CHAPTER 1

Construction d'une interface Web en Seaside pour TinyBlog

1.1 Correction semaine précédente

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

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

Assurez vous que votre serveur Teapot est bien stoppé.

```
TBTeapotWebApp stop
```

1.2 Une interface Web pour TinyBlog en Seaside

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

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

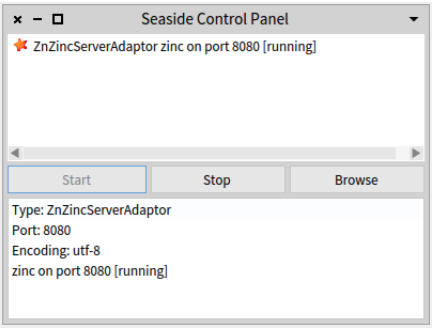

Figure 1.1 Lancer le serveur.

1.3 Démarrer Seaside

Il existe deux façons pour démarrer Seaside. La première consiste à exécuter le code suivant :

```
ZnZincServerAdaptor startOn: 8080.
```

La deuxième façon est graphique via l'outil Seaside Control Panel (World Menu>Tools>Seaside Control Panel). Dans le menu contextuel de cet outil (clic droit), cliquez sur "add adaptor..." pour ajoutez un serveur `ZnZincServerAdaptor`, puis définissez le port (e.g. 8080) sur lequel le serveur doit fonctionner (comme illustré dans la figure 1.1). 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 1.2.

1.4 Point d'entrée de l'application

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

```
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'.
```

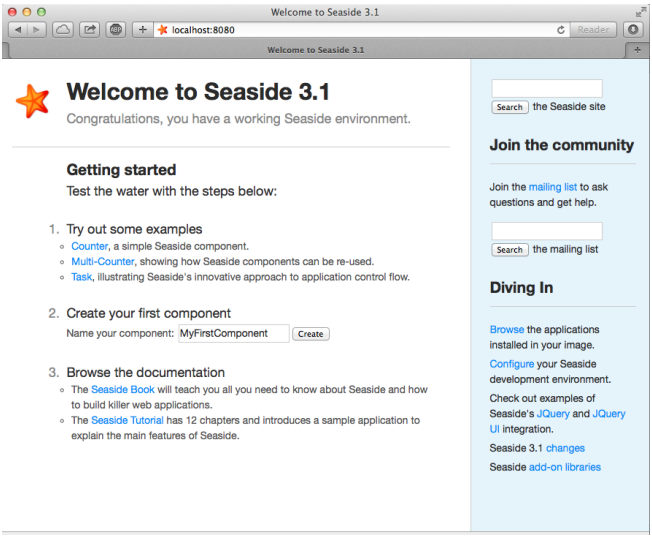

Figure 1.2 Vérification que Seaside fonctionne.

```
app
   addLibrary: JQDeploymentLibrary;
   addLibrary: JQUiDeploymentLibrary;
   addLibrary: TBSDeploymentLibrary
```

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

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

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

```
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 1.3.

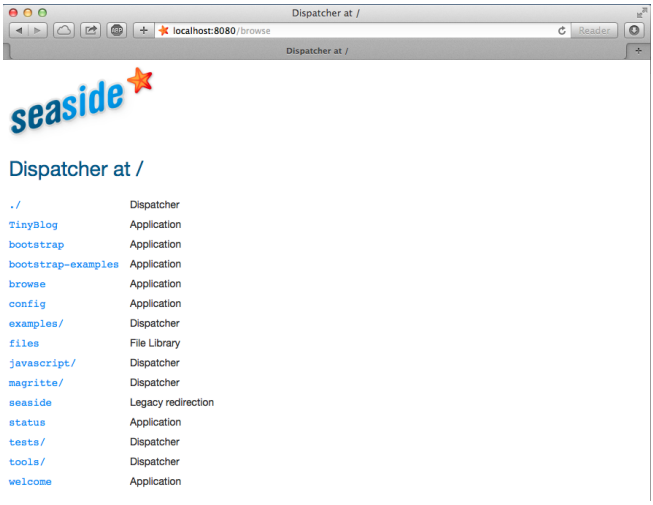

Figure 1.3 TinyBlog est bien enregistrée.

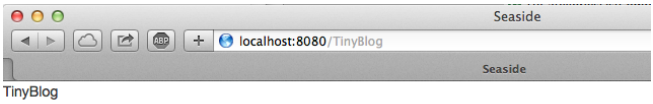

Figure 1.4 Une page quasi vide mais servie par Seaside.

1.5 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 1.4.

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.
   anHtmlRoot beHtml5.
```

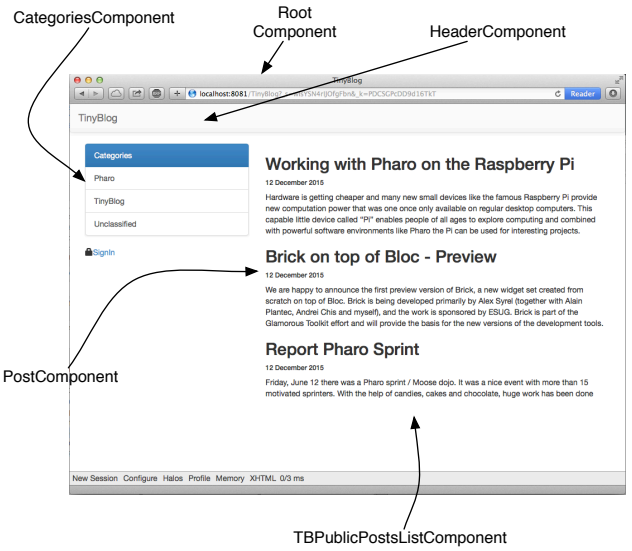

Figure 1.5 Les composants composant l'application TinyBlog.

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

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

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

1.6 Composants visuels pour TinyBlog

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

La figure 1.5 montre les différents composants que nous allons développer et ou 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 auront besoin d'accéder aux règles métier de l'application. Dans le protocole 'accessing', créons une méthode `blog` qui retourne une instance de `TBBlog` (ici notre singleton).

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

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

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

1.7 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 1.6.

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 1.7.

Bootstrap

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

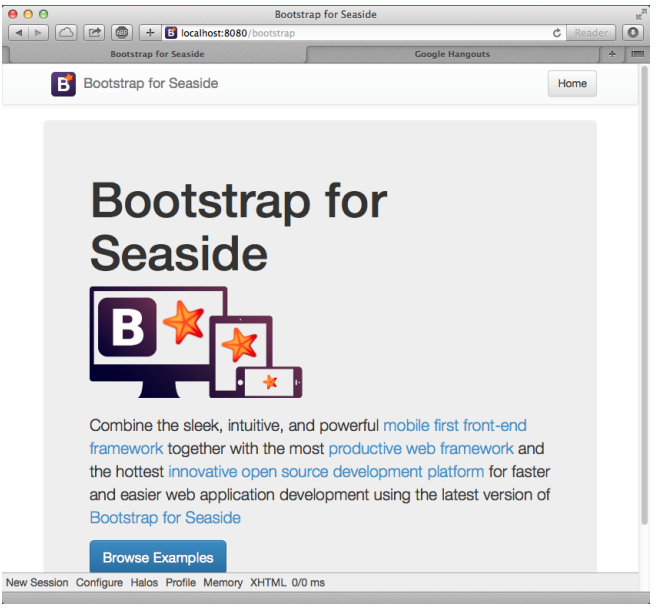

Figure 1.6 Accès à la bibliothèque Bootstrap.

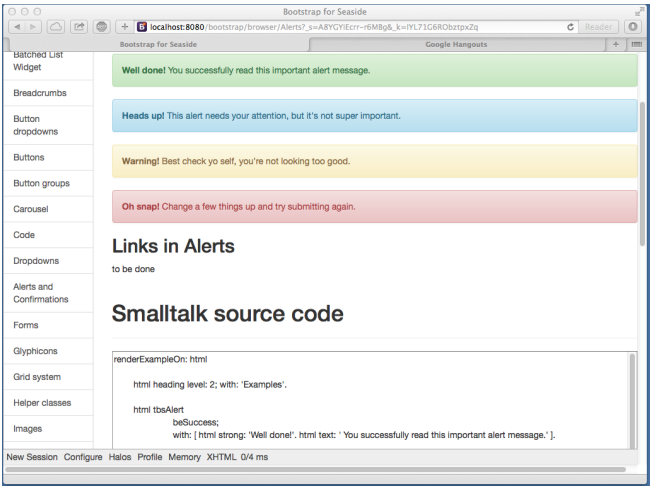

Figure 1.7 Comprendre un élément et son code en Bootstrap.

1.8 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.

```
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 1.8)

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 paramétrable à l'aide d'une variable d'instance dans la classe `TBBlog` et le header pourrait afficher ce titre.

1.9 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` que 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.
```

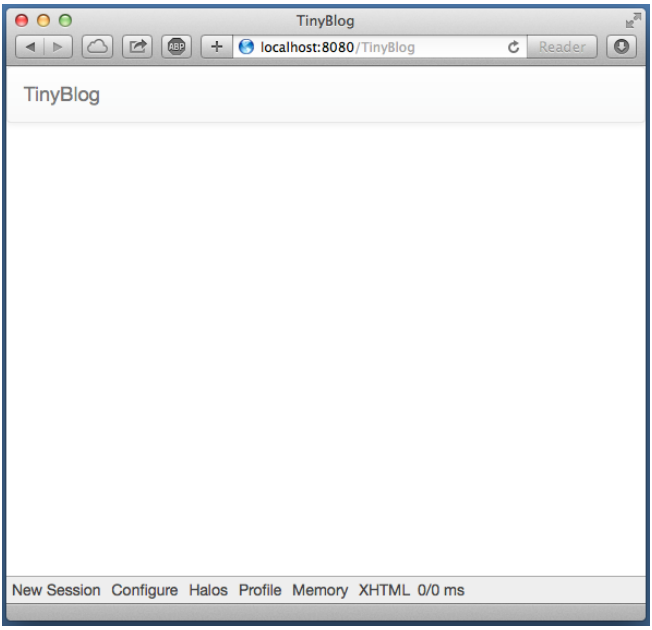

Figure 1.8 TinyBlog avec une barre de navigation.

Relation composite-composant

En Seaside, les sous-composants d'un composant doivent être retournés par le composite en réponse au message `children`. Définissons que l'instance du composant `TBHeaderComponent` est un enfant de `TBScreenComponent` dans la hiérarchie des composants Seaside (et non entre classes Pharo). Nous faisons cela en spécialisant la méthode `children`.

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

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

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

1.10 Utilisation du composant Screen

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

d'instance dans la classe `TBApplicationRootComponent`. Nous l'initialisons dans la méthode `initialize` suivante.

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

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

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

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

Si vous faites un rafraichissement de l'application dans votre navigateur web vous devez voir la Figure 1.8.

Amélioration possibles

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

1.11 Liste des posts

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

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

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

Ajoutons une méthode `renderContentOn:` (protocole rendering) provisoire pour tester l'avancement de notre application (voir Figure 1.9).

```
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 modifions-la ainsi:

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

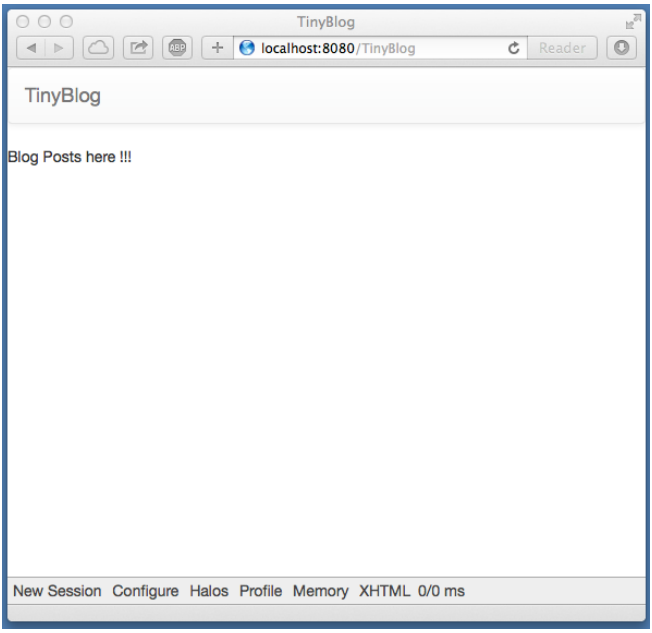

Figure 1.9 TinyBlog avec une liste de posts plutot élémentaire.

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

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

1.12 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 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 et celui sur l'interface d'administration et qui utilise Magritte , nous montrerons qu'il est rare de définir un composant de manière aussi manuelle. En effet, Magritte permet de décrire les données manipulées et offre ensuite la possibilité de générer automatiquement des composants Seaside. Le code équivalent serait comme suit:

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

1.13 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 de votre navigateur et vous devez obtenir un message d'erreur.

1.14 Debugger 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

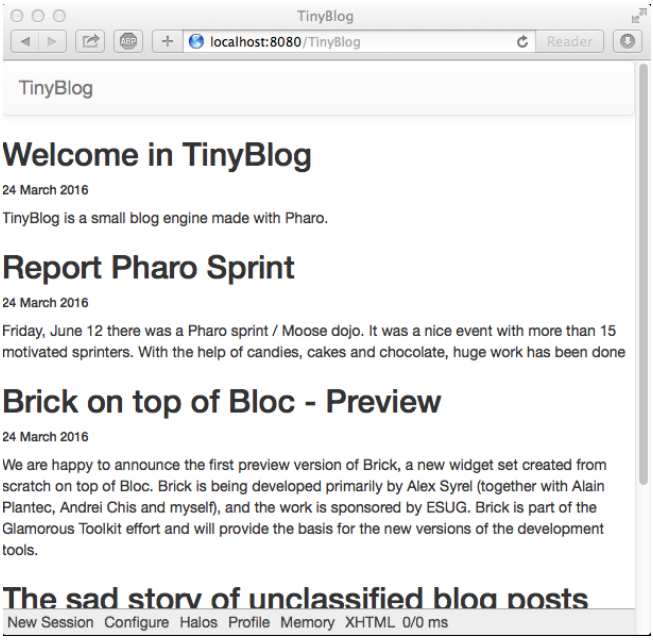

Figure 1.10 TinyBlog avec une liste de posts.

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 1.10.

1.15 Affichage de la liste des posts avec Bootstrap

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

```
TBPostsListComponent >> renderContentOn: html
   super renderContentOn: html.
```

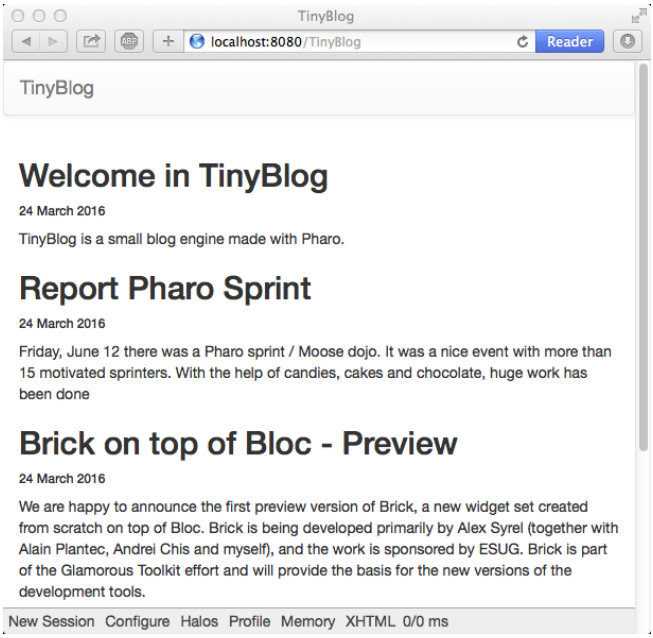

Figure 1.11 TinyBlog avec une liste de posts élémentaire.

```
html tbsContainer: [
self blog allVisibleBlogPosts do: [ :p |
   html render: (TBPostComponent new post: p) ] ]
```

Rafraichissez la page et vous devez obtenir la Figure 1.11.

1.16 Affichage des posts par catégorie

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

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

Definition d'un composant pour les catégories

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

```
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 pour stocker la catégorie courante dans `TBPostsListComponent`.

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

TBScreenComponent >> currentCategory
   ^ currentCategory

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

1.17 Rendu des catégories

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

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

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

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

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

1.18 Mise à jour des Posts

Pour cela, modifions 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 sélectionné 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
               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 1.12).

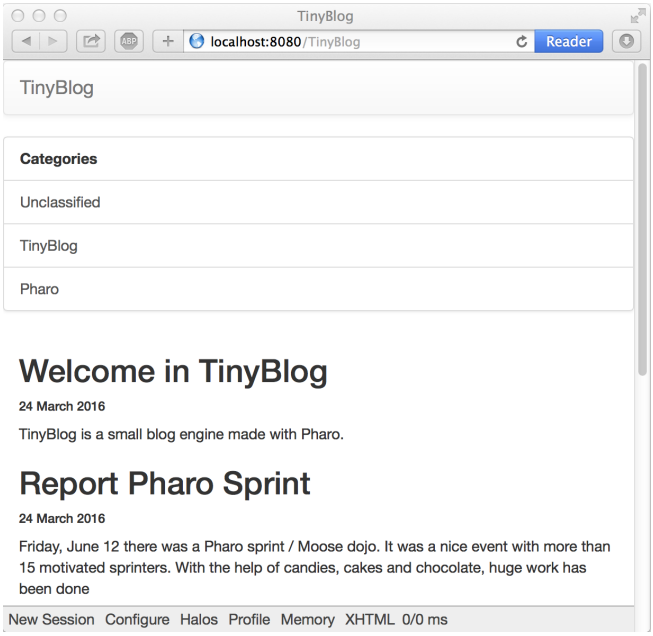

Figure 1.12 Catégories afin de sélectionner les posts.

1.19 Look et agencement

Nous allons maintenant agencer le composant `TBPostsListComponent` en utilisant une mise en place d'un 'responsive design' pour la liste des posts. Cela veut dire que le style CSS va adapter les composants à l'espace disponible.

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

```
TBPostsListComponent >> renderContentOn: html
   super renderContentOn: html.
   html tbsContainer: [
      html tbsRow showGrid;
         with: [
            html tbsColumn
               extraSmallSize: 12;
               smallSize: 2;
               mediumSize:  4;
               with: [
```

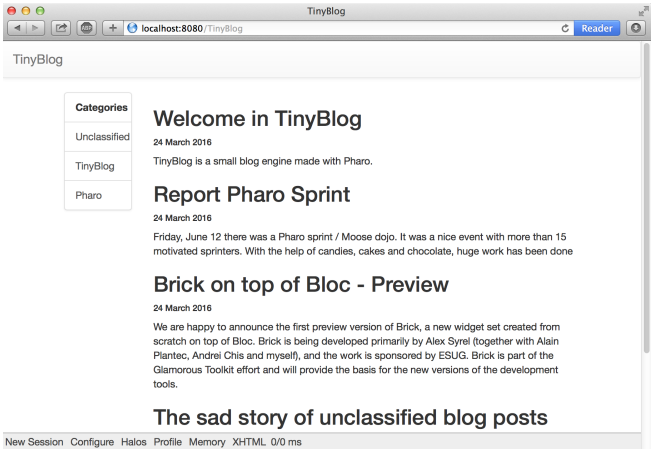

Figure 1.13 Avec un meilleur agencement.

```
                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 1.13.

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

1.20 Notre solution: Plein de petites méthodes

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

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

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

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

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

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

1.21 Futures évolutions

A titre d'exercice, vous pouvez :

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

L'application finale devrait ressembler à la figure 1.14.

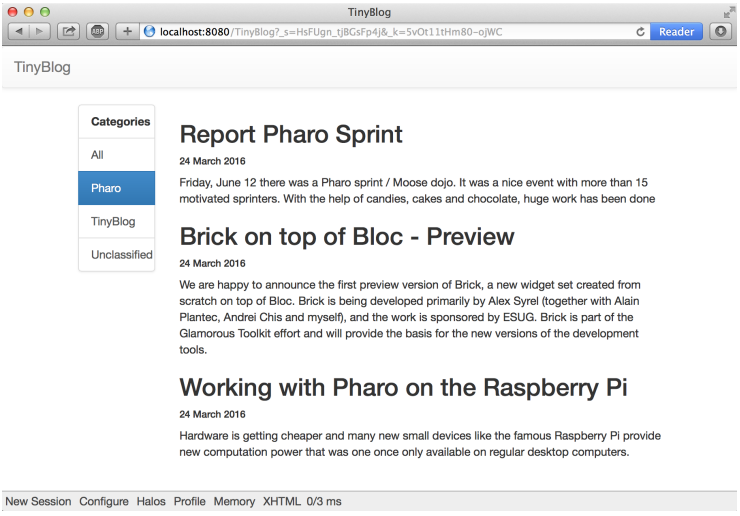

Figure 1.14 Final TinyBlog Public UI.