Chapter 12: Supporting User Login

Many web sites and web applications limit access based on a user. We will demonstrate this functionality by identifying some functionality that is available only to registered users.

1. Create a class to model the user.

    a. Create the LBUser class with an id, a name, and a password.

```
Object subclass: #LBUser
   instanceVariableNames: 'id name password'
   classVariableNames: ''
   category: 'LosBoquitas'
```

    b. Create accessors using the class refactoring menu (accepting all the proposed new methods). Note that one of the created methods is 'name1' (to return the 'name' instance variable). This happened because there was already a 'name' method (here it happens to be in a superclass), and the refactoring tool did not want to override the existing method. We do want to override the method, so add a 'name' method.

```
name

   ^ name.
```

    c. To remove the 'name1' method, select 'name1' in the method list, right-click, and select 'remove method…'. If there are any senders of 'name1' you will be asked to confirm the delete.

    d. In general, it is considered a poor practice to store passwords. Instead, applications should store some sort of one-way encryption of the password. To do this, modify the 'password:' method that was generated so that instead of storing the passed-in string, we store a hash of the string. This is (only) a little bit more secure than a free-text string.

```
password: anObject

   password := anObject hash.
```

    e. Add a method to verify passwords.

```
verifyPassword: aString

   ^aString hash = password.
```

    f. Add a method to initialize the values of the instance variables.

```
initialize

   super initialize.
   id := ''.
   name := ''.
   password := 0.
```

g. Add a method to support sorting.

```
<= aUser

   ^self id <= aUser id.
```

h. Define a class instance variable to hold a cache of users (this is similar to what we did on LBEvent to cache events). Make sure that LBUser is selected, and then click on the 'class' button below the class list. This will replace the class definition with a place to define class instance variables.

```
LBUser class
   instanceVariableNames: 'users'
```

i. Add the following *class-side* method to return the user list.

```
users

   users isNil ifTrue: [
      users := IdentitySet with: (self new
         id: 'admin';
         name: 'Site Administrator';
         password: 'passwd';
         yourself).
   ].
   ^users.
```

j. Add the following *class-side* method to lookup a user.

```
userWithID: idString password: passwordString

   ^self users
      detect: [:each | each id = idString and:
         [each verifyPassword: passwordString]]
      ifNone: [nil].
```

2. In LBMain class>>#'initialize' (the initialize method on the class-side of LBMain) we define our application to use WASession. This is an object that holds various session information that is available to all components. We would like to hold a user as part of the session, so we will create a subclass that has an additional instance variable.

   a. Define LBSession with 'user' as an instance variable.

```
WASession subclass: #LBSession
   instanceVariableNames: 'user'
   classVariableNames: ''
   category: 'LosBoquitas'
```

   b. Create instance variable accessors using the class refactoring menu.

   c. Modify LBMain class>>initialize to use this new session class.

```
initialize
"
   LBMain initialize.
"
   super initialize.
   (self registerAsApplication: 'boquitas')
      preferenceAt: #sessionClass put: LBSession;
      yourself.
```

d. Initialize LBMain so that it uses the new class. You can click anywhere on the third line of the method which contains the LBMain initialize comment and press <Ctrl>+<d> (for 'do-it').

3. Create a login component that we can use.

   a. Create a class 'LBLoginComponent' with two instance variables, 'userID' and 'password.'

```
WAComponent subclass: #LBLoginComponent
   instanceVariableNames: 'userID password'
   classVariableNames: ''
   category: 'LosBoquitas'
```

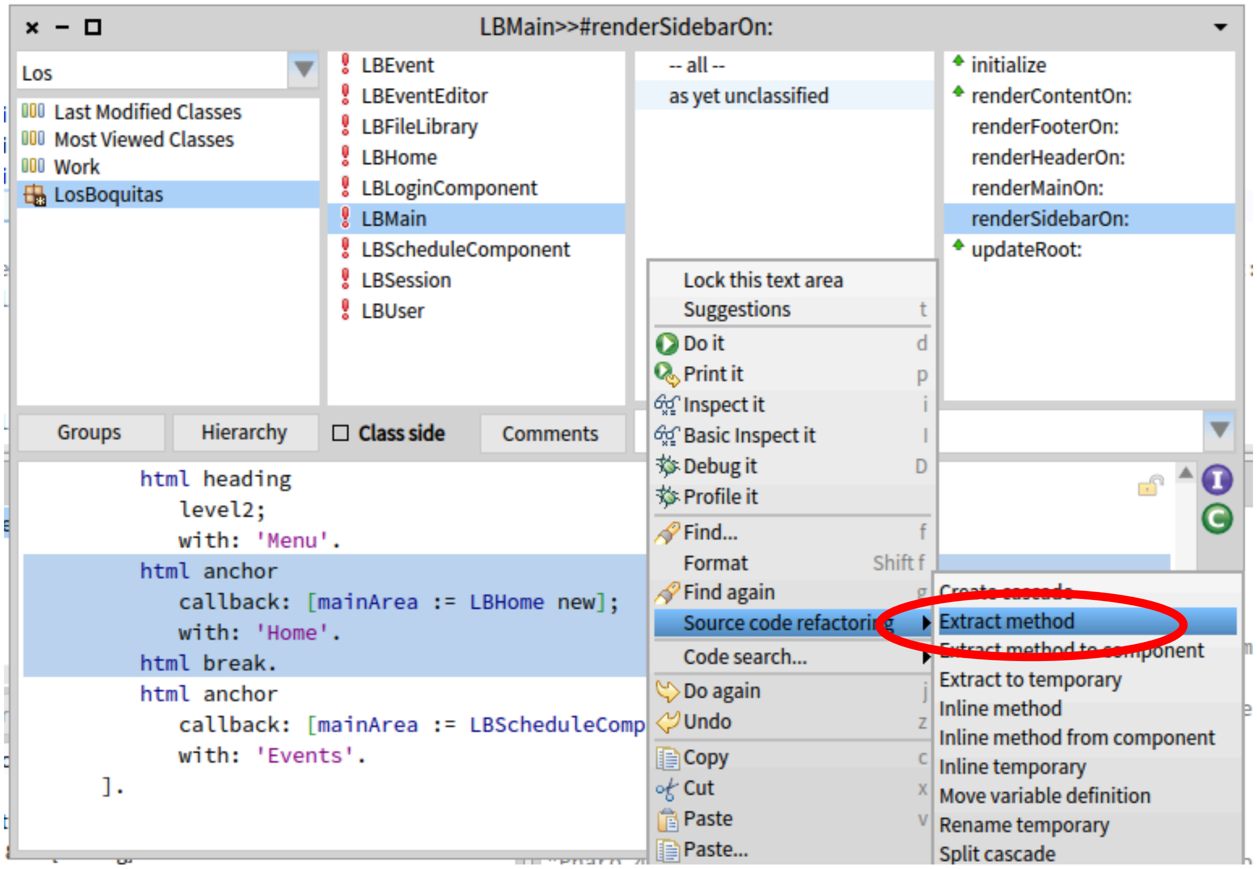
   b. Add a render method to show that it is being called.

```
renderContentOn: html

   html heading: self class name.
```

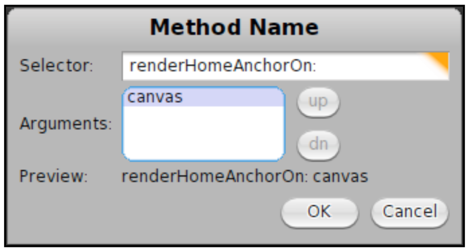
4. Refactor LBMain>>#'renderSidebarOn:' so that we have more small methods rather than a few large methods. This is a much-favored practice in the Smalltalk community.

   a. Select the four lines of code defining the home anchor in the class 'LBMain' and the method #'renderSidebarOn:'. Right-click after selecting the code and select 'refactor source' and then 'extract method.'



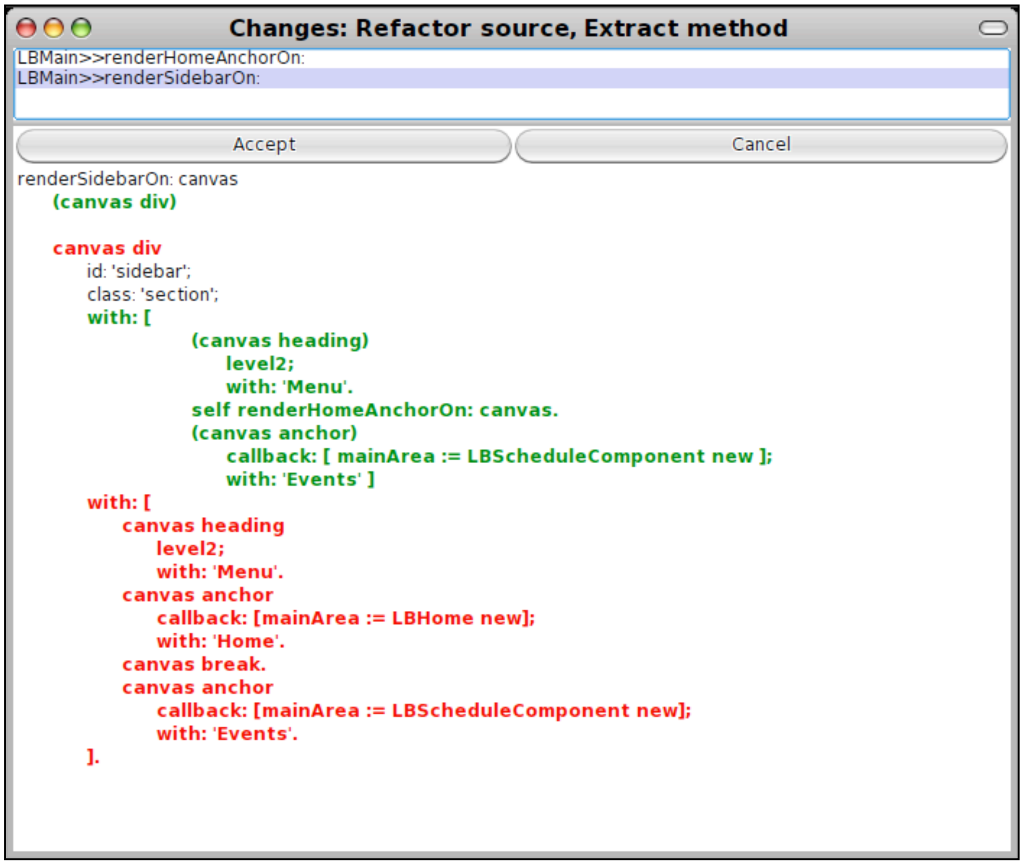
```
        html heading
           level2;
           with: 'Menu'.
        html anchor
           callback: [mainArea := LBHome new];
           with: 'Home'.
        html break.
        html anchor
           callback: [mainArea := LBScheduleComp
           with: 'Events'.
     ].
```

   b. This will pop up a dialog asking for a new name for the method. Enter 'renderHomeAnchorOn: html' as the text and click 'OK' or press <Enter>.



Selector: renderHomeAnchorOn:
Arguments: canvas
Preview: renderHomeAnchorOn: canvas

   c. This will show a Changes dialog on the ExtractMethodRefactoring where you can see two methods involved. One is a new method ('renderHomeAnchorOn:') and the other is a modified method ('renderSideBarOn:'). If you select the modified method you can see the current code (in red type) and code that will be installed if you click the 'accept' button (in green). Note that the refactoring will change the formatting somewhat and overstates the extent of the changes. Go ahead and click ‘accept’.



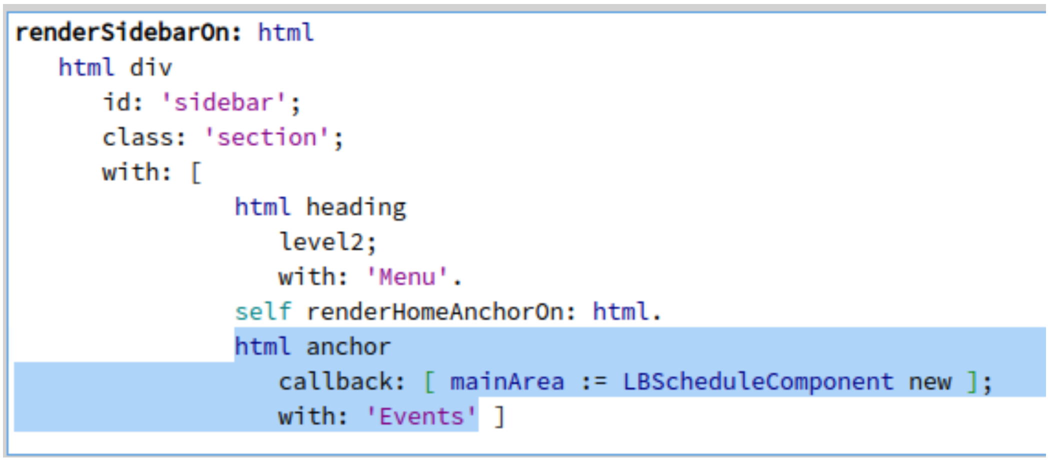
Changes: Refactor source, Extract method

```
LBMain>>renderHomeAnchorOn:
LBMain>>renderSidebarOn:
```

```
renderSidebarOn: canvas
    (canvas div)

    canvas div
        id: 'sidebar';
        class: 'section';
        with: [
                    (canvas heading)
                        level2;
                        with: 'Menu'.
                    self renderHomeAnchorOn: canvas.
                    (canvas anchor)
                        callback: [ mainArea := LBScheduleComponent new ];
                        with: 'Events' ]
        with: [
            canvas heading
                level2;
                with: 'Menu'.
            canvas anchor
                callback: [mainArea := LBHome new];
                with: 'Home'.
            canvas break.
            canvas anchor
                callback: [mainArea := LBScheduleComponent new];
                with: 'Events'.
        ].
```

d. In a similar manner, extract the ‘Events’-link creation code. Note that the refactoring changed the placement of the square brackets in the method. This means that we can no longer select full lines, but must select through the ‘yourself’ but not the closing square bracket. Extract this code into a new method named ‘renderEventsAnchorOn: html’ and accept the changes.



```
renderSidebarOn: html
    html div
        id: 'sidebar';
        class: 'section';
        with: [
                html heading
                    level2;
                    with: 'Menu'.
                self renderHomeAnchorOn: html.
                html anchor
                    callback: [ mainArea := LBScheduleComponent new ];
                    with: 'Events' ]
```

5. Add a ‘Login’ anchor.

   a. First, edit ‘renderEventsAnchorOn:’ to add a break at the end so that subsequent elements are on a new line.

```
renderEventsAnchorOn: html

   html anchor
      callback: [mainArea := LBScheduleComponent new];
      with: 'Events'.
   html break.
```

   b. Next, create a new method very similar to the above to render the login link.

```
renderLoginAnchorOn: html

   html anchor
      callback: [mainArea := LBLoginComponent new];
      with: 'Login'.
   html break.
```

   c. Finally, modify ‘renderSidebarOn:’ to call our new method (and get back our formatting).

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         self
            renderHomeAnchorOn: html;
            renderEventsAnchorOn: html;
            renderLoginAnchorOn: html;
            yourself.
      ].
```

   d. In a web browser, navigate to the application and note the new link. Clicking on the link should show the login component.

6. Add a login form to the login component.

    a. Add various render methods to LBLoginComponent. Note that we can set focus to a particular field using explicit JavaScript. It would be more elegant to use a library (such as jQuery), but this demonstrates the general capability.

```
renderUserOn: html

   | htmlID |
   html div: [
      html label
         for: (htmlID := html nextId);
         with: 'User:'.
      html textInput
         id: htmlID;
         value: userID;
         callback: [:value | userID := value];
         script: 'document.getElementById(' ,
            htmlID printString , ').focus()';
         yourself.
   ].
```

```
renderPasswordOn: html

   | htmlID |
   html div: [
      html label
         for: (htmlID := html nextId);
         with: 'Password:'.
      html passwordInput
         id: htmlID;
         value: password;
         callback: [:value | password := value];
         yourself.
   ].
```

```
warning

   self session user notNil ifTrue: [
      ^'Logged in as ' , self session user name.
   ].
   (userID isNil or: [userID isEmpty]) ifTrue: [
      ^'Please enter User ID and Password'.
   ].

   ^'Login failed!'.
```

```
renderWarningOn: html

   html div: [
      html
         span: '';
         span: self warning;
         yourself.
   ].
```

```
renderSubmitOn: html

   html div: [
      html submitButton
         callback: [self login];
         with: 'Login'.
   ].
```

```
renderFormOn: html

   html form
      class: 'loginForm';
      with: [
         self
            renderUserOn: html;
            renderPasswordOn: html;
            renderWarningOn: html;
            renderSubmitOn: html;
            yourself.
      ].
```

```
renderContentOn: html

   self renderFormOn: html.
```

b. Return to the web browser and click the ‘Login’ link. Note that the formatting is not quite right since the text entry fields are not aligned.

c. Edit LBFileLibrary>>#'boquitasCss' so that the lines beginning with '.eventEditor' now begin with 'form' (referring to the element rather than the class). In this way CSS applies to both forms.

```
form { display: table; }
form > div { display: table-row; }
form > div > * { display: table-cell; }
form textarea { height: 4em; width: 40em; }
form div.hidden { display: none; }
```

d. Refresh the browser and note that the fields are now aligned in a table layout.

e. Clicking the 'Login' button should give an error since the 'login' method is not yet implemented. Add the following method to LBLoginComponent.

```
login

   | user |
   user := LBUser
      userWithID: userID
      password: password.
   self session user: user.
   user notNil ifTrue: [
      userID := nil.
      password := nil.
   ].
```

f. Now try the application again. If you give a wrong user ID/password, you should get a message displayed with that information. If you give a correct user ID/password ('admin' and 'passwd'), you should get a message identifying the logged in user.

g. Modify 'renderContentOn:' so that if a user is logged in we do not display the login form.

```
renderContentOn: html

   self session user isNil ifTrue: [
      self renderFormOn: html.
   ] ifFalse: [
      html heading: 'Welcome, ' , self session user name.
   ].
```

7. Modify LBMain to handle the presence of a session user.

    a. Instead of always rendering a login link, we need an alternative. Add a ‘renderLogoutOn:’ method.

```
renderLogoutAnchorOn: html

   html anchor
      callback: [self session user: nil];
      with: 'Logout ' , self session user name.
```

    b. Modify the ‘renderLoginAnchorOn:’ method to remove the break at the end.

```
renderLoginAnchorOn: html

   html anchor
      callback: [mainArea := LBLoginComponent new];
      with: 'Login'.
```

    c. Create a ‘renderUserOn:’ method that will call one or the other of the above methods.

```
renderUserOn: html

   self session user isNil ifTrue: [
      self renderLoginAnchorOn: html.
   ] ifFalse: [
      self renderLogoutAnchorOn: html.
   ].
   html break.
```

    d. Modify the ‘renderSidebarOn:’ method so that it calls the ‘renderUserOn:’ method instead of the ‘renderLoginAnchorOn:’ method.

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         self
            renderHomeAnchorOn: html;
            renderEventsAnchorOn: html;
            renderUserOn: html;
            yourself.
      ].
```

    e. Try the application with various combinations of correct and incorrect passwords.

8. Restrict some features to logged-in users.

    a. Modify LBScheduleComponent>>#renderContentOn: as follows:

```
renderContentOn: html

   listComponent rows: LBEvent events asSortedCollection.
   html render: listComponent.
   self session user notNil ifTrue: [
      html anchor
         callback: [self add];
         with: 'Add'.
   ].
```

    b. Try the application when logged in and when not logged in. The ‘Add’ link should appear and disappear based on the user state.

    c. Modify LBScheduleComponent>>#initialize as follows:

```
initialize

   | columns |
   super initialize.
   columns := OrderedCollection new
      add: self whoReportColumn;
      add: self whatReportColumn;
      add: self whenReportColumn;
      add: self whereReportColumn;
      yourself.
   self session user notNil ifTrue: [
      columns add: self actionReportColumn.
   ].
   listComponent := WATableReport new
      columns: columns;
      rowPeriod: 1;
      yourself.
```

    d. Try the application when logged in and when not logged in. The ‘delete’ link should appear and disappear based on the user state.

9. Change the window title based on the subcomponent being viewed.

   a. As we discovered in chapter 4, the #updateRoot method provides a way to set the title of the web browser window (and/or tab) to something appropriate for the page being displayed. Now we have a main component (LBMain) and three subcomponents (LBHome, LBLoginComponent, and LBScheduleComponent). Add LBHome>>#updateRoot as follows:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: anHtmlRoot title , ' -- Home'.
```

   b. Add LBLoginComponent>>#updateRoot as follows:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: anHtmlRoot title , ' -- Login'.
```

   c. Add LBScheduleComponent>>#updateRoot as follows:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: anHtmlRoot title , ' -- Event'.
```

   d. Try navigating to the various subcomponents and observe that the title does not change. This is because Seaside is rendering the page head element (which contains the page title element) before it discovers what components will be included in the page which are in the page body element. To provide a solution to this problem, Seaside inquires of each component for a list of 'children' before it starts rendering. This allows the subcomponents to participate more fully in preparing the page. Add LBMain>>#children as follows:

```
children

   ^super children , (Array with: mainArea).
```

   e. Now try navigating to the various subcomponents and observe that the title does change to match the current subcomponent.

      While the #children method is not necessary to process callbacks in Seaside 3.0 (as it was in earlier version of Seaside), it is considered good practice to do so. The absence of the #children method can cause subtle errors where components are displayed but seem to be ignored for some purposes.

10. Save your Pharo image.