

CS 683 Emerging Technologies
Fall Semester, 2004
Doc 14 Seaside Login
Contents

References.....	2
Seaside Login	3
Using Built-in Authentication system.....	3
Modifying the Existing System.....	5
Your own Password System	8
Task & Login	11
Session & Login.....	13
Don't Do this.....	16

Copyright ©, All rights reserved. 2004 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (<http://www.opencontent.org/opl.shtml>) license
defines the copyright on this document.

References

Postings to the Seaside mailing list

<http://lists.squeakfoundation.org/listinfo/seaside>

Seaside Mailing list archives

<http://lists.squeakfoundation.org/pipermail/seaside/>

Seaside Login

There are times when we need to have users log in. We can do this in two ways:

- Use built-in authentication system
- Create our own authentication system

Using Built-in Authentication system

To use the built-in authentication system add one of the following class methods (not instance methods) to your component class. The built-in system uses the standard http authentication scheme.

The first method when run will prompt you to set a user name and password, that the user will have to enter to access your component. Before you can access the web application you need to enter those. This method is used by the Seaside config page.

```
initialize
  "self initialize"
  (self registerAsAuthenticatedApplication:
'standardAuthentication')
```

The second method allows you to specify the username and password in code. It just makes explicit what `registerAsAuthenticatedApplication:` does. The class method `initialize` is called automatically when a class is loaded from a file or parcel/package. You may not want the user prompted at that time – for example when you are installing a headless server. In that case you might wish to have the `initialize` method call the method below.

```
initializeUser: aName password: aString
  "self initializeUser: 'Foo' password: 'bar' "
  | application |
  application :=
    self registerAsApplication: 'standardAuthentication'.
  application configuration
    addAncestor: WAAuthConfiguration localConfiguration.

  application preferenceAt: #login put: aName.
  application preferenceAt: #password put: aString.

  ^ application
```

Modifying the Existing System

The problem with the previous methods is that you only have one user and one password. We may need to support many different users. We can modify the existing system to use our class to validate usernames and passwords, thus allowing multiple users and passwords.

Create a subclass of WAAuthConfiguration with the method mainClass. This method just returns the class to use to validate usernames and passwords. (Note using the Seaside configuration pages one can avoid the need for this class.)

```
Smalltalk.Seaside defineClass: #CS683AuthConfiguration
  superclass: #{Seaside.WAAuthConfiguration}
  instanceVariableNames: "
  classInstanceVariableNames: "
```

Instance Method

```
mainClass
  ^ CS683AuthMain
```

Then we need the authorization class. It needs one method `verifyPassword:forUser:.` This method returns true or false depending on if the username & password are correct. The example here only has one username & password. However the could for example open a file and read a list of usernames & passwords.

```
Smalltalk.Seaside defineClass: #CS683AuthMain
  superclass: #{Seaside.WAAuthMain}
  instanceVariableNames: "
  classInstanceVariableNames: "
```

Instance Method

```
verifyPassword: password forUser: username
  ^'foo' = username and: ['bar' = password]
```

Now in your WAComponent subclass and the following class method and use it to registration your class.

```
YourComponent Class>>initialize
  "self initialize"
  | application |
  application := self registerAsApplication: myApp.
  application configuration
    addAncestor: CS683AuthConfiguration localConfiguration.

  ^ application
```

Now when anyone accesses the component they will be required to give a username and password. They will be required to do this only once per session.

Note that the class CS683AuthConfiguration and the configuration in the above method can be replaced by using the configuration page.

Your own Password System

There are some situations where the above solutions are adequate. We may need:

- Know who login in
- Provide more information on the login page
- Only parts of application are password protected

The following is based on a post to seaside mailing list by Michal. The following class implements a login page.

```
Smalltalk.Seaside defineClass: #SeasideLogin
  superclass: #{Seaside.WAComponent}
  instanceVariableNames: 'username password '
  classInstanceVariableNames: "
```

Instance Methods

```
password
  ^password ifNil: ["].
```

```
password: aString
  ^password := aString
```

```
username
  ^username ifNil: ["].
```

```
username: aString
  ^username := aString
```

SeasideLogin Continued

```
renderContentOn: html
  html attributeAt: 'align' put: 'center'.
  html form:
    [(html attributes)
     align: 'center';
     cellpadding: '10'.
     html table:
       [html tableRowWith: 'Username'
        with: [html textInputOn: #username of: self].
        html tableRowWith: 'Password: '
        with: [html passwordInputWithCallback:
              [:value | self password: value]]].
     html
       break;
       submitButtonWithAction: [self handleLogin]]

handleLogin
  self username = 'Bar' ifFalse:[^self].
  self password = 'foo' ifFalse:[^self].
  self answer: username.
```

The above method only handles one user, but can be modified to handle more.

Using SeasideLogin

One can now use SeasideLogin in another component. One can use it with a task or use it in a session.

Task & Login

We can use task to password protect any component. One creates a WATask subclass that first calls the authentication and then calls the component. The WATask is then made the root level component. This is shown below.

With Built-in Authentication

```
Smalltalk.Seaside defineClass: #LoginTask
  superclass: #{Seaside.WATask}
  instanceVariableNames: "
  classInstanceVariableNames: "
```

Class Method

```
canBeRoot
  ^ true
```

Instance Method

```
self authenticateWith: CS683AuthMain new
  during: [self call: HelloWorld new]
```

Here we call a HelloWorld component, but it could be any component. One advantage of this method is that we can pass a parameter to the CS683AuthMain instance if needed. The parameter could be the file containing the passwords or information needed to connect to a database.

With Home Grown Authentication

Here we use the SeasideLogin component to get the username and password.

```
Smalltalk.Seaside defineClass: #LoginTask
  superclass: #{Seaside.WATask}
  instanceVariableNames: "
  classInstanceVariableNames: "
```

Class Method

```
canBeRoot
  ^ true
```

Instance Method

```
go
  | user |
  user := self call: SeasideLogin new.
  self call: HelloWorld new
```

If needed we could pass a parameter to SeasideLogin say to tell it which file to read the password information. “user” is not passed to the HelloWorld instance in this example but the example could easily be modified to do so.

Session & Login

We can have a user login only once per session. To do this create a subclass of WASession with the following two methods.

```
Smalltalk.Seaside defineClass: #CS683Session
  superclass: #{Seaside.WASession}
  instanceVariableNames: 'user '
  classInstanceVariableNames: "
```

Instance Methods

```
loginUser
```

```
  ^user := self mainClass new call: SeasideLogin new
```

```
user
```

```
  user ifNil: [self loginUser].
```

```
  ^user
```

Configuring the Component to use CS683Session

Now given a component must be configured to use the new session. One way to do this is to via the configuration page for the component. When you configure a component toward the bottom of the page you will see a line starting with Session class.

Session Class	Seaside.WASession	override	inherited from
Session Expiry Seconds	600	override	inherited from

On that line click on override. Once you have done this you will be given a list of which type of session to use with the component. Select CS683.

Insuring the User has Logged in

Now in your component to insure the user has logged in and to get the user name in your component use:

```
self session user.
```

If the user has not logged in they will be sent to the login page. If they already have logged in during that session you will just get the user name.

Making the User Login First

If you want to make the user login in before anything else happens you can call `self session user` in your `renderContentOn:` method

```
renderContentOn: html  
  user := self session user.  
Etc.
```

Automatic Login

You can force the user to login before accessing the component automatically by adding the following method to the `CS683Session` class

```
CS683Session>> start: aRequest  
  self loginUser.  
  ^super start: aRequest
```

Before any component is displayed during that session, the login page is displayed.

Don't Do this

One might be tempted to avoid the overhead of using a session by doing:

```
MyComponent>> renderContentOn: html  
  user := self call: SeasideLogin new.  
  Rest of your renderContentOn: code.
```

To not use “call:” in the renderContentOn: method as doing this makes the application unstable.