Controlling the Back Button in SeaSide

Avi Bryant and Stéphane Ducasse

Seaside is a framework for developing sophisticated web applications in Smalltalk. Its most unique feature is its approach to session management: unlike servlet models which require a separate handler for each page or request, Seaside models an entire user session as a continuous piece of code, with natural, linear control flow - pages can call and return to each other like subroutines, complex sequences of forms can be managed from a single method, objects are passed by reference rather than marshalled into URLs or hidden fields - while fully supporting the backtracking and parallelism inherent to the web browser.

Seaside also features a callback-based event model, a "transaction" system for auto-expiring pages, programmer-friendly HTML generation and designer-friendly templates, a system of reusable and embeddable UI components, and handy web-based development tools.

In the previous tutorial we show how you can easily convert an application into a web application because SeaSide way of managing sessions and control flow amongst pages. We also show how SeaSide supports the control of the back button. We suppose that the reader read the previous tutorial.

1. To get started

You need a working installation of Squeak Smalltalk. Version 3.4 is now available for multiple platforms from www.squeak.org .The Comanche webserver for Squeak. A changeset for version 5.0 is available from http://fce.cc.gatech.edu/~bolot/kom/kom-5-0.02May1824.cs.gz. The current release of Seaside 2.21, available from http://beta4.com/seaside2/downloads/latest.html.
A basic familiarity with Smalltalk and with HTML.

Once you have installed Squeak and filed in both Comanche and Seaside 2.21, you need to start the Seaside service running. Seaside provides the `WAKom` class to interface with the Comanche web server. To start up an instance of Comanche configured with Seaside, evaluate the following piece of code in a Workspace: `WAKom startOn: 9090`
Seaside should now be listening for requests on port 9090. Whenever you save and restart this image from now on, the Seaside service will

come up automatically.
Try the following url `http://localhost:9090/seaside/counter/` to access the simple counter that is described in the preceeding tutorial. You should obtain the Figure 1.

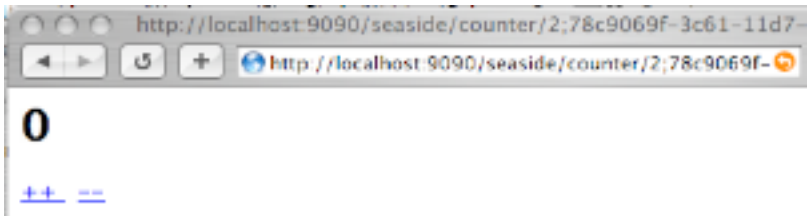

Figure 1

2.The Back Button: The Dark Force

In the earlier tutorial, we looked at the WACounter component, which maintains a simple integer count. Clicking on the ++ or -- links increments and decrements the count. What happens when you use the back button? Try this:

- start a new counter (`http://localhost:9090/seaside/counter`)
- click on ++ until the counter shows 6
- hit the back button to where the counter shows 2
- hit the -- link

You probably expect it to show 1 (2--), but it shows 5 instead. Why? We've been using the *same* instance of this component the whole time, and chronologically, the last value of the counter was 6. We're asking it to decrement, and it does.

This may not seem like a huge problem in the case of the counter, but inconsistency is a Bad Thing. The page was showing 2, and you asked it to do something with (you thought) 2, but it applied the action to 6 instead. If, say, the page was showing a product instead of a number, and the link was "buy" instead of "--", you could have real problems.

3.The SeaSide Way

Seaside solves this by introducing a class for holding state that needs to be backtrack-aware. A `StateHolder` is just like a `ValueHolder`, but it detects use of the back button, and adjusts its state accordingly. To be more specific: whenever a link is clicked on a page, *all* of the

StateHolders for the current session take on the value that they had when that particular page *was produced*.

We can get a StateHolder by asking the current session for one (WASession>>stateHolder). So, to modify *WACounter* to be consistent under backtracking, we must do two things - first, use the "abstract variable" refactoring to replace all references to 'count' with sends to accessor methods, so, for example,

```
WACounter>>increment
  count := count + 1
```

becomes

```
WACounter>>increment
  self count: self count + 1

WACounter>>decrement
  self count: self count - 1
```

(this is something the RB can do for us).

Then, we must make the 'count' instance variable hold a StateHolder object:

```
WACounter>>initialize
  count := session stateHolder.
  self count: 0.
```

Finally, we must implement the accessors:

```
WACounter>>count
  ^ count contents

WACounter>>count: aNumber
  count contents: aNumber
```

If we perform the experiment from before, when we click the -- when showing 2, we should now (correctly) see 2.

4.StateHolder or Not: That is the Question

Although you can use a StateHolder anywhere you like in your code, it should only be necessary to use it in very specific cases. First of all,

you should never use it in your domain model code - it would be very odd for customer data, for example, to disappear just because the user hit the back button. This is *not* meant to be a global state rewind mechanism. Instead, it is meant to be used for *UI* state. Examples of UI state are:

- the current record being shown on a page
- the currently selected page in a batch of search results
- the currently selected pane of a tab panel
- which nodes on a tree widget are expanded

and so on. This is all state that is non-crucial to the domain, but that the user expects to be consistent with what they see on the page, when they click a link.

There are essentially three kinds of UI state you might find in a component. These are:

- immutable state, that is set on initialization of a component and never changes. For example, a form that edits a customer's info will likely be instantiated with the customer it is to edit, and this value never changes.
- transient state, that is reset with every request. For example, the current values of form fields (which will be modelled by inst vars in the component) are changed every time the form is submitted.
- persistent but mutable state, that doesn't fall into either of the previous two categories. It is only this last kind for which StateHolders are necessary.

5. Limiting Backtrack

It is great to be able to provide the abilitiy to safely backtrack, but sometimes you want to prevent your users from backtracking. For example, consider a typical online store, where after filling a shopping cart, users are led through a checkout process. You may well want to be able to backtrack during that process (mistyped the shipping address), but once the credit card has been charged, you don't want the user to be able to backtrack change the contents of the shopping cart!

Seaside provides the `WASession>>isolate:` method to control backtracking. #isolate: is passed a block, and the block is immediately evaluated. When the block ends, any pages that were created within it immediately expire.

If the user backtracks to them and clicks on any link or submits any form, they will be notified that that page has expired and be redirected to a non-expired page. #isolate: calls can also be nested.

Références

Squeak: http://www.squeak.org/
SeaSide: http://beta4.com/seaside2/
ESUG: http://www.esug.org/
Wiki Francophone: http://www.iutc3.unicaen.fr:8000/fsug/
Livres Gratuits :
http://www.iam.unibe.ch/~ducasse/WebPages/FreeBooks.html
ESUG CD : http://www.ira.uka.de/~marcus/EsugcD.html
Squeak CD : http://www.ira.uka.de/~marcus/SqueakCD.html