sur github ou gitlab avec Iceberg.

14.1 Jusqu'a Pharo 6o

Nous allons vous montrer comment les pharoers sauvent leur code sous forme de packages sur un serveur dédié à l'aide de Monticello: le gestionnaire de versions de Pharo.

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

Créer un dépôt de code

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

- Créer un compte sur le site http://smalltalkhub.com/.
- Se connecter au site.

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

Sauver votre package

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

14.2 En Pharo 70

Nous vous suggèrons de lire le chapitre dédié à la gestion de code dans le livre "Managing Your Code with Iceberg" (disponible à http://books.pharo.org).

Ici nous rappellons les points clefs :

- Créer un projet sur http://www.github.com.
- Utiliser Iceberg puis ajouter un projet, choisir clone from github par exemple.
- Créer un dossier `'src'` avec le FileList ou la ligne de commande dans le dossier que vous avez choisi sur votre système de fichier local.
- Ouvrir votre projet et ajouter vos packages (Il est conseillé de definir une baseline pour être facilement rechargeable).
- Commiter le code.
- Publier votre code sur github.