CHAPTER 1

Previous Week Solution

You can load the solution of the previous week using the following snippet:

```
Gofer new
   smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
   package: 'ConfigurationOfTinyBlog';
   load.
#ConfigurationOfTinyBlog asClass loadWeek4Correction
```

To test the code, you should start the Seaside HTTP server using the Seaside Control Panel tool (cf. previous week) or directly execute the following code:

```
ZnZincServerAdaptor startOn: 8080.
```

You might also need to create some posts:

```
TBBlog reset ;
  createDemoPosts
```

Before continuing, stop the Teapot server:

```
  TBTeapotWebApp stop
```

CHAPTER 2

Describe your Model Data with Magritte

Magritte is a library to describe data. Using Magritte descriptions, you can then generate various representations for your data or operations such as requests. Combined with Seaside, Magritte enable HTML forms and reports generation. The Quuve software (cf. http://www.pharo.org/success) of Debris Publishing company is a brilliant example of Magritte powerfulness: all HTML tables have been automatically generated. Data validation is also defined in Magritte descriptions and not spread in the UI code. This tutorial will not describe this but you can refer to the Seaside book (http://book.seaside.st) and the Magritte tutorial (https://github.com/SquareBracketAssociates/Magritte).

This week, we will start by describing with Magritte the five instance vairables of `TBPost`, then we will use these descriptions to automatically generate Seaside components.

2.1 Magritte Descriptions

The five following methods classified in the 'descriptions' protocol of `TBPost`. Note that the name of these methods does not matter although we use naming convention. In fact, the `<magritteDescription>` pragma allows Magritte to retrieve descriptions.

A post title is a string and must be filled (required).

```
TBPost >> descriptionTitle
   <magritteDescription>
   ^ MAStringDescription new
```

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

The text of a post is a multi-line string that is also mandatory.

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

The category of a post is an optional string. If it is not specified, the post will belong to the 'Unclassified' category.

```
TBPost >> descriptionCategory
   <magritteDescription>
   ^ MAStringDescription new
      accessor: #category;
      yourself
```

The creation date of a post is important to sort posts before displaying them.

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

The `visible` instance variable must be a boolean value.

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

2.2 Possible Enhancements

We could improve these descriptions and make them more complete. For example, ensure that the date of a new post cannot be before the current date. We could also define the category of post must be one the already existing categories. With richer descriptions, you can produce more complete generated UI elements.

CHAPTER 3

Administration UI of TinyBlog

We will now develop the Administration UI of TinyBlog. Through this exercise, we will show how to use session information and Magritte descriptions to define reports. Our objective is: the user should be able to log in using a login and a password to access the administration part of TinyBlog. The link to log in will be placed below the list of categories.

3.1 Authentification Component

Let's start by developing an authentification Component that will open a modal dialog asking for a login and a password. Note that such a functionnality should be part of a component library of Seaside.

This component illustrates how values can be elegantly retrieved from the user directly the instance variable of the component.

```
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
```

The `component` instance variable is initialized by the following class-side method:

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

The `renderContentOn:` method define the content of the modal dialog.

```
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'.
      ] ] ] ] ] ]
```

Criticism

What do you think about the above method? Propose a refactoring using multiple methods to better modularize the differents elements of the form.

Our 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
   renderOkCancelOn: html
```

```
      html tbsButton
       attributeAt: 'type' put: 'button';
       value: 'Cancel'.
      html tbsSubmitButton
         bePrimary;
         callback: [ self validate ];
         value: 'SignIn'
```

When the user click on the 'SignIn' button, the `validate` message is sent and it verifies the login/password entered by the user to access the 'admin' part.

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

Criticism

Authentication should not be the responsability of the modal dialog. It would be better that it delegates this task to another model object that interact the backend to authenticate users. You can look for another method to achieve user authentication (using a database backend, LDAP or simply text files).

Moreover, the `TBAuthentificationComponent` component could display the name of the currently connected user.

Integrate Authentication

We now integrate a link in the application that will trigger the display of the authentication modal dialog. At the beginning of the `renderContentOn:` method of `TBPostsListComponent`, we add the render of `TBAuthentificationComponent`. We also pass to this component a reference to the component that display the posts.

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

We define a method that displays a key logo and the 'SignIn' link.

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

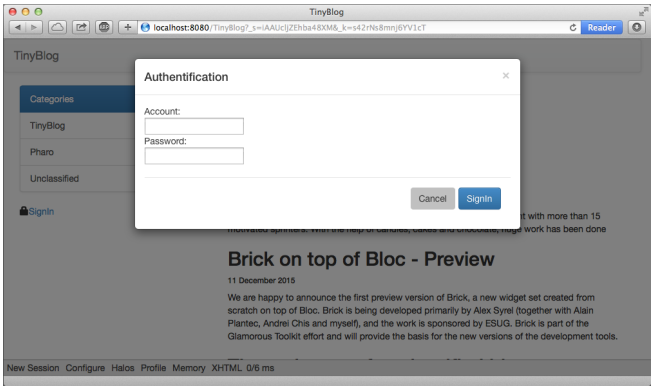

Figure 3.1 Authentication Modal Dialog.

We introduce this link below the list of categories.

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

Figure 3.1 shows what is displayed when the user click on the 'SignIn' link.

3.2 Administration of Posts

We will create two components. The first one will be a report that contains all posts and the second one will contain this report. The report will be automatically generated with Magritte as a Seaside component and we could have only one component. However, we believe that sperating the administration component from the report is a good pratice regarding for evolution. Let's start by the administration component.

Creating the Administration Component

`TBAdminComponent` inherit from `TBScreenComponent` to benefit from the header and access to the blog model. It will contain the report that will create in the following.

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

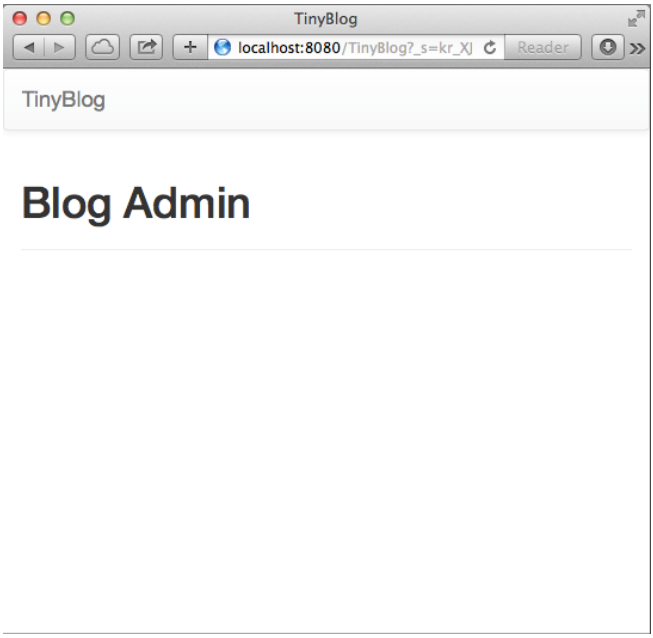

Figure 3.2 An Empty Administration Component.

```
    category: 'TinyBlog-Components'
```

We define a first testing version of the `renderContentOn:` method:

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

We modify the `validate` method to invoke the `gotoAdministration` method defined in `TBPostsListComponent`. This latter method calls the administration component.

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

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

Figure 3.2 illustrates what you obtain after logging in into your application.

The Report Component

The list of posts is displayed by a dynamically generated report with Magritte. We use Magritte here to create the fonctionnalities of the administration part of TinyBlog (list, create, edit and remove posts). For modularity purpose, we create a Seaside component for the report.

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

We add a class-side method named `from:` and pass it the blog object to use to create the report. Since all posts have the same magritte descriptions, we use one post object to retrieve them.

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

We can now, add a report to the `TBAdminComponent`.

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

```
TBAdminComponent >> report
   ^ report

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

Since the report is a child component of `TBAdminComponent`, we must redefine the `children` method as follows.

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

In the `initialize` method of `TBAdminComponent` we instanciate a `TBPostsReport` and pass it the current blog object to access posts.

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

We can now display the report in the `renderContentOn:` method.

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

```
      html horizontalRule.
      html render: self report ]
```

By default, the report display all data available in posts even if some columns are not useful. We can filter columns and only display the title, the category and the creation date.

We add a class-side method on `TBPostsReport` to select columns and we modify the `from:` methode to use it.

```
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)
```

Improve the Report

Currently, the generated report is raw. There are no titles on columns, columns order is not fixed (it can change from instance to another). We will modify Magritte descriptions of posts to improve this.

```
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
```

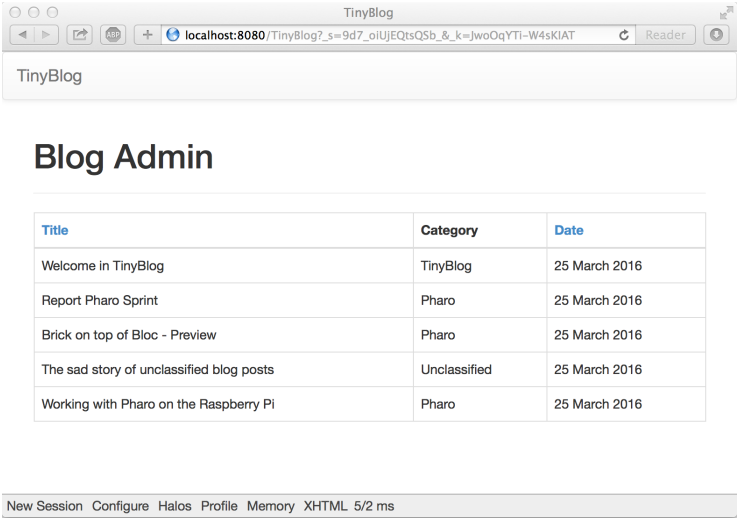

Figure 3.3 Administration of Posts with a Report.

```
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
```

Figure 3.3 shows what the report looks like after logging in.

Manage Posts

We now set up CRUD (Create Read Update Delete) actions to let administrators manage posts. We will add a enew colum (instance of `MACommandColumn`) in the report that will group all operations on posts using `addCommandOn:`. This is done during the report creation and we modify the report to have access to the blog.

```
TBSMagritteReport subclass: #TBPostsReport
  instanceVariableNames: 'report 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';
    yourself;
    addCommandOn: report selector: #editPost: text: 'Edit'; yourself;
    addCommandOn: report selector: #deletePost: text: 'Delete';
    yourself).
   ^ report
```

A link is displayed above the report to add a post (add). Since this link is part of the `TBPostsReport` component, we redefine its `renderContentOn:` to introduce this `add` link.

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

Figure 3.4 shows the new version of the posts report.

Implementing CRUD Actions on Posts

Each action (Create/Read/Update/Delete) is associated to one method of the `TBPostsReport` objet. We will detail the implementation of each of them that consists in creating a customized form for each action. Indeed, if the user wants to read a post, it does not need a 'save' button that is only needed when editing the post.

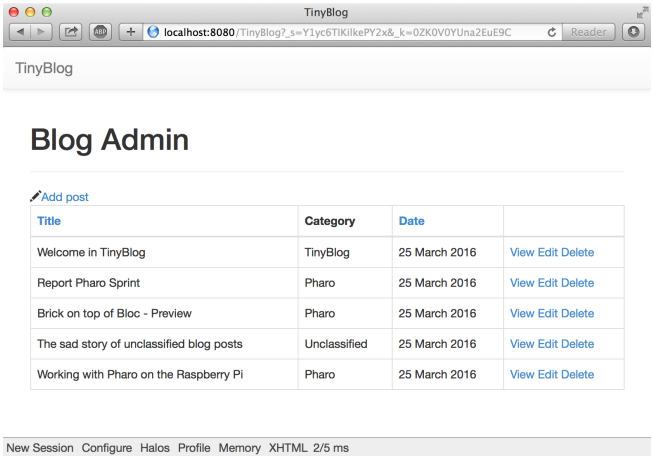

Figure 3.4 Link to Add a Post.

Adding a Post

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

The `renderAddPostForm` method demonstrates the power of Magritte to generate forms. In this example, the `asComponent` message sent to a model object (instance of `TBPost`) directly creates a Seaside component. By adding a decoration to this Seaside component, we can introduce the ok/cancel buttons.

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

The `addPost` method first displays generated form component returned by `renderAddPostForm:` and then add the newly created post to the blog.

Figure 3.5 shows the form to add a post.

Edit a Post

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

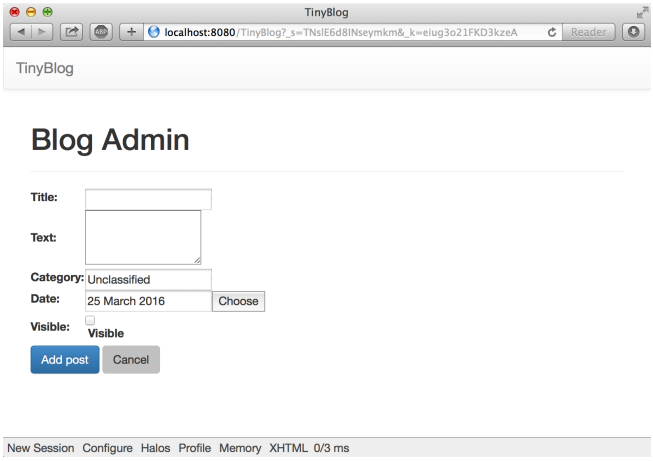

Figure 3.5 Form to Add a Post.

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

Read a Post

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

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

Remove a Post

To prevent mistakes, we introduce a modal dialog to make the user confirm a post removal. Once removed, the list of posts displayed by the `TBPostsReport` component should be refreshed as we will see in the following.

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

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

```
  self save.
```

The last method above has been added in the `TBBlog` class that belongs to the model of our application. We must write a new unit test to cover this functionnality.

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

Dealing with Data Update

Methods `TBPostsReport >> addPost:` and `TBPostsReport >> deletePost:` correctly modify data in the model (and the database) but the displayed data on screen are not correctly updated. There is a mismatch between data in the model and data displayed by the view. The view (the report) should be refreshed.

```
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 ]
```

Now, the form works well and it also take into account contraints expressed in Magritte descriptions such mandatory fields.

Improve the Form Skin

We will now modify Magritte descriptions to make form generators use Bootstrap. First, we specify that the form should be rendered inside a Bootstrap conainer.

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

We can now, improve the style of the input fields with Bootstrap specific annotations.

```
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
```

```
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
```

Figure 3.6 shows what looks like a form to add a post.

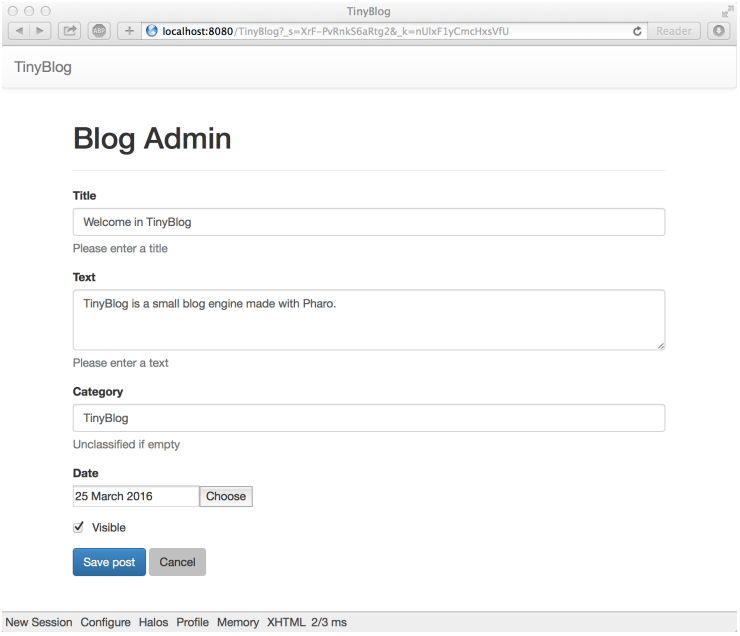

Figure 3.6 Bootstrap-based Generated Form to Add a Post.

3.3 Session Management

A session object is associated to each instance of Seaside application. A session is dedicated to store informations shared and accessible by all components of the application such as the currently authenticated user. We will describe now how to use a session to manage log in.

The blog admin may want to switch between the private (admin) and public (readers) part of TinyBlog.

We introduce a new subclass of `WASession` named `TBSession`. To know wether a user is connected or not, we define a session object with an instance variable named `logged` that contains a boolean value.

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

TBSession >> logged
  ^ logged

TBSession >> logged: anObject
  logged := anObject
```

```
TBSession >> isLogged
  ^ self logged
```

We initialize this instance variable to `false` when the session is created.

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

In the admin part of TinyBlog, we add a link to switch to the public part. We use here the `answer` message because the administration composant has been called using the `call:` message.

```
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.
  ]
```

In the public part, we modify the behavior of the application when the user click on the link to access the admin part. This link only opens the authentication modal dialog if the user is not already connected.

```
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' ]
```

The `TBAuthentificationComponent` component shoudl now update the `logged` instance variable of the session if the user successfully log in as an administrator.

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

Finally, we have to configure Seaside to use session object instance of `TBSession` for the TinyBlog application. This is done in the `initialize` class-side method of `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
```

Before testing, remember that this method must be executed manually `TBApplicationRootComponent initialize`, because the class already exists.

3.4 Possible Enhancements

- Add a ”Disconnect” button
- Manage multiple administrator accounts which implies to improve session management and store the current user login
- Manage multiple blogs