Web Applications, Continuations, & Seaside

Andrew P. Black

CS 510 Advanced Programming
Why are Web Apps hard?

- Desktop applications ask the user questions
- Web apps put the web browser in charge
Why are Web Apps hard?

- Desktop applications ask the user questions
- Web apps put the web browser in charge

![Diagram showing the interaction between Application and User over time with call and return arrows.](image)
Why are Web Apps hard?

- Desktop applications ask the user questions
- Web apps put the web browser in charge

![Diagram showing the interaction between Application, User, User's Web browser, and Application Server.]
Why are Web Apps hard?

- Desktop applications ask the user questions
- Web apps put the web browser in charge

No shared state
Consider “logging in”

- “log in” button creates a new Login web page asking for user name and password.

- “submit” button on the Login page passes the results of the fields to a validation routine, which determines if login is successful.

- One of two response pages must be generated and displayed.

Diagram:

User's Web browser

<table>
<thead>
<tr>
<th>login request</th>
</tr>
</thead>
<tbody>
<tr>
<td>login page</td>
</tr>
<tr>
<td>submit</td>
</tr>
<tr>
<td>login OK</td>
</tr>
<tr>
<td>login failed</td>
</tr>
</tbody>
</table>

Application Server
Compare with a desktop application:
Compare with a desktop application:

```plaintext
| user |
user := self attemptAuthentication
user username = ''
  ifFalse: [ self inform: 'Login successful ', user username]
  ifTrue: [ self inform: 'Login failed' ]
```
Compare with a desktop application:

```
| user |
user := self attemptAuthentication
user username = ''
  ifFalse: [ self inform: 'Login successful ', user username]
  ifTrue: [ self inform: 'Login failed' ]
```

- Seaside lets you write more or less the same thing in a web application:
Compare with a desktop application:

```plaintext
| user |
user := self attemptAuthentication
user username = ''
   ifFalse: [ self inform: 'Login successful ', user username]
   ifTrue: [ self inform: 'Login failed' ]
```

• Seaside lets you write more or less the same thing in a web application:
Compare with a desktop application:

```plaintext
| user |
user := self attemptAuthentication
user username = ''
  ifFalse: [ self inform: 'Login successful ', user username]
  ifTrue: [ self inform: 'Login failed' ]
```

- Seaside lets you write more or less the same thing in a web application:

```plaintext
| user |
user username = ''
  ifFalse: [ self inform: 'Login successful ', user username]
  ifTrue: [ self inform: 'Login failed' ]
```
• AuthenticationComponent is also straightforward:

AuthenticationComponent >> renderContentOn: html
   | user |
   user := AuthUser new.
   html form: [
      html paragraph with: [
         html span with: 'Username'.
         html textInput on: #username of: user.
      ].
      html paragraph with: [
         html span with: 'Password'.
         html textInput on: #password of: user.
      ].
      html submitButton callback: [ self answer: user ].
   ].
How does this work?

• The keys are the call: and answer: messages, which save and resume a computation.

• They are implemented using continuations
Continuations in Smalltalk

- Continuations are not “built in” to Smalltalk
  - but Smalltalk has enough reflective capability to build continuations into a library

- **thisContext** is the sixth keyword in Smalltalk
  - What are the other five?
  - **thisContext** answers the current execution context, usually a MethodContext or a BlockContext.
uses of **thisContext**

- Most obvious use is in the debugger:
  - the context objects make up the stack
  - each Context object is linked to the previous one using the `sender` instance variable
- **thisContext** can also be used to implement Continuations
Class Continuation

- let’s look at the implementation
- let’s try some examples using continuations
Seaside

- Presentation based on a chapter from the as-yet-unpublished volume 2 of “Squeak by Example” (on class web page)
How to get Seaside

• The Seaside “one click experience”
  ▶ available from http://www.seaside.st
  ▶ designed for people who don’t already know how to run Squeak.
  ▶ Multi-platform
  ▶ All you really need is the Seaside image
In the Seaside image...

• There is a web server
  ▶ you have to start it!
    ◦ WAKom startOn: 8080.
  ▶ and eventually, stop it
    ◦ WAKom stop.

• Then, point your web browser at it:
  ▶ http://localhost:8080/seaside
Components

• Seaside web pages are built from Components
  ‣ subinstances of WACComponent

• Similar to on-screen GUIs
  ‣ built from subinstances of Morph

• Each Component is responsible for rendering itself onto an HTML “canvass”
  ‣ has application-specific state in its instance vars
Components

• Components are reusable
  ▶ a component can be instantiated many times, in different contexts

• Some components can be top-level “applications”
Components

- Components are reusable
  - a component can be instantiated many times, in different contexts
- Some components can be top-level “applications”
Examples Directory
Examples Directory

- Counter
- Config page
- MultiCounter
Examples Directory

WAPresenter

states

WACounter

count

canBeRoot

initialize

increase

decrease

renderContentOn: html

states

WACOMPONENT

false

canBeRoot

initialize

increase

decrease

renderContentOn: html

states

true

canBeRoot

initialize

increase

decrease

renderContentOn: html

states

Array with: self

Array with: self

self

registerAsApplication

: 'examples/counter'

super initialize.

self count: 0

html heading: count.

html anchor

callback: [ self increase ];

with: '++'.

html space.

html anchor

callback: [ self decrease ];

with: '--'.
Cleint-side Editing

- Toggle Halos gives access to
  - class browser
  - object inspector
  - CSS Style editor
“Hello World” in Seaside

• Define a subclass of WACComponent called WAHelloWorld.

• Implement the renderContentOn: method

  ➤ WAHelloWorld renderContentOn: html
      html text: 'hello world'

• Tell Seaside that WAHelloWorld is an “application”

  ➤ WAHelloWorld class canBeRoot
      true

• Configure seaside to launch the application

  ➤ Point the browser to http://localhost:8080/seaside/config
Backtracking

• When we went back to an earlier counter, the state of the counter was correctly backtracked
  ▶ What makes this happen?

• Each component is sent the message `states`: it answers the objects that should be (shallow) copied into a `WASnaphot`
  ▶ `WACounter>>states` answers `{self}`
• Rendering html is a bit like drawing onto a graphics canvas:
  ▶ each component is responsible for drawing itself
  ▶ the Seaside framework starts the process by creating the html canvas and asking the top-level component to draw itself
renderContentOn: html
  html heading: count.
  html anchor
    callback: [ self increase ];
    with: '++'.
  html space.
  html anchor
    callback: [ self decrease ];
    with: '--'
Multicounter

• **WAMulticounter** has **WACounters** as components

```plaintext
WAMultiCounter
  counters
  canBeRoot
  initialize
  renderContentOn: html
  children

super initialize.
counters := (1 to: 5) collect:
  [ :each | WACounter new ]

counters
do: [ :each | html render: each ]
  separatedBy: [ html horizontalRule ]
```

WACountert

↑ #()

↑ true

↑ counters
WACanvas

• the “html” argument to a rendering method is a WARenderCanvas
  ▶ it provides “brushes” for many html markups