TinyBlog: Un Tutorial Web en Pharo

Olivier Auverlot and Stéphane Ducasse

February 16, 2016

CHAPTER 1

TinyBlog: Préparation

Ce tutoriel va vous enseigner comment définir et déployer une application en utilisant Pharo/Seaside/Mongo ainsi que des frameworks disponibles en Pharo comme NeoJSON. Nous exposerons aussi comment exposer votre application via un serveur REST. Nous allons définir un mini moteur de blogs. Les solutions proposées dans le tutoriel sont parfois non optimales afin de vous faire réagir et que vous puissiez proposer d'autres solutions et améliorations.

D'autre part, notre objectif n'est pas d'être exhaustif. Nous montrons une façon de faire cependant nous invitons le lecteur à lire les références sur les autres chapitres, livres et tutoriaux afin d'approfondir son expertise.

1.1 Installation et préparation de Pharo

Créer un projet sur SmalltalkHub

- Depuis votre compte créer un projet sur http://smalltalkhub.com
  - Nommez le "TinyBlog",
  - Récupérer l'URL du projet: 'http://smalltalkhub.com/mc/XXX/TinyBlog/main'

Mettre en place Pharo

- Télécharger Pharo 4.0 à partir du site pharo.org
- A partir du "Configuration Browser", installer les paquets:
  - Seaside3,

  - VoyageMongo,
  - BootstrapMagritte,
  - Mustache
- Créez un paquet nommé TinyBlog

Nous avons préparer une image contenant les projets Seaside, Voyage, Bootstrap et Magritte. Vous pouvez y accéder en utilisant le Pharo Launcher et en cherchant le projet TinyBlogBase dans les contributions Pharo. Sinon cette image se trouve sur le serveur d'integration http://ci.inria.fr dans les Pharo-Contributions et le projet TinyBlogBase.

Créer la configuration du projet

Dans un premier temps vous pouvez sauter cette étape.

- Créer un projet avec Versionner
  - Créer un nouveau projet "TinyBlog",
  - Dans development, ajoutez les paquets dont votre projet dépend:
    * Seaside3,
    * VoyageMongo,
    * BootstrapMagritte,
    * Mustache
  - Dans Packages, ajoutez le paquet TinyBlog,
    * Définissez son repository: 'http://smalltalkhub.com/mc/XXX/TinyBlog/main'
    * Cliquer sur le bouton "Save to development"

Démarrer le serveur HTTP

- Aller dans Tools pour ouvrir le Seaside Control Panel,
- Faire un clic droit dans la partie supérieure et sélectionner le ZnZinServerAdaptor,
- Choisir le port 8080,
- Cliquer sur le serveur pour le sélectionner et cliquez sur le bouton "Start".

CHAPTER 2

Le modèle de TinyBlog

Le modèle de TinyBlog est extrêmement simple: Les deux classes `TBPost` et `TBBlog`. Notez que comme nous le verrons plus tard, nous allons utiliser le fait que Voyage offre une base de données en mémoire pour développer la solution sans avoir besoin de se connecter en permanence à la base.

La classe TBPost

Définissez la class `TBPost` comme suit:

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

Description d'un post

Cinq variables d'instance pour décrire un post sur le blog.

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

Ces variables ont des méthodes accèsseurs dans le protocole 'accèssing'.

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

```
TBPost >> title: anObject
   title := anObject

TBPost >> text
   ^ text

TBPost >> text: anObject
   text := anObject

TBPost >> date
   ^ date

TBPost >> date: anObject
   date := anObject

TBPost >> visible
   ^ visible

TBPost >> visible: anObject
   visible := anObject

TBPost >> category
   ^ category

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

Gérer la visibilité d'un post

Il faut avoir la possibilité d'indiquer qu'un post est visible ou pas. Il faut également pouvoir demander à un post s'il est visible. Les méthodes sont définies dans la protocole 'action'.

```
TBPost >> beVisible
   self visible: true

TBPost >> notVisible
   self visible: false

TBPost >> isVisible
   ^ self visible
```

Initialisation

La méthode `initialize` (protocole 'initialize-release') fixe par défaut la date et la visibilité à faux (l'utilisateur devra par la suite activer la visibilité ce qui permet de rédiger des brouillons et de publier lors que le post est terminé).

Un post est également rangé par défaut dans la catégorie 'Unclassified' que l'on définit au niveau classe. La méthode `unclassifiedTag` renvoie une valeur indiquant que le post n'est pas rangé dans une catégorie.

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

TBPost >> initialize
  self category: TBPost unclassifiedTag.
  self date: Date today.
  self visible: false.
```

Amélioration

Il serait préférable de ne pas faire une référence en dur à la class `TBPost` comme dans la méthode `initialize`. Proposer une solution.

Méthode de création

Coté classe, on définit que `TBPost` est un objet géré par Voyage et on spécifie des méthodes pour faciliter la création de post appartenant ou pas à une catégorie.

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

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

Test

Savoir si un post est classé dans une catégorie

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

Il serait préférable de ne pas faire une référence en dur à la class `TBPost`. Proposer une solution.

La classe TBBlog

La classe `TBBlog` contient des posts. Nous allons développer `TBBlog` en écrivant des tests puis les implémentant.

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

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

Une seule base

Dans un premier temps nous supposons que nous allons gérer qu'un seul blog. Dans le future, nous ajouterons la possibilité de gérer plusieurs blogs comme un par utilisateur de notre application. Pour l'instant nous utilisons donc un singleton pour la class `TBBlog`.

```
TBBlog class
   instanceVariableNames: 'uniqueInstance'

TBBlog class >> reset
   uniqueInstance := nil

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

TBBlog class >> initialize
   self reset
```

2.1 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 premiers puis en définissant les fonctionalités jsuqu'à ce que les tests passent.

Les tests unitaires sont regroupés dans le paquet `TinyBlog-Tests` qui contient la classe `TBBlogTest`.

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

Avant le lancement des tests, la méthode `setUp` initialise la connexion vers la base, efface son contenu, ajoute un post et en crée un autre qui provisoirement n'est pas enregistré.

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

   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
```

On en profite pour tester différentes configuration. Les posts ne sont pas dans la même catégorie, l'un est visible, l'autre pas.

La méthode `tearDown` exécutée au terme des tests remet à zéro la connexion.

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

Noter que si vous déployer un blog puis exécuter les tests vous perdrez les postes que vous avez crées car nous les remettons à zéro.

Nous allons développer les tests d'abord puis ensuite passer à l'implémentation des fonctionalités.

Un premier test

Commençons par écrire un premier test qui ajoute un post.

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

Ce test ne marche pas car nous n'avons pas défini `writeBlogPost:` et `remove`.

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

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

TBBlog >> size
   ^ posts size
```

Le test précédent doit maintenant passer.

Ecrivons un test pour couvrir les fonctionalités que nous venons de développer.

Obtenir le nombre de posts dans la base

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

Effacer l'intégralité des posts

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

Quelques autres fonctionalités

Nous définissons les fonctionalités et nous assurons que les tests passent. Les règles métiers sont regroupées dans le protocole 'action' de la classe `TBBlog`.

Obtenir l'ensemble des posts (visibles et invisibles)

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

TBBlog >> allBlogPosts
   ^ posts
```

Obtenir tous les posts visibles

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

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

Obtenir tous les posts d'une catégorie

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

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

Obtenir tous les posts visibles d'une catégorie

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

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

Vérifier la gestion des posts non classés

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

Obtenir la liste des catégories

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

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

2.2 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 contenus, renommer un catégorie, déplacer un post d'une catégorie à une autre, rendre visible ou invisible une catégorie et son contenu, etc. Nous vous encourageons à les développer.

Afin de nous aider à tester l'application nous définissons quelques postes.

```
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)
```

CHAPTER 3

Préparation de sauvegarde extérieure avec Voyage

Alors qu'avoir un modèle d'objets en mémoire fonctionne bien, et que des sauvegardes de l'image Pharo sauve aussi de tels objets, il est préférable de pouvoir sauver les objets dans une base de données extérieure. Voyage permet de sauver les objets dans une base de donnée Mongo. C'est ce que nous allons voir. Dans un premier temps nous allons utiliser la capacité de Voyage à simuler une base extérieure. Ceci est très pratique en phase de développement. Dans un second nous installerons une base de données Mongo et nous y accéderons via la couche Voyage.

3.1 Sauvegarde d'un blog

La première méthode à modifier est la méthode `writeBlogPost:` pour sauver le blog.

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

Si vous essayez d'exécuter le test suivant, vous allez voir qu'il ne passe pas car le système rend 0 au lieu de 2 instances de posts.

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

Pourquoi cela arrive-t-il? En fait nous n'avons jamais dit quels objets devaient être sauvés dans la base de données.

La méthode de class `isVoyageRoot` permet de déclarer que les objets de la classe `TBBlog` doivent être sauvés dans la base.

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

De plus nous devons soit créer une connexion sur une base de données réelle soit travailler en mémoire. C'est cette dernière option que nous faisons maintenant en utilisant cette expression.

```
VOMemoryRepository new enableSingleton.
```

Le message `enableSingleton` indique à Voyage que nous n'utilisons qu'une seule base de donnée ce qui nous permet de ne pas avoir à préciser avec laquelle nous travaillons. `enableSingleton` met à jour le repository.

Nous définissons une méthode `initializeMongo` dont le role est d'initialiser correctement la base.

```
TBBlog class >> initializeMongo
   | repository |
   repository := VOMemoryRepository new.
   repository enableSingleton.
```

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

Nous redéfinissons la méthode `reset` et la méthode `initialize` pour nous assurer que l'endroit où la base est sauvée est réinitialisé lorsque que l'on charge le code.

La méthode `reset` réinitialize la base.

```
TBBlog class >> reset
     "self reset"
     self initializeMongo

TBBlog class >> initialize
     self initializeMongo
```

Le cas de 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 ]
```

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

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

3.2 Utilisation de la base

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

Nous créons des posts dans

```
TBBlog createDemoPosts.
```

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

```
TBBlog count
>1
```

De la même manière, nous pouvons sélectionner touts objets sauvés.

```
TBBlog selectAll
```

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 extérieur.

Ces queries sont plus pertinentes quand on a plus d'objets comme lorsque nous ajouterons plusieurs blogs.

3.3 Changement de TBBlog

Notez qu'à chaque fois que vous décidez de changer soit en ajoutant une nouvelle racine d'objets ou soit lors d'une modification de la définition d'une classe racine (ajout, retrait, modification d'attribut) il est capital de réinitializer Voyage car Voyage maintient un cache.

La réinitialisation se fait comme suit:

```
VOMongoRepository current reset
```

3.4 Si nous devions sauvegarder les posts

Cette section n'est pas à implémenter et elle est juste donner à titre d'exemple. Nous pourrions aussi définir qu'un post est un élément qui peut être sauvegardé de manière autonome. Cela permettrait de sauver des posts de manière indépendante d'un blog. Si nous represention les commentaires d'un post nous ne les declarerions pas non plus comme racine car sauver ou manipuler un commentaire en dehors du contexte de son post ne fait pas beaucoup de sens.

Déclarer les posts comme racine aurait pour conséquence que les blogs sauvés aurait une référence à un objet TBPost alors que sans cela le post est inclus dans la sauvegarde du blog. Notez aussi que lorsqu'un post n'est pas un root, 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, après le chargement depuis la base, ce post sera dupliqué car recréer à partir de la définition du blog (et le blog contient alors complètement le post).

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

Pour cela on nous définirions les méthodes suivantes:

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

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

```
   aPost save.
   self save

TBBlog >> remove
   posts do: [ :each | each remove ].
   posts := OrderedCollection new.
   super remove.
```

Ici dans la méthode `remove` nous enlevons chaque posts puis nous remontons à jour la collection et nous enlevons le blog lui-même.

Encore une fois, notez qu'à chaque fois que vous décider de changer la définition d'une classe racine ou que vous ajoutez une nouvelle classe racine, il est capital de réinitializer Voyage car Voyage maintient un cache.

Pour cela exécutez l'expression suivante.

```
VOMongoRepository current reset
```

D'autres requêtes

Nous illustrons maintenant comment nous pouvons faire des requêtes sur la base en mémoire en incluant les posts.

```
TBBlog reset.
TBBlog current allBlogPosts size.
> 0

TBBlog createDemoPosts.
TBBlog current allBlogPosts size.
> 5

"Via repository"
TBBlog count.
> 1
TBPost count.
> 5

TBPost selectAll.
TBPost selectAll do: [:each | Transcript show: each text; cr]

TBBlog current remove.
TBPost removeAll.
TBPost count.
> 0
```

Nous allons maintenant utiliser une base mongo externe à Pharo.

CHAPTER 4

Mongo

En utilisant Voyage nous pouvons rapidement sauver nos posts sur une base de données Mongo. Ce chapitre explique rapidement la mise en oeuvre et les quelques modifications que nous devons apporter a notre projet pour sauver en Pharo. Nous commencons

4.1 Obtenir une base de données MongoDB

Installation locale

Mac OS X

- Installer Brew (`http://brew.sh`)
- Dans le terminal, mettre à jour les paquets et installer MongoDB:

```
brew update
brew install mongodb
```

- Créer un répertoire pour le stockage des données et attribution des droits

```
mkdir -p /data/db
chmod 770 /data/db
```

Lancement de mongo: `[sudo] mongod`

Attention une fois que votre application utilise Mongo, il est important que Mongo soit lancé avant Pharo. Vous pouvez tester cela avec ce script.

```
| server |
```

```
server := Mongo default.
[ server open ]
   on: ConnectionTimedOut
   do: [ :e | ^ false ].
^ server isOpen
```

Vous pouvez aussi utiliser la méthode `VOMongoRepository class >> #validateConectionHost:port:`

Linux Debian

Google pour le moment. A venir

Windows

Google pour le moment. A venir

Dans le cloud avec MongoLab

Dans un premier temps vous pouvez passer cette étape si vous avez une installation locale.

- Se connecter sur `https://mongolab.com`
- Cliquer sur signup,
- Créer un compte utilisateur (un mail de vérification de l'adresse mail est envoyé. Il faut confirmer le compte).
- Cliquer sur "Create New"
- modèle d'hébergement: option "Single-node" et on sélectionne "Sandbox" (gratuit pour 0.5 Go)
- On fournit un nom ("tinyblog") pour la base de données (Database name)
- On clique sur "Create new MongoDB deployment",
- En cliquant sur le nom de la base Mongo, on accède à l'écran de configuration,
  - Les paramètres de configuration sont dans l'URL,
  - On clique sur l'onglet "Users", puis le bouton "Add database user" pour ajouter un nouvel utilisateur,

Configuration du compte

Pensez à changer les valeurs!

Champs	Valeur
Account name	tinyblog
Username	tinyblog
Email	olivier.auverlot@free.fr
Password	tinyblog2015

Paramètres du serveur

Lors de votre enregistrement sur mongolab vous obtenez des valeurs que nous pouvons utiliser.

Paramètre	Valeur
Serveur	ds045064.mongolab.com
Port	45064
Nom de la base	tinyblog

Utilisateur de la base tinyblog

Pour le moment Voyage ne fonctionne qu'avec Mongo 2.0 car la version 3.0 a changé son mécanisme de vérification de mots de passe.

Champs	Valeur
Compte	tbuser
Mot de passe	tbpassword

4.2 Revisitons la connexion à la base

Nous définissions les méthodes `initializeMongo` pour établir la connexion vers la base de données.

```
TBBlog class >> initializeMongo
   | repository |
   repository := VOMongoRepository
      host: 'localhost'
      database: 'tinyblog'.
   repository enableSingleton.
```

> **Note** Pas d'authentification utilisée car impossible de le faire avec MongoDB 3.0 (nouvelle méthode de chiffrement SCRAM pas encore supportée avec Pharo)

Notez que si vous avez besoin de réinitialiser la base exterieure completement, vous pouvez utiliser la méthode `dropDatabase`

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

CHAPTER 5

Infrastructure Web

Nous commençons par définir une interface telle que celle que les utilisateurs la verrons. Dans un prochain chapitre nous développerons une interface d'administration que le possesseur du blog utilisera. Nous allons définir des composants Seaside `http://www.seaside.st`. L'ouvrage de référence est disponible en ligne à `http://book.seaside.st`

5.1 Démarrer Seaside

En utilisant le Seaside Control Panel et son menu, ajoutez un serveur `ZnZincServerAdaptor`, puis définissez le port surlequel le serveur doit fonctionner (comme illustré dans la figure 5.1 et 5.2).

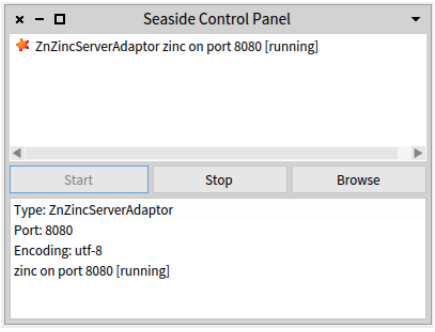

Figure 5.1: Lancer le serveur.

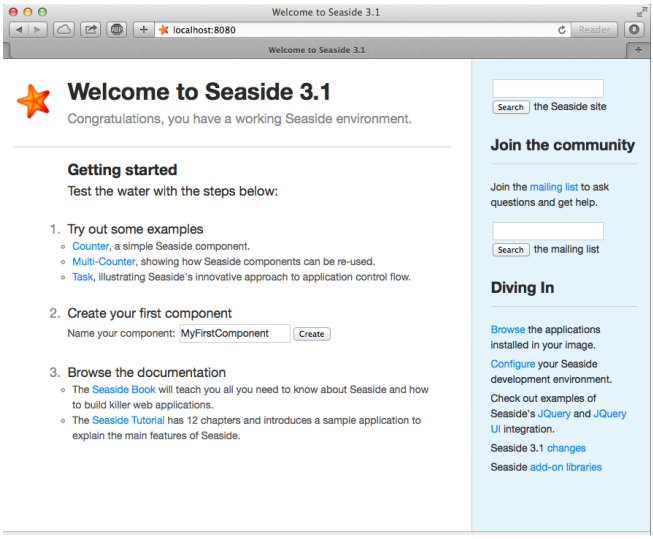

Figure 5.2: Vérification que Seaside fonctionne.

5.2 Initialisation de l'application

Création d'une classe `TBApplicationRootComponent` qui est le point d'entrée de l'application. Il sert à l'initialisation de l'application.

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

On déclare l'application au serveur Seaside, coté classe, dans le protocole `'initialization'`. 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
```

Dans un Playground ou dans l'éditeur de méthode, exécuter `TBApplicationRootComponent initialize` pour forcer l'exécution de la méthode `initialize`. Notez que nous venons de la définir et donc il est nécessaire de l'exécuter pour en voir les bénéfices.

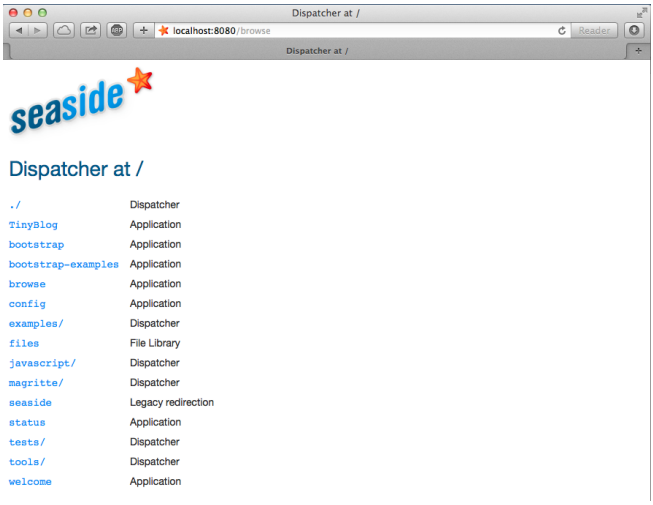

Figure 5.3: TinyBlog est bien enregistrée.

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

Une connexion sur le serveur Seaside ("Browse the applications installed in your image") permet de vérifier que l'application est bien enregistrée comme le montre la figure 5.3.

Ajoutons également la méthode `canBeRoot` afin de préciser que la classe `TBApplicationRootComponent` est la première instanciée lors qu'un utilisateur se connecte sur l'application et représente une application et pas seulement un composant.

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

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

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

Connexion avec un navigateur sur `http://localhost:8080/TinyBlog`. La page doit apparaître comme dans la figure 5.4.

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

```
TBApplicationRootComponent >> updateRoot: anHtmlRoot
   super updateRoot: anHtmlRoot.
```

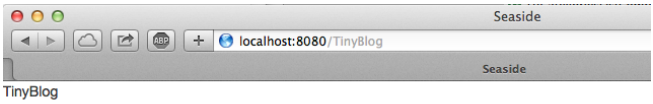

Figure 5.4: Une page quasi vide mais servie par Seaside.

```
anHtmlRoot beHtml5.
anHtmlRoot title: 'TinyBlog'.
```

Le composant `TBApplicationRootComponent` est le composant principal de l'application, il ne fait que rendu graphique limité en affichant dans le future le composant principal qu'il contiendra. Il contiendra plus tard soit les composants qu'un lecteur du blog pourra utiliser soit les composants pour administrer le blog et ses postes.

Nous avons décidé que le composant `TBApplicationRootComponent` contiendra des composants inhéritants de la classe abstraite `TBScreenComponent` que nous allons définir dans le prochain chapitre.

CHAPTER 6

Composants Visuels de TinyBlog

Nous sommes prets à définir les composants visuels de notre petite application. Les premiers chapitres de `http://book.seaside.st` peuvent vous aider.

La figure 6.1 montre les différents composants que nous allons développer.

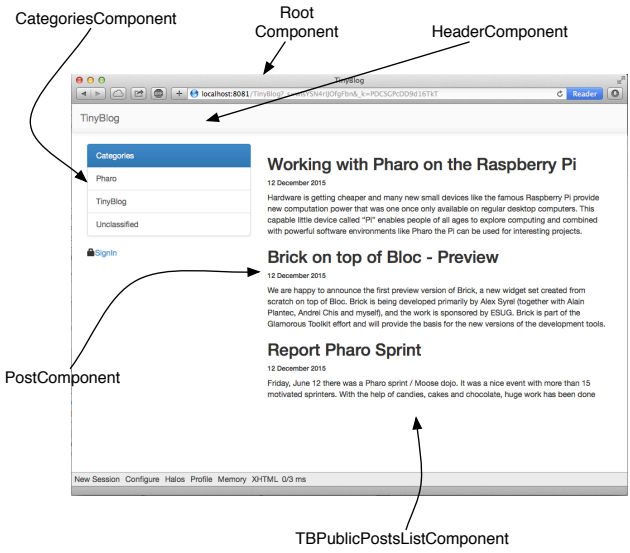

Figure 6.1: Les composants composant l'application TinyBlog.

6.1 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 auront besoin d'accéder aux règles métier de l'application. Dans le protocole 'accèssing', créons un méthode `blog` qui retourne l'instance de `TBBlog`.

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

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

6.2 Bootstrap for Seaside

La bibliothèque Bootstrap est totallement accèssible 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 6.2

Cliquer sur le lien **Examples** au bas de la page et vous pouvez ainsi voir les éléments graphiques ainsi que le code pour les obtenir comme montré par la figure 6.3.

Bootstrap

Le repository pour le source et la documentation est `http://smalltalkhub.com/#!/~TorstenBergmann/Bootstrap`. Une demo en ligne est disponible à `http://pharo.pharocloud.com/bootstrap`.

6.3 Définition du composant TBHeaderComponent

Profitons de ce composant pour insérer dans la partie supérieure de chaque composant, l'instance d'un composant représentant l'entête de l'application.

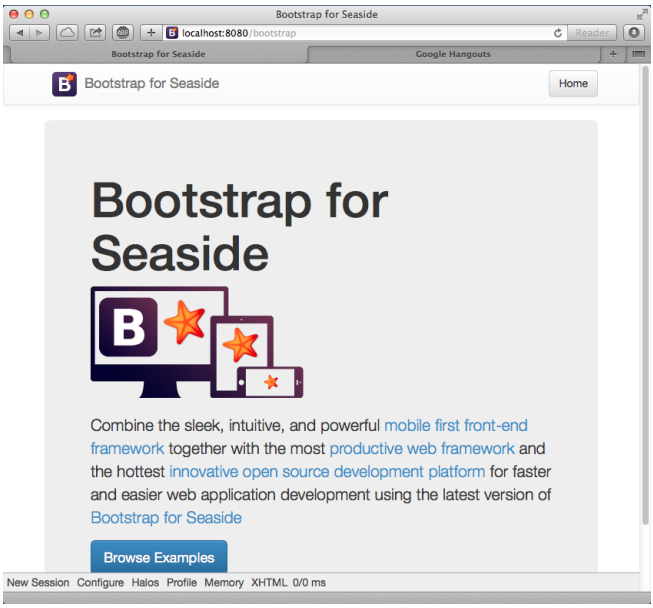

Figure 6.2: Accès à la bibliothèque Bootstrap.

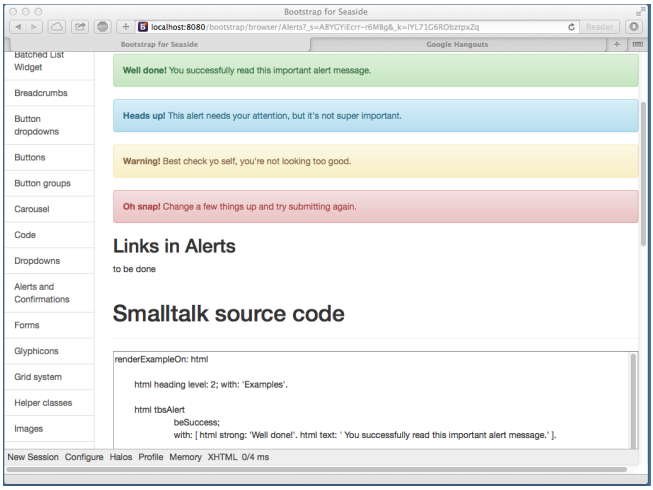

Figure 6.3: Comprendre un élément et son code en Bootstrap.

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

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

```
TBHeaderComponent >> renderContentOn: html
   html tbsNavbar beDefault with: [
      html tbsNavbarBrand
         url: '#';
         with: 'TinyBlog' ]
```

L'entête (header) est affichée à l'aide d'une barre de navigation Bootstrap (voir la figure 6.4)

Par défaut dans une barre de navigation Bootstrap, il y a un lien sur tbsNavbar-Brand qui est ici inutile (sur le titre de l'application). Ici nous l'initialisons avec une ancre `'#'` de façon à ce que si l'utilisateur clique sur le titre, il ne se passe rien. En général, cliquer sur le titre de l'application permet de revenir à la page de départ du site.

Améliorations possibles

Le nom du blog devrait être paramêtrable à l'aide d'une variable d'instance dans la classe `TBBlog` et le header pourrait représentait ce titre.

6.4 Utilisation du Header

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

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

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

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

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

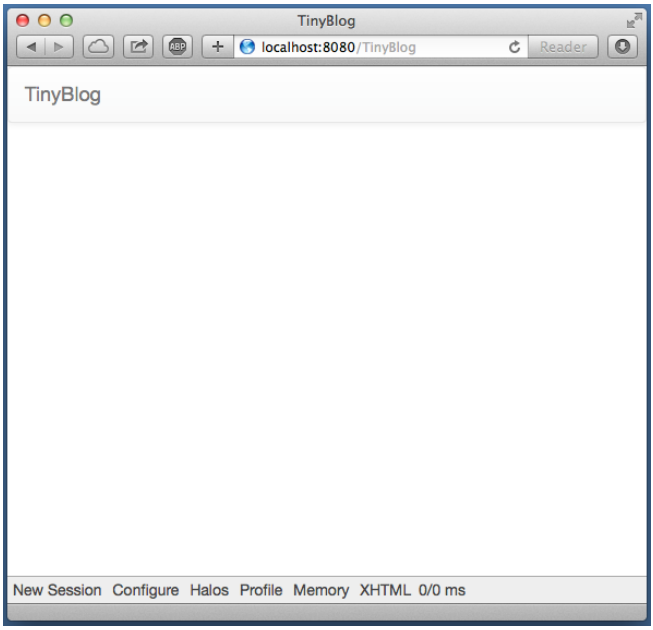

Figure 6.4: TinyBlog avec une barre de navigation.

composant `TBHeaderComponent` est un enfant de `TBScreenComponent` dans la hiérarchie de composants Seaside (en non entre classes Pharo).

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

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

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

6.5 Utilisation du Composant Screen

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

Ainsi nous l'instancions dans la méthode `initialize` suivante.

```
TBApplicationRootComponent >> initialize

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

```
TBApplicationRootComponent >> renderContentOn: html

   html render: main

TBApplicationRootComponent >> children

   ^ { main }
```

Si vous faites un rafraichissement de l'application vous devez voir la Figure 6.4.

6.6 Liste des Posts

Nous allons afficher la liste des posts - ce qui reste d'ailleurs le but d'un blog. Ici nous parlons de l'accès public au lecteur du blog. Dans le future, nous proposerons une interface administrative des posts.

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

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

Ajoutons une méthode `renderContentOn:` (protocole rendering) provisoire pour tester l'avancement de notre application (voir Figure 6.5).

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

Nous pouvons maintenant dire au composant de l'application d'utiliser ce composant. Pour cela nous redéfinissons la

```
TBApplicationRootComponent >> initialize

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

Editer cette méthode n'est guère propre. Nous ajoutons un setter qui nous permettra de changer dynamiquement de composant dans le future tout en gardant le composant actuel pour une initialisation par défaut.

```
TBApplicationRootComponent >> main: aComponent

   main := aComponent
```

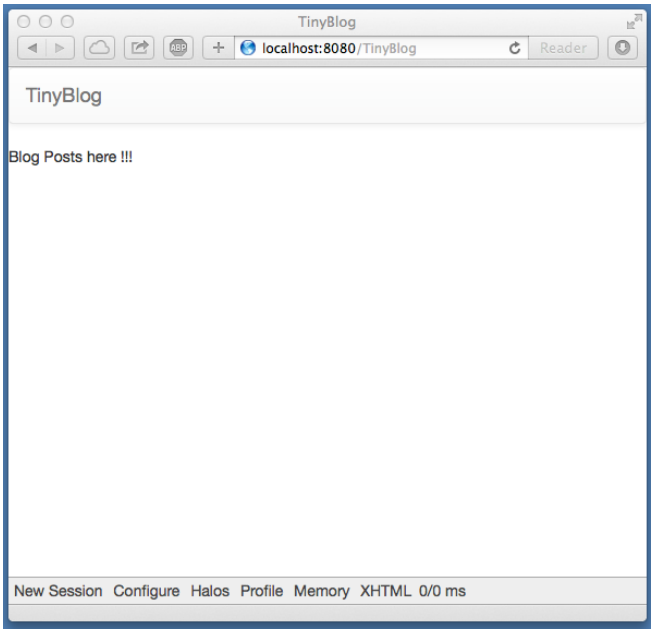

Figure 6.5: TinyBlog avec une liste de posts plutot élémentaire.

6.7 Le composant Post

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

Chaque post du blog sera représenté visuellement par une instance de `TBPostComponent` qui affiche le titre, la date et le contenu d'un post.

```
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 définie 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 de formulaire

Nous montrerons dans le chapitre décrivant Magritte puis les interfaces d'administration, qu'il est rare de définir un composant de manière aussi manuelle. En effet Magritte en décrivant les données manipulées permet de générer automatiquement des composants Seaside. Le code équivalent serait comme suit:

```
TBPostComponent >> renderContentOn: html
   html render: self asComponent
```

6.8 Afficher les posts

Maintenant nous pouvons afficher des posts présents dans la base.

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

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

Rafraichissez la page et vous devez obtenir la Figure 6.6.

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

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

Rafraichissez la page et vous devez obtenir la Figure 6.7.

Les dates sont affichées de manière étrange. Ceci est dû au fait que les dates sont en fait des instance de `DateAndTime` donc nous les convertissons dans date comme suit.

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

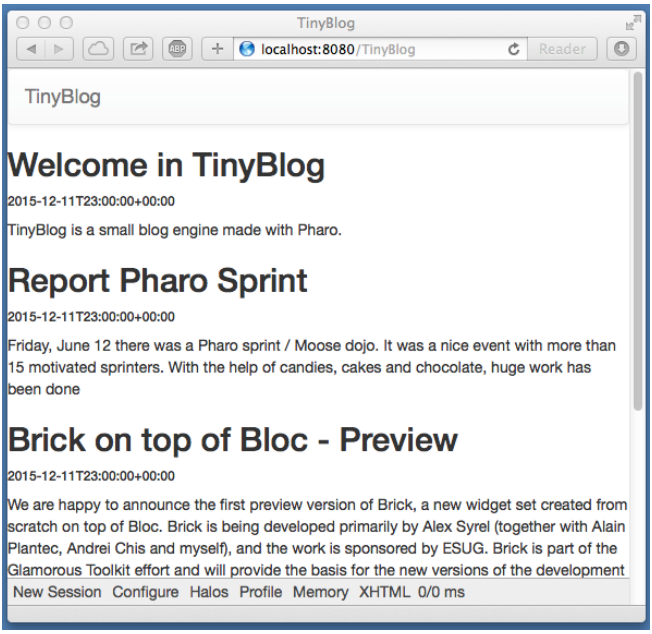

Figure 6.6: TinyBlog avec une liste de posts.

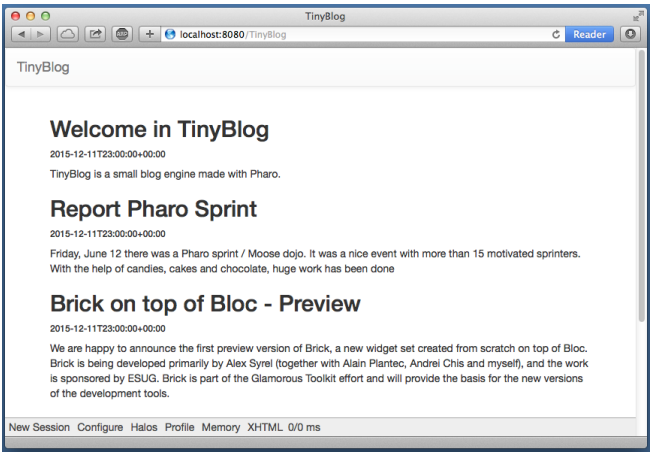

Figure 6.7: TinyBlog avec une liste de posts élémentaire.

6.9 Affichage des posts par catégorie

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

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

Categories

Nous avons besoin d'un composant qui affiche la liste des catégories présentes dans la base et permet d'en sélectionner une. Ce composant devra donc avoir la possibilité de communiquer avec le composant `TBPostsListComponent` afin de lui communiquer la catégorie courante.

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

TBCategoriesComponent >> categories
   ^ categories

TBCategoriesComponent >> categories: aCollection
   categories := aCollection

TBCategoriesComponent >> postsList: aComponent
      postsList := aComponent

TBCategoriesComponent >> postsList
   ^ postsList
```

Nous définissons aussi une méthode de création au niveau classe.

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

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

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

Nous avons donc besoin d'ajouter une variable d'instance dans `TBPostsList-Component`.

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

```
TBScreenComponent >> currentCategory
   ^ currentCategory

TBScreenComponent >> currentCategory: anObject
   currentCategory := anObject
```

Nous pouvons maintenant ajouter une méthode (protocole 'rendering') pour afficher les catégories sur la page. En particulier pour chaque catégorie nous définissons le fait que cliquer sur la catégorie la sélectionne comme la catégorie courante.

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

Reste maintenant à écrire la méthode de rendu du composant:

```
TBCategoriesComponent >> renderContentOn: html
   html tbsListGroup: [
     html tbsLinkifyListGroupItem beActive; with: 'Categories'.
     categories do: [ :cat | self renderCategoryLinkOn: html with: cat ]]
```

Nous avons presque fini. En effet, l'affichage des posts ne tient pas en compte de la catégorie courante.

Mise à jour des Posts

Nous devons aussi mettre à jour la liste des posts. Il faut gérer le rafraîchissement de la liste des posts en fonction de la catégorie choisie et donc modifier la méthode de rendu du composant `TBPostsListComponent`.

La méthode `readSelectedPosts` récupère dans la base les posts à afficher. Si elle vaut `nil`, l'utilisateur n'a pas encore sélectionner une catégorie et l'ensemble des posts 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 posts attachés à la 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
```

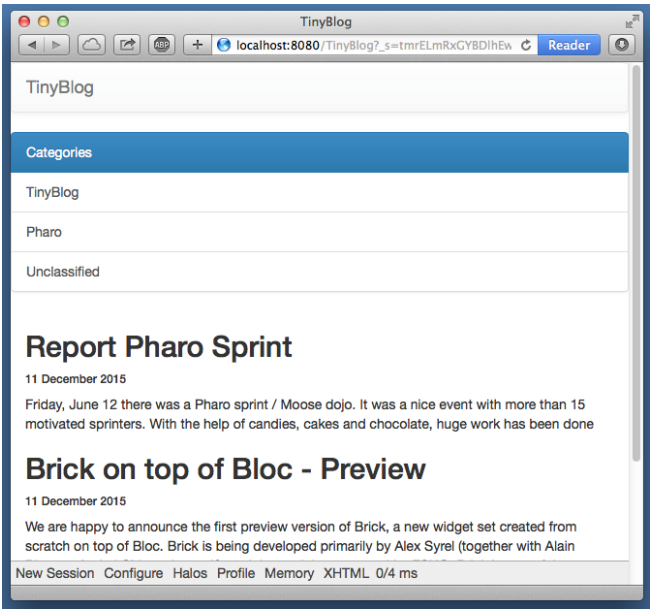

Figure 6.8: Catégories afin de sélectionner les posts.

```
            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 figure 6.8).

6.10 Look et Agencement

Nous allons maintenant agencer le composant `TBPostsListComponent`.

Mise en place d'un responsive design pour la liste des posts. 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: [
```

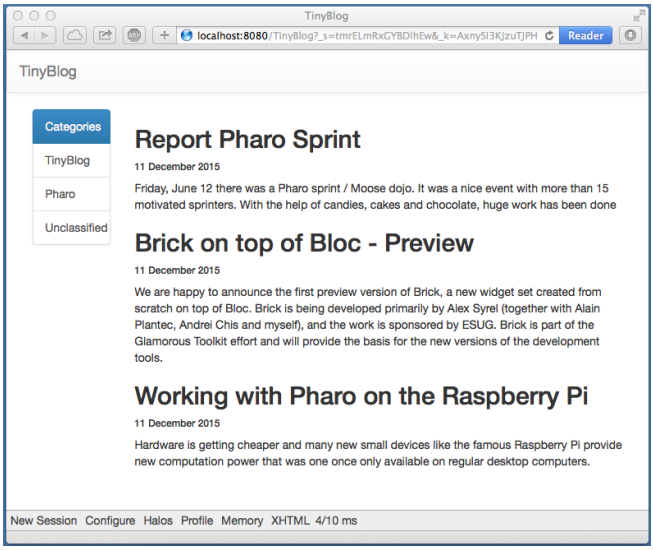

Figure 6.9: Avec un meilleur agencement.

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

Vous devez obtenir une application proche de celle représentée figure 6.9.

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

Notre solution

`TBPostsListComponent >> renderContentOn: html`

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

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

TBPostsListComponent >> basicRenderCategoriesOn: html
   ^ html render: (TBCategoriesComponent
         categories: self blog allCategories
         postsList: self)

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

basicRenderPostsOn: html
   ^ self readSelectedPosts do: [ :p |
        html render: (TBPostComponent new post: p) ]
```

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

CHAPTER 7

Décrire les données avec Magritte

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

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

Dans ce chapitre, les cinq variables d'instance de l'objet `TBPost` sont décrites à l'aide de Magritte. Dans ce chapitre nous décrivons seulement les données et le chapitre suivant en tirera avantage.

7.1 Descriptions

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

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

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

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

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

La catégorie d'un post est une chaine 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
      accèssor: #category;
      yourself
```

La date de création d'un post 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
      accèssor: #date;
      beRequired;
      yourself
```

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

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

Ameliorations possibles

Voici quelques améliorations. Nous pourrions améliorer cette description pour qu'il ne soit pas possible de poster un post ayant une date antèrieure celle d'ajourd'hui. Nous pourrions changer la description d'une categorie pour en definir un enuméré parmi les catégories existantes. Ceci permettrait de faire une interface plus simplement.

CHAPTER 8

Administration de TinyBlog

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

Le scénario assez classique que nous allons développer est le suivant : l'utilisateur doit s'authentifier pour accéder à la partie administration de TinyBlog. Il le fait à l'aide d'un compte et d'un mot de passe. Le lien permettant d'afficher le composant d'authentification sera placé sous la liste des catégories.

8.1 Composant d'identification

Nous allons commencer par développer un petit composant d'identification qui lorsqu'il sera invoqué ouvrira une petite boite de dialogue pour demander les informations d'identification. Remarquer qu'une telle fonctionnalité devrait faire partie d'une bibliothèque de composants de base en Seaside.

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

```
WAComponent subclass: #TBAuthentificationComponent
   instanceVariableNames: 'password account component'
   classVariableNames: ''
   category: '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 est initialisée par la méthode de classe suivante :

```
TBauthenticationComponent class >> from: aComponent
   ^ self new component: aComponent
```

La méthode renderContentOn: définit le contenu d'une boite de dialogue modale.

```
TBAuthentificationComponent >> renderContentOn: html

   html tbsModal id: 'myAuthDialog'; with: [
    html tbsModalDialog: [
      html tbsModalContent: [
        html tbsModalHeader: [
          html tbsModalCloseIcon.
          html tbsModalTitle level: 4; with: 'Authentification'
        ].
        html tbsModalBody: [
          html form: [
            html text: 'Account:'.
            html break.
    html textInput
              callback: [ :value | account := value ];
              value: account.
    html break.
    html text: 'Password:'.
    html break.
    html passwordInput
      callback: [ :value | password := value ];
      value: password.
    html break.
    html break.
    html tbsModalFooter: [
      html tbsSubmitButton value: 'Cancel'.
        html tbsSubmitButton
          bePrimary;
            callback: [ self validate ];
```

```
        value: 'SignIn'.
    ] ] ] ] ] ]
```

Critique

Quelle critique peut on faire sur cette méthode ? Proposer une refactorisation à l'aide de plusieurs méthodes pour définir les différents éléments du formulaire de manière modulaire.

Notre solution

```
TBAuthentificationComponent >> renderContentOn: html
   html tbsModal
      id: 'myAuthDialog';
      with: [ html tbsModalDialog: [
                  html tbsModalContent: [
                        self renderHeaderOn: html.
                        html
                           tbsModalBody: [
                              html
                                 form: [
                                    self renderInputAccountOn: html.
                                    html break.
                                    self renderInputPasswordOn: html.
                                    html break.
                                    html break.
                                    html tbsModalFooter: [ self
                                         renderOkCancelOn: html ] ] ] ] ]
                                         ]

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

TBAuthentificationComponent >> renderInputAccountOn: html
   html text: 'Account:'.
   html break.
   html textInput
         callback: [ :value | account := value ];
         value: account

TBAuthentificationComponent >> renderInputPasswordOn: html
   html text: 'Password:'.
   html break.
```

```
   html passwordInput
      callback: [ :value | password := value ];
      value: password

TBAuthentificationComponent >> renderOkCancelOn: html
   html tbsSubmitButton value: 'Cancel'.
   html tbsSubmitButton
      bePrimary;
      callback: [ self validate ];
      value: 'SignIn'
```

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

```
TBAuthentificationComponent >> validate
   (self account = 'admin' and: [ self password = 'password' ])
      ifTrue: [ self alert: 'Success!' ]
```

Critique

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

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

Intégration de l'authentification

Il faut maintenant intégrer le lien qui déclenchera l'affichage de la boite modale d'authentification. Au tout début de la méthode `renderContentOn:` du composant `TBPostsListComponent`, on ajoute le rendu du composant d'authentification. Ce composant reçoit en paramètre la référence vers le composant affichant les posts.

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

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

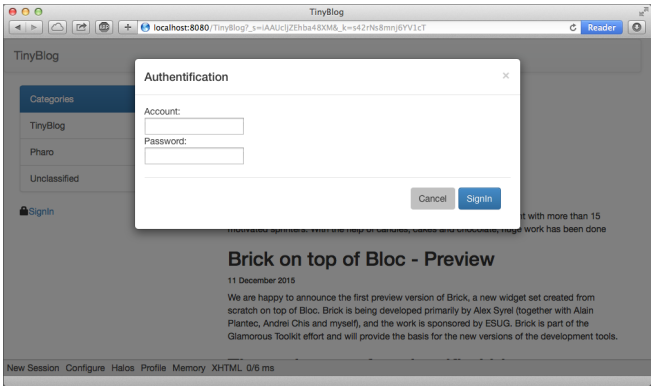

Figure 8.1: Avec un meilleur agencement.

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

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

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

Lorsque nous pressons sur le lien SignIn nous obtenons la figure 8.1.

8.2 Administration des posts

Nous allons développer deux components l'un deux sera un rapport qui contiendra tous les posts et un autre composant qui contiendra ce rapport. Le rapport bien que générer avec Magritte est un composant Seaside nous aurions pu n'avoir qu'un seul composant mais nous pensons que distinguer le composant d'administration du rapport est une bonne chose pour l'évolution de la partie administration. Commençons donc par le component d'administration.

Création d'un composant d'administration

Le composant `TBAdminComponent` hérite de `TBScreenComponent` pour bénéficier du header et de l'accès au blog. Il contiendra en plus le rapport que nous

construisons par la suite.

```
TBScreenComponent subclass: #TBAdminComponent
   instanceVariableNames: ''
   classVariableNames: ''
   category: '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 ]
```

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

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

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

Enregistrez-vous et vous devez obtenir la situation telle que représentée par la figure 8.2.

Rapport

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

Pour rester modulaire, nous allons créer un composant Seaside pour cette tâche.

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

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

```
TBPostsReport class >> from: aBlog
   | allBlogs |
   allBlogs := aBlog allBlogPosts.
```

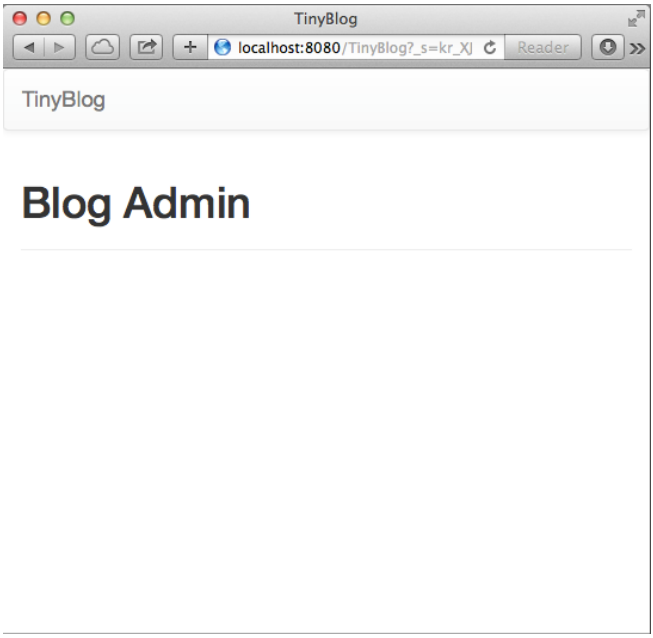

Figure 8.2: Un composant d'administration vide.

```
    ^ self rows: allBlogs description: allBlogs anyOne
        magritteDescription
```

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

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

```
TBPostsReport class >> filteredDescriptionsFrom: aBlogPost
    ^ aBlogPost magritteDescription select: [ :each | #(title category
        date) includes: each accèssor selector ]

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

On ajoute un rapport au composant admin `TBAdminComponent`.

```
TBScreenComponent subclass: #TBAdminComponent
    instanceVariableNames: 'report'
    classVariableNames: ''
```

```
   category: '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.

```
TBAdminComponent >> children
   ^ OrderedCollection with: self report
```

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

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

Nous pouvons maintenant afficher le rapport.

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

Enregistrez-vous et vous devez obtenir la situation telle que représentée par la figure 8.3.

Amélioration des rapports

Le rapport généré est brut. Il n'y a pas de titres sur les colonnes et l'ordre d'affichage des colonnes n'est pas fixé (il peut varier d'une instance à une autre). Pour gérer cela, il suffit de modifier les descriptions Magritte pour chaque variable d'instance.

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

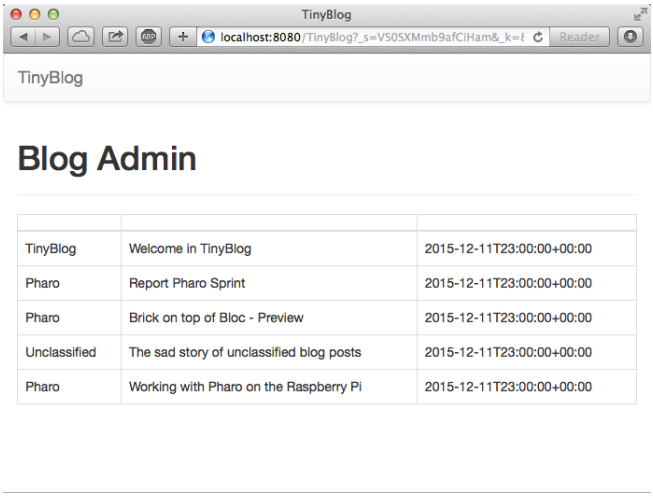

Figure 8.3: Administration avec un rapport.

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

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

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

TBPost >> descriptionVisible
   <magritteDescription>
   ^ MABooleanDescription new
```

```
    label: 'Visible';
    priority: 500;
    accèssor: #visible;
    beRequired;
    yourself
```

Gestion des posts

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

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

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

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
```

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

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

Enregistrez-vous et vous devez obtenir la situation telle que représentée par la figure 8.4.

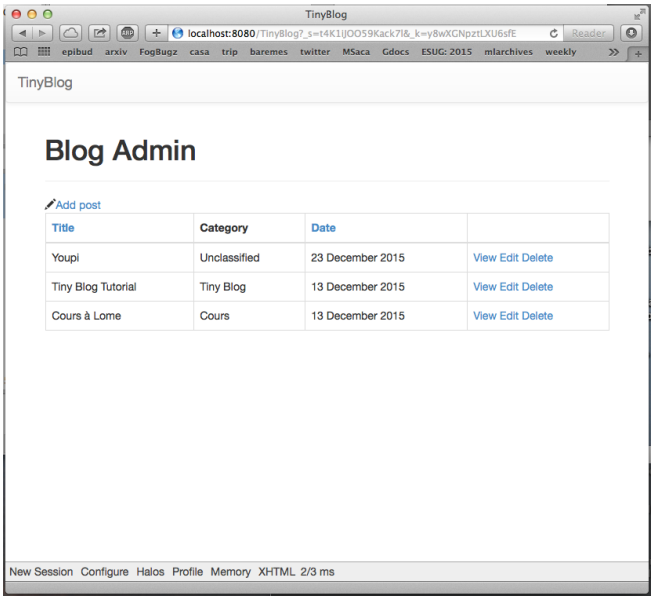

Figure 8.4: Ajout d'un post.

Implémentation des actions du 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 demandé (il n'est pas utile par exemple d'avoir un bouton "Sauver" alors que l'utilisateur veut simplement lire le post).

Ajouter un post

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

La méthode `renderAddPostForm` illustre la puissance de Magritte pour générer des formulaire. Ici le message `asComponent` envoyé à un objet de la class `TBPost` donc du domaine, crée directement un composant Seaside. Nous lui ajoutons une décoration afin de gérer ok/cancel.

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

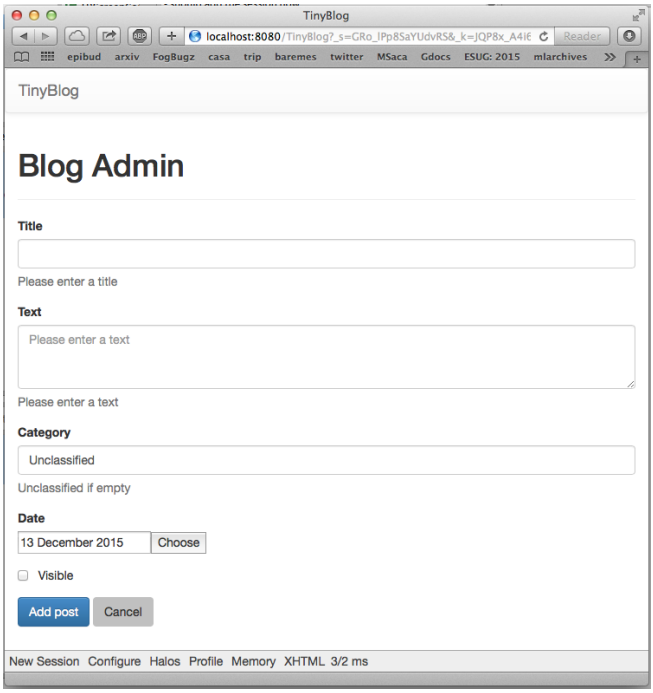

Figure 8.5: Ajout d'un post.

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

Enregistrez-vous et vous devez obtenir la situation telle que représentée par la figure 8.5.

Editer un post

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

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

Consulter un post

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

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

To do we should edit in readonly!

Effacer un post

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 actualisé et le rapport est rafraîchi.

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

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

Nous devons ajouter un test.

Gérer le problème du raffraichissement des données

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

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

TBPostsReport >> addPost
  | post |
  post := self call: (self renderAddPostForm: 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 formulaire est fonctionnel maintenant et gère même les contraintes de saisie.

Amélioration de l'apparence du formulaire

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

```
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;
    accèssor: #title;
    requiredErrorMessage: 'A blog post must have a title.';
    comment: 'Please enter a title';
    componentClass: TBSMagritteTextInputComponent;
    beRequired;
    yourself

TBPost >> descriptionText
  <magritteDescription>
  ^ MAMemoDescription new
    label: 'Text';
    priority: 200;
    accèssor: #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;
    accèssor: #category;
    comment: 'Unclassified if empty';
    componentClass: TBSMagritteTextInputComponent;
    yourself
```

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

8.3 Gestion de Session

To do Should revisit this part.

Un objet session est attribué à chaque instance de l'application. Il permet de conserver principalement des informations qui sont partagées et accèssible entre les composants. Une session est pratique pour gérer les informations de l'utilisateur en cours (loggé). Nous allons voir comment nous l'utilisons pour gérer une connexion.

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

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

```
WASession subclass: #TBSession
  instanceVariableNames: 'logged'
  classVariableNames: ''
  category: 'TinyBlog'

TBSession >> logged
  ^ logged

TBSession >> logged: anObject
  logged := anObject

TBSession >> isLogged
  ^ self logged
```

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

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

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

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

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

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

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

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

Il vous faut maintenant spécifier à Seaside qu'il doit utiliser l'objet `TBSession` comme objet de session courant pour l'application TinyBlog. Pour cela, on utilise l'outil d'administration de Seaside.

- connexion sur `http://localhost:8080/config`,
- on clique sur "TinyBlog",
- Dans "General", cliquez sur le bouton "Override" de "Session Class",
- Choisir 'TBSession' dans la liste déroulante,
- Cliquez sur le bouton "Apply" en bas du formulaire.

On peut automatiser cette initialisation en améliorant la méthode initialize de la class `TBApplicationRootComponent`.

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

Exercices:

Proposer l'ajout d'un bouton "Déconnexion" La gestion de plusieurs blogs pourra aussi passer par une amélioration des sessions qui devront conserver l'identification de l'utilisateur.

CHAPTER 9

REST

9.1 Définir un filtre REST

```
WARestfulFilter subclass: #TBRestfulFilter
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'TinyBlog-REST'
```

Associer un filtre à notre application

```
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.
```

Ne pas oublier d'initialiser.

```
self initialize
```

9.2 Obtenir

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

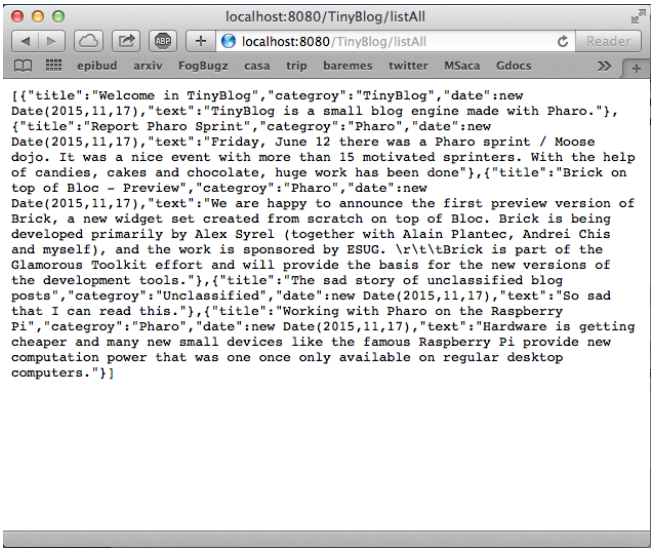

Figure 9.1: accès à tous les posts.

```
  ^ String streamContents: [ :astream |
      TBBlog current allBlogPosts
     do: [ :each | each javascriptOn: stream ]
     separatedBy: [ stream << ',' ]]
   ]
```

Si on veut selectionner les informations que l'on retourne.

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

  ^ String streamContents: [ :astream |
       TBBlog current allBlogPosts
       do: [ :each |
       Dictionary new
         at: #title put: each title;
         at: #category put: each category;
         at: #date put: each date;
         at: #text put: each text;
         javascriptOn: astream ]
       separatedBy: [ astream << ',' ]]
```

Refactoring

Définir la transformation en Javascript dans notre composant n'est pas optimal. Nous allons donc le définir dans la classe et appeller cette fonctionalité.

```
TBPost >> javascriptOn: aStream

  Dictionary new
      at: #title put: self title;
      at: #categroy put: self category;
      at: #date put: self date;
      at: #text put: self text;
      javascriptOn: aStream

TBRestfulFilter >> listAll
  <get>
  <produces: 'text/json'>

  ^ String streamContents: [ :astream |
    astream << '['.
    TBBlog current allBlogPosts
      do: [ :each | each javascriptOn: astream]
      separatedBy: [ astream << ',' ].
    astream << ']'
    ]
```

Nous raffinons afin de pouvoir sélectionner les attributs que nous voulons convertir en introduisant la méthode `javascriptOn: aStream attributes: aCollection`.

```
TBPost >> javascriptOn: aStream attributes: aCollection

  | dictionary |
  dictionary := Dictionary new.
  aCollection do: [ :each |
       dictionary at: each put: (self perform: each asSymbol) ].
  ^ dictionary javascriptOn: aStream

TBPost >> javascriptOn: aStream

  ^ self javascriptOn: aStream attributes: #(title category date)
```

Recherche d'un Post

Maintenant nous allons proposer d'autres fonctionalité comme la recherche d'un poste. Nous définissons donc cette fonctionalité dans la classe `TBlog`.

```
TBBlog >> postWithTitle: aString

  ^ self allBlogPosts
```

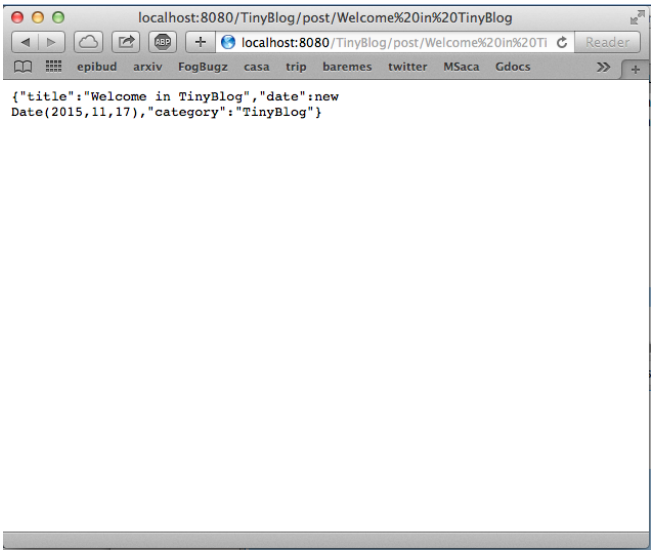

Figure 9.2: REST post.

```
      detect: [ :each | each title = aString ]
      ifNone: [ nil ]

TBRestfulFilter >> post: title
  <get>
  <path: 'post/{title}'>
  <produces: 'text/json'>
   | post |
  post := TBBlog current postWithTitle: title.
  post ifNil: [ ^ self notFound ].
   ^ String streamContents: [ :astream |
    post javascriptOn: astream ]
```

Nous pouvons tester en faisant la requete suivante :

```
http://localhost:8080/TinyBlog/post/Welcome%20in%20TinyBlog

TBRestfulFilter >> notFound

  | |
  "trick self halt puis dans le debugger req := self requestContext."
  self requestContext responseGenerator
    notFound;
    respond

javascriptOn: aStream

   ^ self javascriptOn: aStream attributes: #(title category date text)
```

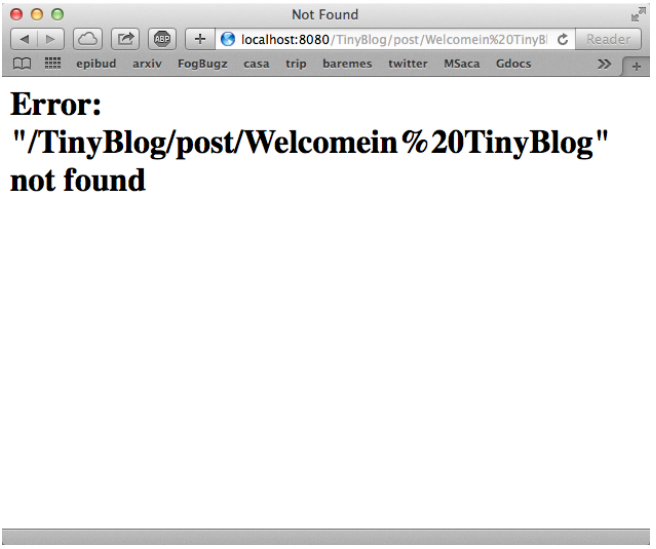

Figure 9.3: REST Error.

```
TBRestfulFilter >> search: title
  <get>
  <path: 'search?title={title}'>
  <produces: 'text/json'>
   | post |
  post := TBBlog current postWithTitle: title.
  post ifNil: [ ^ self notFound ].
   ^ String streamContents: [ :astream |
    post javascriptOn: astream attributes: #(title category date text)]

http://localhost:8080/TinyBlog/search?title=Welcome%20in%20TinyBlog

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

  dateFrom := Date fromString: beginString.
  dateTo := Date fromString: endString.

  posts := TBBlog current allBlogPosts
    select: [ :each | each date between: dateFrom and: dateTo ].

  ^ String streamContents: [ :astream |
    astream << '['.
      posts
```

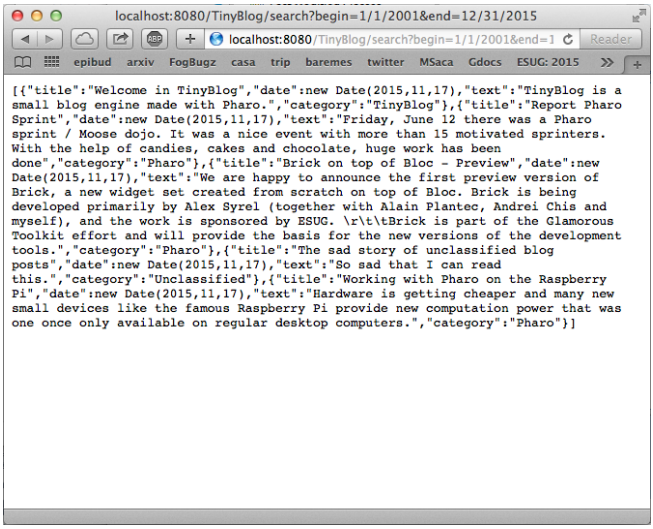

Figure 9.4: Avec des dates.

```
    do: [ :each | each javascriptOn: astream]
    separatedBy: [ astream << ',' ].
  astream << ']'
  ]
```

Dire un mot de

- Put in header vs. Post

Eliminer un post

```
TBRestfulFilter >> remove: title
  <delete>
  <path: '/remove/{title}'>
  <produces: 'text/json'>

  | post |

  post := TBBlog current postWithTitle: title.
  ^ post
    ifNotNil: [
      post remove.
      '{status: "ok"}' ]
    ifNil: [
      '{status: "notFound"}' ]
```

Notez que status n'est pas du REST car REST ne specifie pas la maniere de gerer les erreurs. Maintenant ce code ne fonctionne pas car `TBPost` n'est pas une racine en Voyage.

Essayons mais nous ne pouvons pas tester avec un browser web dans son url ne peut faire qu'un GET. Donc il nous faut utiliser Zinc.

```
ZnEasy delete: 'http://localhost:8080/TinyBlog/remove/RemoveMe'
```

Vous pouvez verifiez que votre post n'est pas enlevé.

```
TBRestfulFilter >> remove: title
  <delete>
  <path: '/remove/{title}'>
  <produces: 'text/json'>

  | post |
  post := TBBlog current postWithTitle: title.
  ^ post
    ifNotNil: [
      TBBlog current removeBlogPost: post.
      '{status: "ok"}' ]
    ifNil: [
      '{status: "notFound"}' ]
```

Maintenant nous réessayons et nous pouvons effectivement éliminer un post.

```
ZnEasy delete: 'http://localhost:8080/TinyBlog/remove/RemoveMe'
```

Vous pouvez verifiez que votre post n'est pas enlever.

Ajout de Blog Post

```
TBRestfulFilter >> addPost
  <post>
  <path: '/add-post'>

  | request title category date text post |

  request := self requestContext request.
  title := request at: #title.
  category := request at: #category.
  text := request at: #text.
  date := Date fromString: (request at: #title).
  "Of course many can go wrong here (Date could be wrong ... and a real
      version we should validate"
  post := TBBlog new
    title: title;
    text: text;
    category: category;
    date: date;
    yourself.
```

```
TBBlog current writeBlogPost: post.
^ '{status: "ok"}'

ZnClient new
  url: 'http://localhost:8080/TinyBlog/add-post';
  formAt: 'title' put: 'Exemple de POST';
  formAt: 'category' put: 'TEST';
  formAt: 'date' put: '12/23/2015';
  contents: 'Ici un super post';
  post.
```

CHAPTER 10

Deployement