Seaside: An Innovative Web Application Framework

Damien Cassou, Stéphane Ducasse and Luc Fabresse

http://www.pharo.org
Seaside

- A powerful, innovative and flexible framework
- Dedicated to build complex Web applications
- Live coding and debugging
- Support reusable Web components
- Secure by default
- Web 2.0 support (Ajax, Reef, ...)
- REST integration
Books and Tutorials

- Seaside http://www.seaside.st
- Seaside tutorial
  http://www.swa.hpi.uni-potsdam.de/seaside/
- Seaside tutorial
  http://seaside.gemtalksystems.com/tutorial.html
- TinyBlog tutorial
- Community: register to seaside mailing list and ask questions
Seaside Little History

- Developed by A. Bryant and J. Fitzell
- Enhanced by L. Renggli and P. Marshall
- In production since 2002
- Actively maintained by J. Brichau, S. Eggermont (web site under full rewrite)
- Foundation of many Pharo success stories
- http://www.pharo.org/success
Seaside in a Nutshell

- Define reusable and stateful components
- Use a DSL for rendering components
- Compose components
  - build coarser-grained components by encasputation
  - schedule components with call: and answer: messages
- A web application is just a root component
- Debug your application on the fly
- Use metadata to generate forms
Seaside in Production Since 2002
Cable eXpertise
Seaside Components

- A component is:
  - an instance of a subclass of \texttt{WAComponent}
  - a reusable and stateful part of a Web page
  - rendered in HTML (<div>)

- A Web application has a root component

\texttt{WAAadmin register: \texttt{WACounter asApplicationAt: 'counter'}}.
The Counter Web Application
WACounter subclass: #WACounter
instanceVariableNames: 'count'
classVariableNames: ''
package: 'Seaside−Examples−Misc'.

WACounter >> initialize
   super initialize.
   count := 0

WACounter >> increase
   count := count + 1

WACounter >> decrease
   count := count − 1
From Components to Valid HTML

- All components respond to `renderContentOn`:
- This method converts a component to valid HTML
- This message is automatically sent to components by Seaside
HTML Rendering

- `renderContentOn:` is dedicated to HTML generation
- `parameter named html (WAHtmlCanvas)` defines a DSL like API to generate valid HTML

```plaintext
WACounter >> renderContentOn: html
  html heading: count.
  html anchor
    callback: [ self increase ];
    with: '++'.
  html space.
  html anchor
    callback: [ self decrease ];
    with: '--'
```
Live Debugging

WACounter>>decrease
  self haltIf: (count−1 < 0).
  count := count − 1
Walking the Application Stack
Pressing the **back button** of the browser desynchronizes server and client

Example:

- Increment the counter 5 times \( \text{(count} = 5 \) )
- Press the **back button** \( \Rightarrow \) the displayed value is 4
- Increment the counter \( \Rightarrow \) the displayed value is 6

How to make it work properly?
A Counter Dealing with Back Button

Just declare the component state to be preserved

```
WACounter >> states
  ^ Array with: self
```
Plain Code in Callbacks

```plaintext
WACounter >> renderContentOn: html
    html heading: count.
    html anchor
        callback: [
            count odd
                ifTrue: [ self increase ]
                ifFalse: [
                    self inform: 'Even number!'.
                    count := count + 2]
        ];
        with: '++'.
    html space.
    html anchor
        callback: [ self decrease ];
        with: '––'
```
Callback Execution

Pressing ++

does not change the counter.

shows

Even number!
A Greeter Application

Seaside

Username: Bob

Seaside

Hi Bob

Seaside
Callbacks with the User Value

A Greeter component

``` Smalltalk
Greeter >> renderContentOn: html
    html form: [
        html text: 'Username:'.
        html textInput
            callback: [:value | username := value ].
        html submitButton
            callback: [ self inform: 'Hi ', username ];
        text: 'Say Hello'. ].
```
Did you see?! 

- **No** manual request parsing
- **No** XML configuration files
- **No** file/page
  - don’t think in terms of pages
  - use components
- **No** hardcoding of next page
- **Live Debugging**
  - use the debugger to modify objects and proceed to generate the HTML response
Conclusion

- A Web application = a root component
- A component renders itself in HTML ($\text{renderContentOn;}$)
- An extensible DSL helps to easily generate HTML
A course by

and

in collaboration with

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
https://creativecommons.org/licenses/by-nc-nd/3.0/fr/