CHAPTER 1

Interface Web d'administration 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: #week4solution;
    configuration: 'TinyBlog';
    load
```

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

```
ZnZincServerAdaptor startOn: 8080.
```

Vous devrez également faire :

```
TBBlog reset ;
    createDemoPosts
```

1.2 Décrire les données métiers avec Magritte

Magritte est une bibliothèque qui permet une fois les données décrites de générer diverses représentations ou opérations (telles des requêtes). Couplé avec Seaside, Magritte permet de générer des formulaires et des rapports. Le

logiciel Quuve de la société Debris Publishing est un brillant exemple de la puissance de Magritte: tous les tableaux sont automatiquement générés (voir http://www.pharo.org/success). La validation des données est aussi définie au niveau de Magritte au lieu d'être dispersée dans le code de l'interface graphique. Ce chapitre ne montre pas cet aspect.

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

Dans ce chapitre, nous allons décrire les cinq variables d'instance de l'objet `TBPost` à l'aide de Magritte. Dans le chapitre suivant, nous tirerons avantage de ces descriptions pour générer automatiquement des composants Seaside.

Descriptions

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

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

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

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

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

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

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

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

1.3 Administration de TinyBlog

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

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

Composant d'identification

Nous allons commencer par développer un composant d'identification qui lorsqu'il sera invoqué ouvrira une 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 :

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

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

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

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

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

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

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

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

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: [ ... ]
```

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'.

```
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 1.1.

Administration des posts

Nous allons développer deux composants. Le premier sera un rapport qui contiendra tous les posts et le second contiendra ce rapport. Le rapport étant généré par Magritte sous la forme d'un composant Seaside, nous aurions pu n'avoir qu'un seul composant. Toutefois, nous pensons que distinguer le composant d'administration du rapport est une bonne chose pour l'évolution de la partie administration. Commençons donc par le composant d'administration.

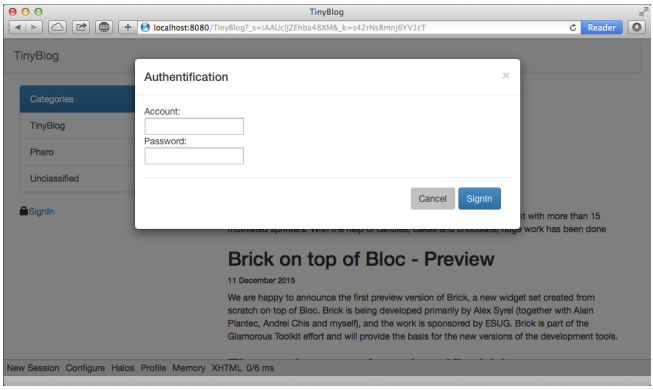

Figure 1.1 Avec un meilleur agencement.

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 ]
```

Pour tester, identifiez-vous sur l'application et vous devez obtenir la situation telle que représentée par la figure 1.2.

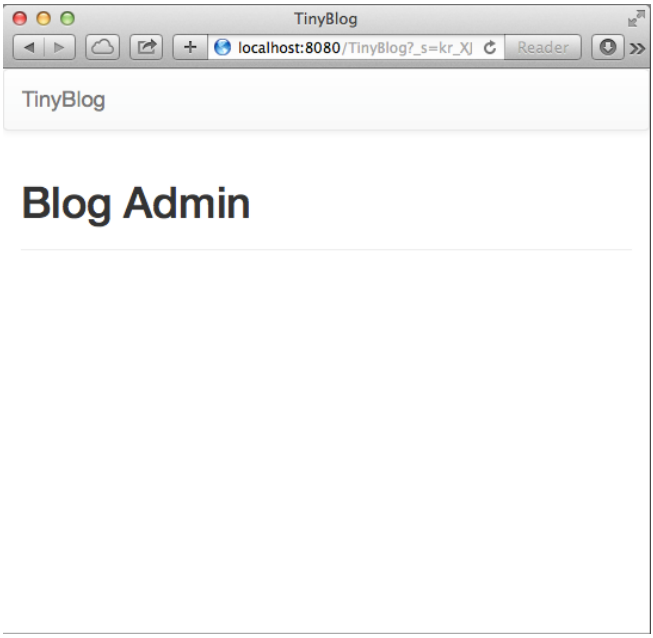

Figure 1.2 Un composant d'administration vide.

Le composant 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.
   ^ self rows: allBlogs description: allBlogs anyOne
    magritteDescription
```

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

TBAdminComponent.

```
TBAdminComponent >> report
   ^ report

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

Comme le rapport est un composant fils du composant admin nous n'oublions pas de redéfinir la méthode `children` comme suit.

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

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

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

Nous pouvons maintenant afficher le rapport.

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

Par défaut, le rapport affiche l'intégralité des données présentes dans chaque 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 accessor selector ]

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

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;
      accessor: #title;
      beRequired;
      yourself
```

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

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

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

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

Identifiez-vous et vous devez obtenir la situation telle que représentée par la figure 1.3.

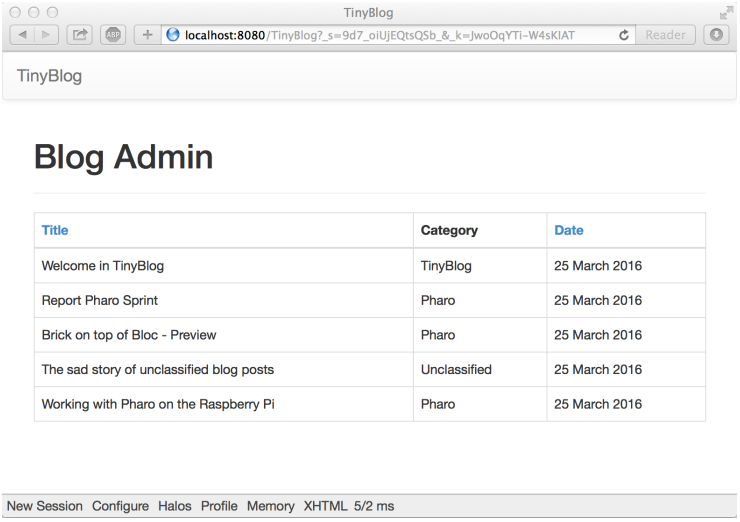

Figure 1.3 Administration avec un rapport.

Gestion des posts

Nous pouvons maintenant mettre en place un CRUD (Create Read Update Delete) permettant de gérer les posts. Pour cela, nous allons ajouter une colonne (instance `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: 'blog'
    classVariableNames: ''
    category: 'TinyBlog-Components'

TBSMagritteReport >> blog
   ^ blog

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

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

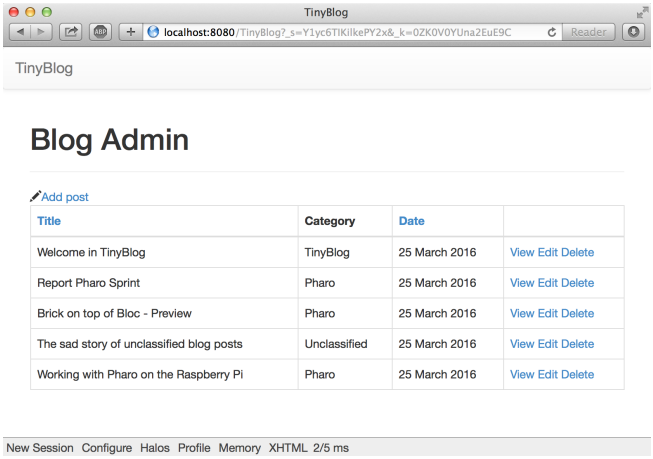

Figure 1.4 Ajout d'un post.

```
    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 redéfinir la méthode `renderContentOn:` du composant `TBPostsReport` pour insérer le lien `add`.

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

Identifiez-vous à nouveau et vous devez obtenir la situation telle que représentée par la figure 1.4.

Implémentation des actions CRUD

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

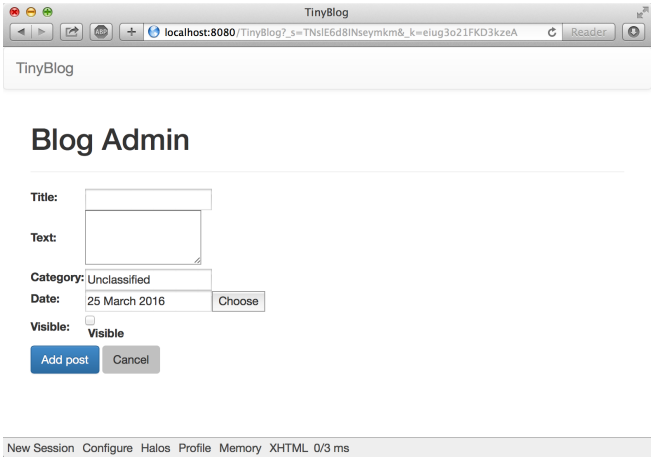

Figure 1.5 Ajout d'un post.

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

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

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

Enregistrez-vous et vous devez obtenir la situation telle que représentée par la figure 1.5.

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

TBPostsReport >> editPost: aPost
```

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

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

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

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

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

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

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

ainsi qu'un test unitaire :

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

Gérer le problème du rafraîchissement des données

Les méthodes `TBPostsReport>>addPost:` et `TBPostsReport>>deletePost:` font bien leur travail mais les données à l'écran ne sont pas mises à jour. Il faut donc 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 c'est-à-dire que le formulaire assure par exemple que les champs déclarés comme obligatoire dans les descriptions Magritte sont bien renseignés.

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

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

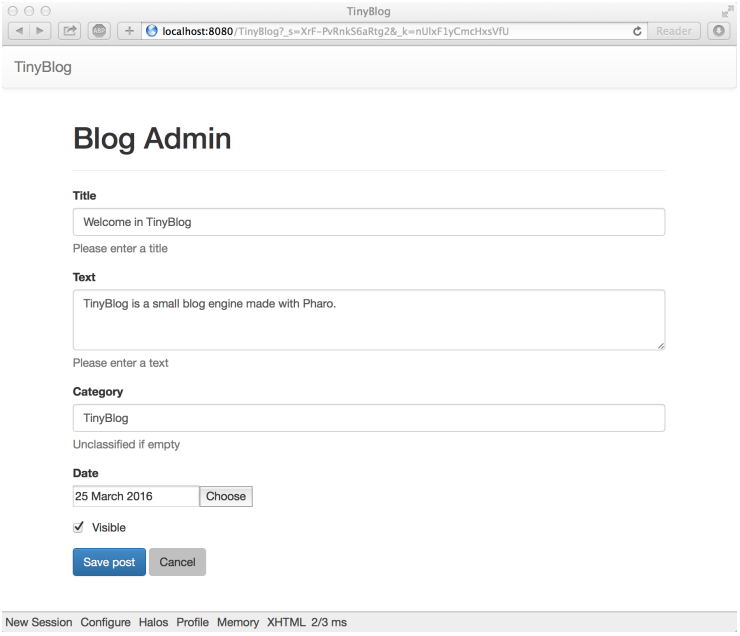

Figure 1.6 Formulaire d'ajout d'un post avec Bootstrap.

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

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

Le formulaire d'édition d'un post doit maintenant ressembler à celui de la figure 1.6.

Gestion de Session

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

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

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

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

TBSession >> logged
    ^ logged

TBSession >> logged: anObject
    logged := anObject

TBSession >> isLogged
    ^ self logged
```

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

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

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

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

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

```
TBPostsListComponent >> 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 gotoAdministration ]
```

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

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

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

Améliorations possibles

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

- gérer plusieurs blogs