Dynamic Web Development with Seaside

Stéphane Ducasse (based on S. Ducasse, D.C. Shaffer, L. Renggli, and R. Zaccone)

March 12, 2024

Copyright 2018 by Stéphane Ducasse (based on S. Ducasse, D.C. Shaffer, L. Renggli, and R. Zaccone).

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: copy and redistribute the material in any medium or format,
- to **Adapt**: remix, transform, and build upon the material for any purpose, even commercially.

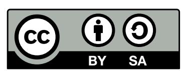
Under the following conditions:

Attribution. You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

Share Alike. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
https://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations **vi**

1 Introduction **1**

1.1 What is Seaside? . . . 1
1.2 Some Seaside Applications . . . 3
1.3 What is Pharo? . . . 4
1.4 Structure of the Book . . . 4
1.5 Formatting Conventions . . . 5
1.6 Acknowledgments . . . 5

I Getting Started

2 Getting Started in Pharo **11**

2.1 Getting Seaside . . . 11
2.2 Starting the Server . . . 11
2.3 A First Seaside Component . . . 13
2.4 Defining Basic Methods . . . 15
2.5 Rendering a Counter . . . 16
2.6 Summary . . . 21

3 Getting Started in Gemstone/S **23**

3.1 Using the GLASS Virtual Appliance . . . 23
3.2 A first Seaside Component . . . 28
3.3 Defining a Component . . . 28
3.4 Defining Some Methods . . . 28
3.5 Rendering a Counter . . . 28
3.6 Registering the application . . . 28
3.7 Adding Behavior . . . 28
3.8 Keeping Up with the Latest Features . . . 28

4 Gemstone version **29**

II Fundamentals

5 Rendering Components **35**
5.1 Brush concept 35
5.2 First component: Hello World 36
5.3 Fun with Seaside XHTML Canvas 37
5.4 Rendering Objects 38
5.5 Brush Structure 41
5.6 Learning Canvas and Brush APIs 43
5.7 Rendering Lists and Tables 46
5.8 Style Sheets 48
5.9 Summary 50

6 CSS in a Nutshell **51**
6.1 CSS Principles 51
6.2 CSS Selectors 52

7 Anchors and Callbacks **57**
7.1 From Anchors to Callbacks 57
7.2 Callbacks 58
7.3 About Callbacks 60
7.4 Contact Information Model 60
7.5 Listing the Contacts 61
7.6 Adding a Contact 62
7.7 Removing a Contact 63
7.8 Creating a mailto: Anchor 65
7.9 Summary 65

8 Forms **67**
8.1 Text Input Fields and Buttons 67
8.2 Convenience Methods 69
8.3 Drop-Down Menus and List Boxes 70
8.4 Radio Buttons 73
8.5 Check Boxes 74
8.6 Date Inputs 76
8.7 File Uploads 78
8.8 Summary 79

III Using Components

9 Calling Components **85**
9.1 Displaying a Component Modally 85
9.2 Example of Call/Answer 85
9.3 Call/Answer Explained 87
9.4 Component Sequencing 88
9.5 Answer to the Caller 88
9.6 Don’t call while rendering 89
9.7 A Look at Built-In Dialogs 90

9.8 Handling The Back Button . . . 90
9.9 Show/Answer Explained . . . 91
9.10 Transforming a Call to a Show . . . 92
9.11 Summary . . . 93

10 Embedding Components **95**
10.1 Principle: Component Children . . . 95
10.2 Example: Embedding an Editor . . . 96
10.3 Components All The Way Down . . . 99
10.4 Intercepting a Subcomponent's Answer . . . 102
10.5 A Word about Reuse . . . 102
10.6 Decorations . . . 103
10.7 Component Coupling . . . 108
10.8 Summary . . . 109

11 Tasks **111**
11.1 Sequencing Components . . . 111
11.2 Hotel Reservation: Tasks vs. Components . . . 113
11.3 Mini Inn: Embedding Components . . . 115
11.4 Summary . . . 115

12 Writing good Seaside Code **117**
12.1 A Seaside Program Checker . . . 117
12.2 Summary . . . 118

IV Seaside in Action

13 A Simple ToDo Application **121**
13.1 Defining a Model . . . 121
13.2 Defining a View . . . 123
13.3 Rendering and Brushes . . . 125
13.4 Adding Callbacks . . . 126
13.5 Adding a Form . . . 128
13.6 Calling other Components . . . 130
13.7 Answer . . . 131
13.8 Embedding Child Components . . . 132
13.9 Summary . . . 134

14 A Web Sudoku Player **135**
14.1 The solver . . . 136
14.2 Sudoku . . . 136
14.3 Rendering the Sudoku Grid . . . 137
14.4 Adding Input . . . 142
14.5 Back Button . . . 144
14.6 Summary . . . 145

15 Serving Files **147**
15.1 Images 147
15.2 Including CSS and Javascript 150
15.3 Working with File Libraries 151
15.4 Example of FileLibrary in Use 154
15.5 Which method should I use? 156
15.6 A Word about Character Encodings 158

16 Managing Sessions **163**
16.1 Accessing the Current Session 163
16.2 Customizing the Session for Login 164
16.3 LifeCycle of a Session 166
16.4 Catching the Session Expiry Notification 167
16.5 Manually Expiring Sessions 169
16.6 Summary 170

17 A Really Simple Syndication **171**
17.1 Creating a News Feed 172
17.2 Render the Channel Definition 172
17.3 Rendering News Items 173
17.4 Subscribe to the Feed 174
17.5 Summary 175

18 REST **177**
18.1 REST in a Nutshell 177
18.2 Getting Started with REST 179
18.3 Matching Requests to Responses 180
18.4 Handler and Filter 185
18.5 Request and Response 186
18.6 Advices and Conclusion 187

19 Deployment **189**
19.1 Preparing for Deployment 189
19.2 Deployment with Apache 192
19.3 Serving File with Apache 198
19.4 Load Balancing Multiple Images 199
19.5 Using AJP 201
19.6 Maintaining Deployed Images 202
19.7 Deployment tools 203
19.8 Request Handler 204
19.9 Summary 205

20 JQuery **207**
20.1 Getting Ready 207
20.2 JQuery Basics 208
20.3 Creating Queries 210
20.4 Refining Queries 211

20.5 Performing Actions 212
20.6 Adding JQuery 214
20.7 AJAX 215
20.8 How tos 216
20.9 Enhanced To Do Application 217
20.10 Summary 219

21 Wish list of new chapter **221**
21.1 Willow and more 221
21.2 Bootstrap 221
21.3 MDL 221
21.4 WebSockets 221

Illustrations

2-1 The Seaside development environment. 12
2-2 The Seaside server running. 12
2-3 Stop the server and start a new one. 13
2-4 Pharo class template for the `Web Counter` 13
2-5 Creating the class `WebCounter`. 14
2-6 The class has been created. 14
2-7 WebCounter initialize method. 15
2-8 Compiling a method. 15
2-9 The method has been compiled. 16
2-10 Increase method. 16
2-11 Decrease method. 16
2-12 Example of `renderContentOn:` method 16
2-13 A simple counter. 17
2-14 Register a component as an application from a workspace. 18
2-15 Automitically register your application with an initialize method. 18
2-16 Adding the executable comment. 19
2-18 Add anchors and callbacks to your counter 19
2-17 A simple counter with actions. 20
2-19 A simple counter with a different value. 20
2-20 A class comment. 21

3-1 GLASS Virtual Appliance status page. 24
3-2 GLASS Virtual Appliance status. 25
3-3 GLASS Virtual Appliance Seaside page. 26
3-4 GLASS Virtual Appliance menu. 27

5-1 Hello World component. 37
5-2 ScrapBook with vertical elements. 38
5-3 ScrapBook with horizontal elements. 39
5-4 Rendering object with the `render:` method. 40
5-5 Rendering object with the `render:` method. 41
5-6 Select brush, configure it, and render it. 42
5-7 Select brush, configure it, and render it. 43
5-8 A list of items. 47
5-9 A table of items. 49

6-1 Essential CSS structural elements. 52

7-1 A simple anchor. 58
7-2 Using a callback. 59
7-3 Displaying the contact database. 62
7-4 Contact list with Add contact link. 63
7-5 Contacts can now be removed. 63
7-6 Contacts can now be removed. 64
7-7 With mailto anchor. 65

8-1 Filling up our contact view. 69
8-2 Gender as a drop-down menu. 72
8-3 With radio buttons. 73
8-4 Filling up our contact view. 74
8-5 With Check boxes. 76
8-6 Full contact view. 78
8-7 File uploads. 79

9-1 New version of the `ContactListView`. 86
9-2 Call and Answer at Work. 87
9-3 Contact edition with a cancel button. 89

10-1 Embedding a ContactView into another component. 96
10-2 Embedding a ContactView into another component with Halos. 98
10-3 With components all the way down. 100
10-4 A readonly view of the ContactView. 104
10-5 Adding a message around a component. 105
10-6 Decorating a component with a window. 106
10-7 Using a decoration to add buttons and form to a ContactView. 107

11-1 Guessing Game interaction. 112
11-2 A simple reservation based on task. 114
11-3 A simple reservation with feedback. 116

13-1 A simple model with items and an item container. 121
13-2 The application is registered in Seaside. 124
13-3 Our application is there, but nothing is rendered. 124
13-4 Our todo application simply displaying its title. 125
13-5 Our todo application, displaying its title and a list of its items colored according to status. 127
13-6 Our todo application with anchors. 128
13-7 Our todo application with add functionality. 128
13-8 Our todo application with checkboxes and save buttons. 129
13-9 Getting an editor to edit new item. 132
13-10 With an item added. 133
13-11 Our final todo App 134

14-1 Just started playing. 135
14-2 The column labels look like this. 138
14-3 Row labels are letters, column labels are numbers. 139
14-4 The Sudoku board with the style sheet applied. 140
14-5 The Sudoku grid is showing the possible values for each cell. 141
14-6 A partially filled Sudoku grid. 143
14-7 A solved Sudoku grid. 143
14-8 Error when trying to replace 6 by 7. 144

15-1 Including an external picture into your components. 148
15-2 Including an external picture into your components. 149
15-3 Displaying Pharo graphical object.. 150
15-4 Application with enabled style sheet. 151
15-5 Configuring file libraries through the web interface: clicking on files - configure. 152
15-6 File libraries. 153
15-7 File libraries. 153
15-8 An empty CounterLibrary. 155
15-9 Adding files to the CounterLibrary. 155
15-10 Counter with the updateRoot: method defined. 157
15-11 The Pharo String Library. 159

16-1 The session of miniInn is now InnSession. 165
16-2 With Session. 167
16-3 With Session: Enter your name. 168
16-4 With Session: Starting Date and Ending Date. 169
16-5 Life cycle of a session. 170

17-1 The ToDo Feed subscribed. 174

18-1 First architecture: adding REST to an existing application. 178
18-2 Second architecture: REST centric core. 178
18-3 Request handling of `WARestfulFilter` and `WAApplication`. 186
18-4 Inspecting a request. 187

19-1 Configure an application for deployment. 191
19-2 Configuration options for absolute URLs. 192
19-3 Apache Load Balancer Manager. 201
19-4 System Status Tool. 203

20-1 jQuery Demo and Functional Test Suite. 208
20-2 jQuery Lifecycle. 209

CHAPTER 1

Introduction

Seaside is an excellent framework for easily developing advanced and dynamic web applications. Seaside lets you create reusable components that you can freely compose using Pharo — a simple and pure object-oriented language.

Seaside offers a powerful callback mechanism that lets you trigger code snippets when the users clicks on a link. With Seaside, you can debug your web application with a powerful dynamic debugger and modify the code on the fly while your server is running. This makes the development of complex dynamic applications smooth and fast.

With Seaside, you have the time to focus on your design and solutions to your problems. In this chapter, we give an overview of Seaside and present some Smalltalk basics to help you to follow along with the book. In the next chapter, we will show you how you can program your first Seaside component in just 15 minutes.

1.1 What is Seaside?

Seaside is a free, open-source framework to build powerful and dynamic web applications. The developer can use and extend these classes to produce highly dynamic web-based *applications*. By applications, we mean real applications with potentially complex workflows and user interactions, in contrast to just collections of static pages. Seaside makes web development simpler and can help you build applications that are cleaner and easier to maintain because it has:

- a solid component model and callbacks,
- support for sequencing interactions,

- powerful Pharo debugging interface, and
- support for using AJAX and other Web 2.0 technologies such as REST and websockets.

Seaside applications are based on the composition of independent components. Each component is responsible for its rendering, its state, and its own control flow. Seaside enables you to freely compose such components, creating advanced and dynamic applications comparable to widget libraries such as Swing or Morphic. What is really powerful is that the control flow of an application is written in plain Pharo code.

A bit of history

Seaside was originally created by Avi Bryant and Julian Fitzell. It is mature frameworks used by several companies worldwide.

Features

Seaside is often described as a *heretic web framework* because by design it went against what is currently considered best practice for web development – such as using templates or clean, carefully chosen, meaningful URLs. Seaside does not follow REST (Representational State Transfer) by default. Instead, URLs hold session key information, and meaningful URLs have to be generated explicitly, if needed. Seaside also offers a smooth integration and let you expose functionality via REST. So you can get the best of both worlds.

When using a template system (PHP, JSP, ASP, ColdFusion, and so on), the logic is scattered across many files, which makes the application hard to maintain. Reuse, if possible at all, is often based on copying files. The philosophy of the template approach is to separate the responsibilities of designers and programmers. This is a good idea that Seaside also embraces. Seaside encourages the developer to use CSS to describe the visual appearance of a component, but it does not use a templating engine, and encourages developers to programmatically generate meaningful and valid XHTML markup.

Seaside is easy to learn and use. By comparison, JSF (JavaServer Faces) requires you to use and understand several technologies such as Servlets, XML, JSP, navigation configuration in `faces.config` files, and so on. In Seaside, you only need to know Pharo, which is more compact and easier to learn than Java. Furthermore, it is good to know some basics about CSS. Seaside lets you to concentrate on the problem at hand and not on integrating technologies. Seaside ensures that you always generate valid XHTML and that all your code is in Pharo.

Now you can also mix Javascript code within Seaside. You can also use different libraries such as Bootstrap, MDL and take advantage of Websockets and others systems such Mustache.

Summary

In summary, several aspects of Seaside's design differ from most mainstream web application frameworks. In Seaside

- Session state is maintained on the server.
- HTML is generated completely in Smalltalk. There are no templates or "server pages", although it isn't hard to build such things in Seaside.
- You use callbacks for anchors and buttons, rather than loosely coupled page references and request IDs.
- You use plain Smalltalk to define the flow of your application. You do not need a dedicated language or XML configuration files.

Combined, these features make working with Seaside much like writing a desktop GUI application. Actually, in some ways it is simpler, since the web browser takes a lot of the application details out of your hands.

The next section lists some real-world Seaside applications that you can browse to understand the kind of applications you can build with Seaside. Each of these applications allows complex interaction with the user, rather than a simple collection of pages.

1.2 Some Seaside Applications

With Seaside, you will be able to build any kind of web application. You can see some Seaside applications running on the web. You can find more information athttp://www.pharo.org/success. Seaside is used in many intranet web applications, that are often not readily visible without going behind the scenes.

We have selected two Seaside examples from the publicly available projects. Have a look at them to see the kind of interaction and application flow that can be built with Seaside.

Yesplan (http://www.yesplan.behttp://www.yesplan.be/) Yesplan is a collaborative event planning web application. A video on the website shows a nice summary of the application. The user interaction and smooth application flow is really striking and a good illustration of the power of Seaside to build complex applications.

Cmsbox (http://www.cmsbox.chhttp://www.cmsbox.ch/) An AJAX-based content management system designed for usability. Here the navigation is more the kind we expect from a web application.

There are also several open-source projects based on Seaside; we list two interesting ones, since you may use them when going through the book.

Pier (http://www.piercms.comhttp://www.piercms.com/) Pier is a kind of meta content management system into which a Seaside application can be plugged. Pier is the second generation of an industrial strength content application management system. It is based on Magritte, a powerful meta-description framework. Pier enables easy composition and configuration of interactive web sites with new and ready-made Seaside application or components through a

convenient web interface without having to write code. The Seaside website is based on Pier, as is the online version of this book.

SqueakSource (http://www.squeaksource.comhttp://www.squeaksource.com/) SqueakSource is a web-based source management system for Squeak using the Monticello code versioning system and it is more traditional in the kind of flow it presents.

1.3 What is Pharo?

First, the design of Pharo is still one of the best in terms of elegance, purity, and consistency. Second, the set of tools is really good: debuggers, browsers, refactoring engines, and unit testing frameworks are all available in Pharo. Third, and most important, in Pharo you constantly interact with live objects. This is particularly exciting when developing web applications. There's no need to constantly recompile your code or restart the server. Instead, you debug your applications on the fly, recompile running code, and access your business objects right in the browser, which gives you a huge productivity gain.

Experience has proven to us that Pharo is not difficult to learn, it provides many advantages and it's no hindrance to using Seaside. In fact we often see people starting to learn Pharo because of Seaside. To help you get up to speed, we suggest you read *Pharo by Example*. It is a free book available at http://books.pharo.orghttp://books.pharo.org/. Chapters 3, 4 and 5 contain a minimal description of Pharo, its object-oriented model and the elementary syntax elements that you need to know to follow this book. In the next chapter, we will help you to get started with the environment step by step.

In this book, we use the *Seaside 3.1 Image* which you can find on the Seaside website at http://www.seaside.sthttp://www.seaside.st/. The One Click Image is a bundle of everything you need to run Seaside once you unzip it. We suggest you use this image to start. It makes things much simpler.

The Seaside mailing list is a good place to ask questions because the subscribers to the list answer questions quickly. Do not hesitate to join and participate in the community. You can also discuss Seaside development on the Discord server: XXXX

Okay then, you now have tools at your disposal to help you through any problems you might encounter.

1.4 Structure of the Book

Getting Started Explains how to get a Seaside application up and running in less than 15 minutes. It will show you some Seaside tools.

Fundamentals Shows you how to manipulate basic elements, such as text, anchors, and callbacks, as well as forms. It presents the notion of a *brush*, which is central to the Seaside API.

Components Describes components, the basic building blocks of Seaside. It shows how components are defined and can populate the screen or be called and embedded within one another. It also presents tasks that are control flow components and describes how reuse is achieved in Seaside via component decoration. It ends with a discussion of the Slime library, which checks and validates your Seaside code.

Seaside in Action This part develops two little applications – a todo list manager and a sudoku player. Then it presents how to serve files, a discussion of character encodings, and how to customize a session to hold application-centric information.

Web 20 This part describes how to create an RSS feed, as well as the details of integrating JavaScript into an application. It finishes by showing some push technology such as *Comet*, which allows you to synchronize multiple applications.

Advanced Presents some details that you face when you configure and deploy a Seaside application. It shows how to test Seaside components, and discusses Seaside security by presenting the most common attacks and how Seaside deals with them effectively. Then, even though Seaside is not about persistency, we discuss some persistency approaches. Finally, we present Magritte, a meta-data framework, and its Seaside integration. Magritte lets you generate forms on the fly.

1.5 Formatting Conventions

We need to say a word about formatting conventions before we proceed. In Pharo, as in most Smalltalk implementations, you edit code using a code browser as we will show you in the next chapter. To look at the code for a method, you select a package, then a class, a method category and finally the method you want to see. The method's class is always visible. When reading a book, a method's class may not be so obvious.

To help your understanding of the code we present, we will follow a common convention to display Smalltalk code: we will prefix a method signature with its class name. Here is an example. Suppose you need to enter the method`renderContentOn:` in your browser, and this method is in the class:`WebSudoku`. You will see the following code in your browser.

```
renderContentOn: html
    html div
        id: 'board';
        with: [ html form: [ self renderBoardOn: html ] ]
```

To help you remember that this method is defined in the class `WebSudoku`, we will write it as follows:

```
WebSudoku >> renderContentOn: html
    html div
        id: 'board';
        with: [ html form: [ self renderBoardOn: html ] ]
```

When you enter the text for this method, you do not type `WebSudoku>>`. It is there only so you will know the method's class. We will use a similar convention in the running text. To be precise about a method and its class, we will use`WebSudoku>>renderContentOn:`.

In Pharo, a class and an instance of a class both have methods. The class methods are analogous to static methods in Java. Class methods respond to messages sent to the class itself. To make it clear that we are talking about a class method, we will refer to it using `WebSudoku class>>canBeRoot`. For example, here is the definition of the class method `canBeRoot`, defined on the class `WebSudoku`:

```
WebSudoku class >> canBeRoot
    ^ true
```

We use the following annotations for specific notes: This is a side-note and might be interesting to readers more curious about the topic.Advanced This is a remark covering advanced topics. It can be safely skipped on the first pass through the book.Important This is an important note, if you do not follow the suggestions you are likely to get into trouble.

1.6 Acknowledgments

We wish to thank all the people who helped to make this book possible. Torsten Bergmann, Damien Cassou, Tom Krisch, Philippe Marshall, Ruben Schempp, Roger Whitney, and Julian Fitzell carefully reviewed the book and provided valuable feedback. Martin J. Laubach for his

Sudoku code. Ramon Leon for letting us use his ideas described on his blog on SandStoneDB, and Chris Muller for Magma. Jeff Dorst provided financial support for supporting student text reading. Markus Gaelli for brainstorming on the book title. Samuel Morello for designing the cover. We thank the European Smalltalk User Group, Inceptive.be, Cincom Systems, GemStone Systems Inc. and Instantiations for the generous financial support.

Furthermore, an uncountable number of people provided feedback through the notes on the website: aaamos, agarcia, alamkhan733, aldeveron, alejperez, alex.albitov, alleagrastudena, amalagsoftware, amalawi, andre, andrew.evil.genius, andy.burnett, anhlh, anitatiwari66, anonimo, antkrause, anukpriya, apstein, arc, ardaliev, artem.voroztsov, asselinraymond, astares, awol, b.prior, bart.gauquie, basilmir, bendict101, benoit.astruc, bgridley, bilesja, bjorn.eiderback, blank, bonzini, bouraqadi, brauer, briannolan45, bromagosa, bruefkasten, bschwab, bugmenot, cacciaresi, carlg, carlos.crosetti, cdrick65, cems, cesar.smx, chaetal, chicoary, chip, chris.pollard, chrismeyer206, christophe.rettien, chunsj, citizen428, cj-bachinger, colson, craig, crystal.dry.eyes, cuyeu, cy.delaunay, dago1965, damien.cassou, damien.pollet, dan, danc, david, davidleonhardt, dev, didier, dmytrenko.d, dsblakewatson, dvciontu, ed.stow, efinleyscience, elendilo, epovazan, fabio.braga, fgadzinski, flipityskipit, fractallyte, fraggerbe, francois.le.coguiec, francois.stephany, frans, frelach, friends.prince, fritz.schenk, galyathee, garybarnett, gaston.dalloglio, geert.wl.claes, george, ginolee859, goaway1000, haga551010, halcyonshizzle, hannes.hirzel, hentai, hichem_warum_nicht, hjhoffmann, hm, ino.santangelo, intrader, ismailshuaibu, itsme213, jailachure11, jayers, jborden23, jeremy.chan, jesusalbertosanchez, jgarcia, jguell, jkiggundu, jnials, joel, john.chludzinski, john_okeefe, josef.springer, jpamayag, jred_xv, jrinkel, juanmfernandez, junkabyss, juraj.kubelka, justin.forder, karsten, kees, kjborden23, kommentaren, kontakinti_11, kremerk, landriese, laurent.laffont, lehoanganh.vn, lenglish5, lgadallah, liangbing64, linuxghostpower, liquidhorse, lorenzo, luis.ramirez, ma.chris.m, mani7info, manishmore14, marcello.rocha, marcos.macedo, mark.owens999, martin.t.krebs, matthias.berth, mcleod, merlyn, michael, misaeboca, miss.martinezsandra, mitul_b_shah, momode56, momoewang, morbusg, mriffe, muzzahmed01, nathan_benninghoff, nath_kamal, ncalexander, netprobe, nick.ager, nielvv, nikita.pristupchik, niko.saint, niko.schwarz, nizar.jouini, nrf, nwmullen, offray, pat.maddox, paulpham, pdebruic, peterg, petton.nicolas, pjw1, qwe517, r.koller, rafael.luque.leiva, rajat.tags, ramiro, ramon.leon, ramshreyas.rao, razavi, rene.mages, rh, rhawley, richard_a_green, riverdusty, robert, robert.reitenbach, robin.luiten, rogthedodge, ron.fucci, rsiel.455, rwelch, samoila.mircea, samthecool7, sean, seansorrel, seaside.web.macta, sebovick, sergio, sergio.rrd, shar_28_min, sheshadri.mantha, simon, simon.denier, smalltalk, smalltalktelevision, snoobabk, sokhoeun.kong, solveig.instantiations, squeakman, ssmith, stefan.izota, stephan, stephen.smith, steve, stevek, sthomas1, stuart, sukumini_g, szaidi6, t.pierce, tallman, tanga, tariqrauf2002, tatacarrera, tfahey, thewinterlion, thiagosl, timloo0710, tobez, tony, tony.fleig, tpburke, tudor.girba, tyusupov, udo.schneider, unixmonkey1, vagy, vanchau, victorct83, vinref, vmusulainen, vsteiss, watchlala, web.macta, wolfopsys, wrcstewart, wrinkles, write.to.me, wsgibson, xekoukou, xs4hkr, y2ahsan, yanni, yasirkaram, zanveb82, zhangxinchun2008. Thank you all.

We give a special thanks to Avi Bryant and Julian Fitzell for inventing Seaside. In particular, they showed us that going against the current is possible when you have brilliant ideas and a powerful language such as Smalltalk.

s

Part I

Getting Started

This part shows you how to get a simple Seaside application up and running in less than 15 minutes. If you're new to Pharo, have a look at the Pharo by Example Seaside Chapter. You can also have a look at the excellent Pharo Mooc http://mooc.pharo.org. Watch the first video and the development session about the counter.

CHAPTER 2

Getting Started in Pharo

In this chapter we will explain how to develop a first simple Seaside application: a simple counter. You will follow the entire procedure of creating a Seaside application. This process will highlight some of the features of Seaside such as callbacks and a DSL to emit HTML.

We will neither explain the Pharo syntax nor the IDE. We will not explain how to define packages, classes or methods but obviously we will present their definition. If you are new to Pharo, we suggest you to read chapters 3, 4 and 5 of *Pharo by Example* which is a free and online book available from http://books.pharo.org and the Pharo mooc available at http://mooc.pharo.org.

2.1 Getting Seaside

In this book, we use Seaside 3.0.4, included in the *One Click Image* which you can find on the Seaside website at http://www.seaside.st/download. The*One Click Image* is a bundle of everything you need to run Seaside.

You should see the Seaside development environment open in a single window on your desktop similar to the one presented in Figure **??**. revise

2.2 Starting the Server

Revise The *One Click Image* image includes a web server called "Comanche" listening on TCP port 8080. You should check that this server is properly running by pointing your web browser to http://localhost:8080/. You should see something like Figure 2-2. If you don't see this page, it is possible that port 8080 is already in use by another application on your computer.

Changing the Seaside port number. If you did not see Figure 2-2, you will need to try modifying the workspace to restart the Comanche web server on a different port number (like 8081). The script 2-3 asks the server to stop serving and start serving on port 8081.

To execute this, you would open a new workspace using *World | Workspace*, enter the text, select it, right-click for the context menu, and select*Do it*.

Once you have done this, you can try to view it in your browser making sure you use the new port number in your URL. Once you have found an available port, make sure you note what

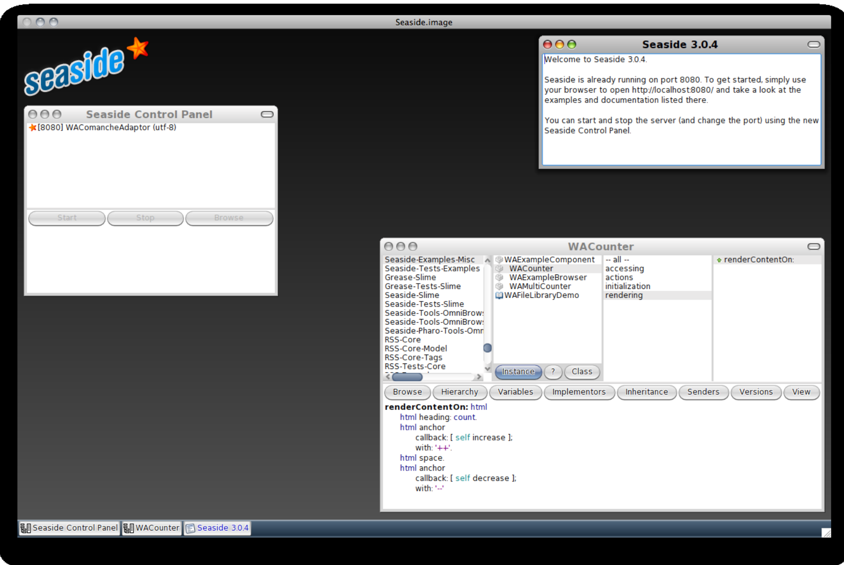


Figure 2-1 The Seaside development environment.

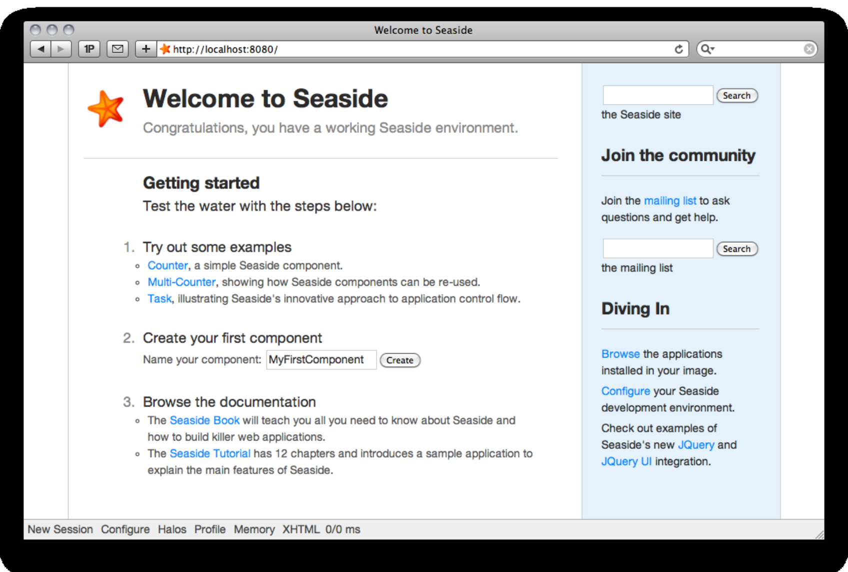


Figure 2-2 The Seaside server running.

Listing 2-3 Stop the server and start a new one.

```
WAKom stop.
WAKom startOn: 8081.
```

Listing 2-4 Pharo class template for the `Web Counter`

```
WAComponent subclass: #WebCounter
    instanceVariableNames: 'count'
    classVariableNames: ''
    package: 'WebCounter'
```

port the server is running on. Throughout this book we assume port 8080 so if you're using a different port you will have to modify any URLs we give you accordingly.

2.3 A First Seaside Component

Now we are ready to write our first Seaside component. We are going to code a simple counter. To do this we will define a component, add some state to that component, and then create a couple of methods that define how the component is rendered in the web browser. We will then register it as a Seaside application.

Defining a Package

To start with, we define a new package that will contain the class that defines our component. We will save our class in this package.

Defining a Component

Now we will define a new component named `WebCounter`. In Seaside, a*component* refers to any class which inherits from the class `WAComponent` (either directly or indirectly). It is only a coincidence that this class has the same name as its package. Normally packages will contain several classes, and the package names and class names are unrelated. To start creating your class, click on the `WebCounter` package you just created. The "class creation template" will appear in the source pane of the browser. Edit this template so that it looks as in the script 2-4

Notice that lines 3 and 4 contain two consecutive single quote characters, not a double quote character. We are specifying that the `WebCounter` class is a new subclass of `WAComponent`. We also specify that this class has one instance variable named count. The other arguments are empty, so we just pass an empty string, indicated by two consecutive quote marks. The "package" value should already match the package name. Note that an orange triangle in the top-right indicates that the code is not compiled yet.

Once you are done entering the class definition, right-click anywhere in that pane to bring up the context menu, and select the menu item *Accept (s)* as shown in Figure **??**. Accept in Pharo jargon means compile.

Once you have accepted, your browser should look similar to the one shown in Figure **??**. The browser now shows the class that you have created in the class pane. Now we are ready to define some behaviour for our component.

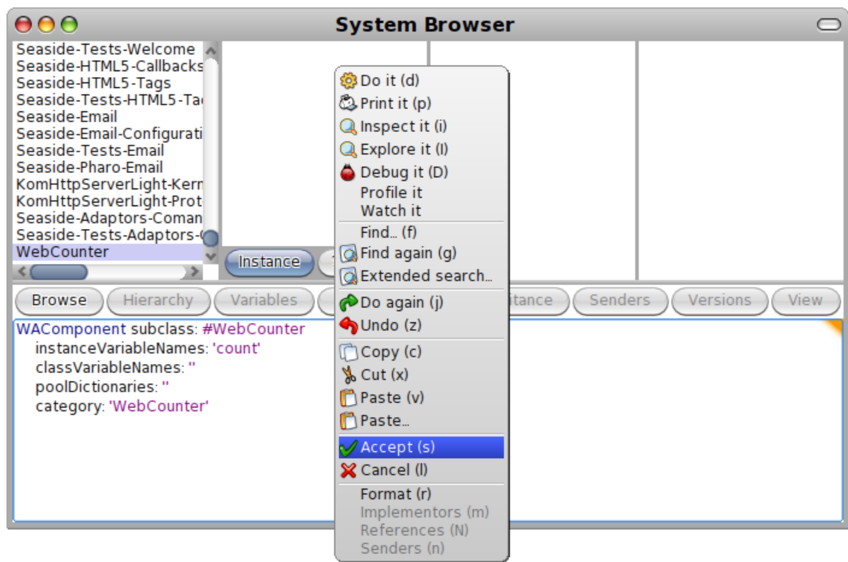


Figure 2-5 Creating the class `WebCounter`.

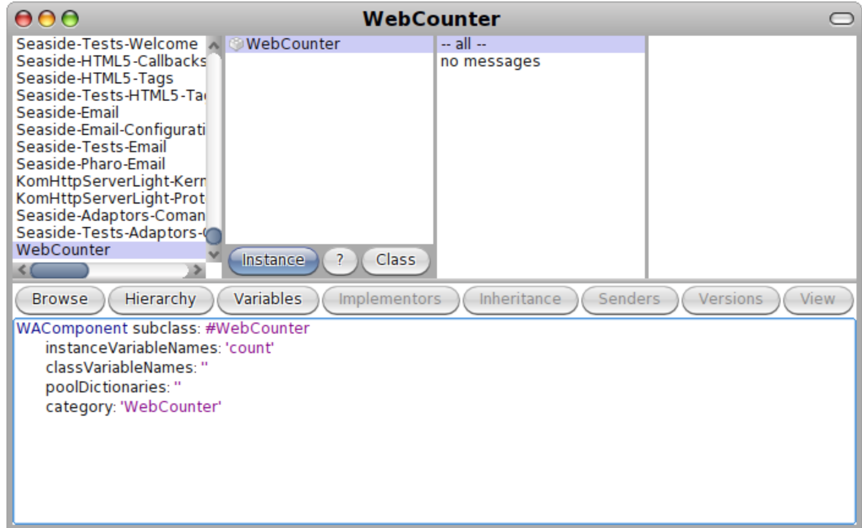


Figure 2-6 The class has been created.

Listing 2-7 WebCounter initialize method.

```
WebCounter >> initialize
    super initialize.
    count := 0
```

2.4 Defining Basic Methods

Now we are ready to define some methods for our component. We will define methods that will be executed on an instance of the `WebCounter` class. We call them instance methods since they are executed in reaction to a message sent to an instance of the component.

The first method that we will define is the `initialize` method, which will be invoked when an instance of our component is created by Seaside. Seaside follows normal Pharo convention, and will create an instance of the component for us by using the message `new`, which will create the new instance and then send the message `initialize` to this new instance.

Remember that this definition states that the method `initialize` is an instance side method since the word `class` does not appear between`WebCounter` and `>>` in the definition.

Once you are done typing the method definition, bring up the context menu for the code pane and select the menu item *accept (s)*, as shown in Figure2-8.

At this point Pharo might ask you to enter your full name. This is for the source code version control system to keep track of the author that wrote this code.

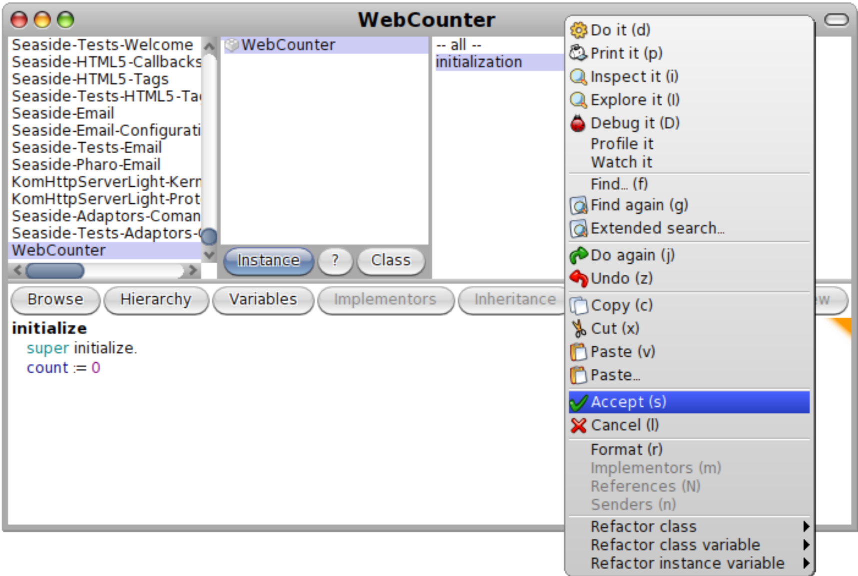


Figure 2-8 Compiling a method.

The method signature will also appear in the method pane as shown in Figure 2-9.

Now let's review what this means. To create a method, we need to define two things, the name of the method and the code to be executed. The first line gives the name of the method we are

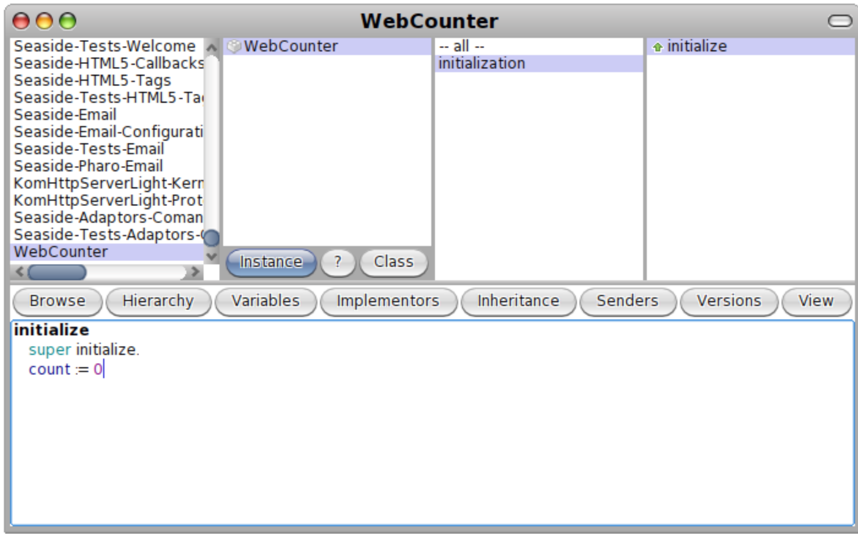


Figure 2-9 The method has been compiled.

Listing 2-10 Increase method.

```
WebCounter >> increase
    count := count + 1
```

Listing 2-11 Decrease method.

```
WebCounter >> decrease
    count := count - 1
```

defining. The next line invokes the superclass`initialize` method. The final line sets the value of the count instance variable to 0.

To be ready to define Seaside-specific behaviour, define two more instance methods to change the counter state as in scripts 2-10 and 2-11. You can group them in a protocol 'action'.

2.5 Rendering a Counter

Now we can focus on Seaside specific methods. We will define the method`renderContentOn:` to display the counter as a heading. When Seaside needs to display a component in the web browser, it calls the `renderContentOn:` method of the component, which allows the component to decide how it should be rendered.

Add a new method category called rendering, and add the method definition in script 2-12

Listing 2-12 Example of `renderContentOn:` method

```
WebCounter>>renderContentOn: html
    html heading: count
```

We want to display the value of the variable count by using an HTML heading tag. In Seaside, rather than having to write the HTML directly, we simply send the message `heading:` to the html object that we were given as an argument.

As we will see later, when we have completed our application, this method will give us output as shown in Figure **??**.

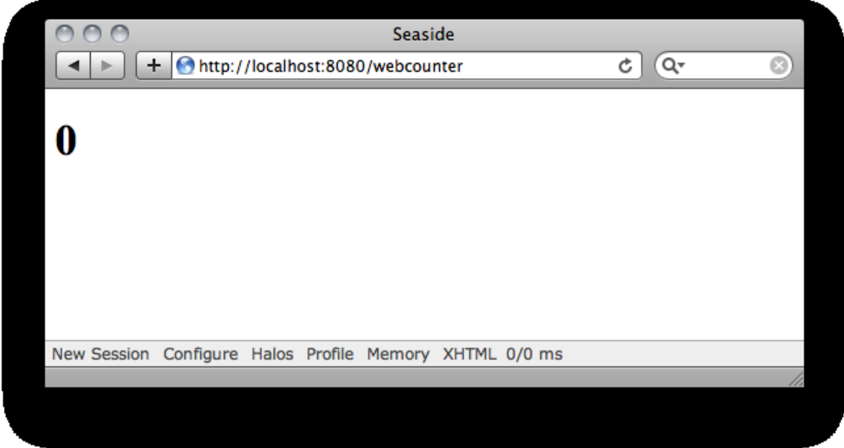


Figure 2-13 A simple counter.

Registering as a Seaside Application

We will now register our component as an application so that we can access it directly from the web browser. To register a component as an application, we need to send the message `register:asApplicationAt:` to `WAAdmin`.

```
==WAAdmin register: WebCounter asApplicationAt: 'webcounter'==
```

This expression registers the component `WebCounter` as the application named `webcounter`. The argument we add to the `register:asApplicationAt:` message specifies the root component and the path that will be used to access the component from the web browser. You can reach the application under the URL http://localhost:8080/webcounter.

Now you can launch the application in your web browser by going tohttp://localhost:8080/webcounter/ and you will see your first Seaside component running. Put the link to section 7.2 in pillar If you're already familiar with HTML, you may want to look at the introduction to halos in Section 7.2 to learn a little more about how to investigate what's happening under the covers.

Automatically Registering a Component

In the future, you may want to automatically register some applications whenever your package is loaded into an image. To do this, you simply need to add the registration expression to the *class* `initialize` method of the component. A *class* `initialize` method is automatically invoked when the class is loaded from a file. The script 2-15 the `initialize` class method definition.

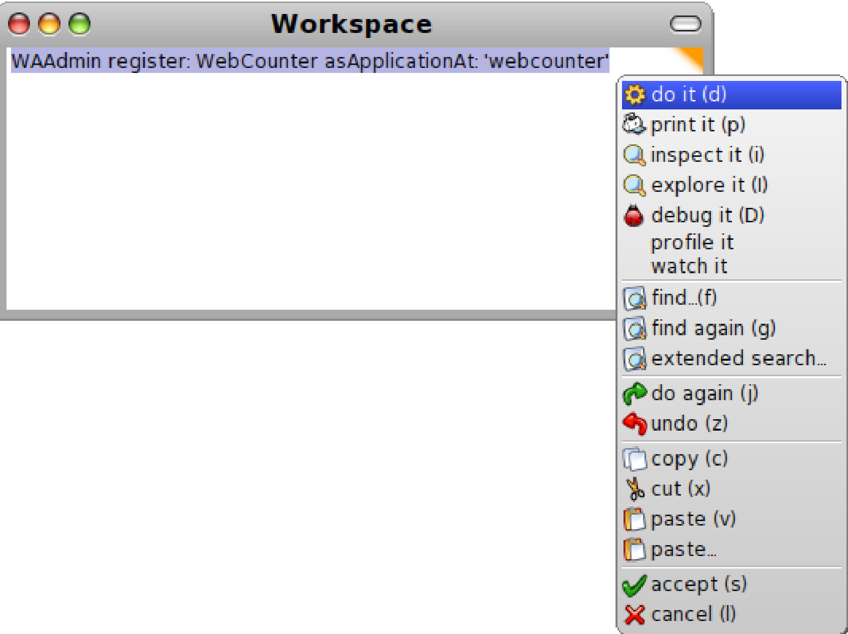


Figure 2-14 Register a component as an application from a workspace.

Listing 2-15 Automitically register your application with an initialize method.

```
WebCounter class >> initialize
    WAAdmin register: self asApplicationAt: 'webcounter'
```

The word "class" in the `WebCounter class>>` first line indicates that this must be added as a class method as described below.

Because this code is in the `WebCounter` class, we can use the term self in place of the explicit reference to *WebCounter* that we used in the previous section. In Smalltalk we avoid hardcoding class names whenever possible.

In the future, we will add configuration parameters to this method, so it is important to be familiar with creating it. Remember that this method is executed automatically only when the class is loaded into memory from some external file/source. So if you had not already executed`WAAdmin register: WebCounter asApplicationAt: 'webcounter'` Seaside would still not know about your application. To execute the `initialize` method manually, execute `WebCounter initialize` in a workspace; your application will be registered and you will be able to access it in your web browser. Important Automating the configuration of your Seaside application via class-side `initialize` methods play an important role in building deployable images because of their role when packages are brought into base images, and is a useful technique to bear in mind for future use. The following Figure 2-16 shows a trick Smalltalkers often use: it adds the expression to be executed as comment in the method. This way you just have to put your cursor after the first double quote, click once to select the expression and execute it using the *Do it (d)* menu item or shortcut.

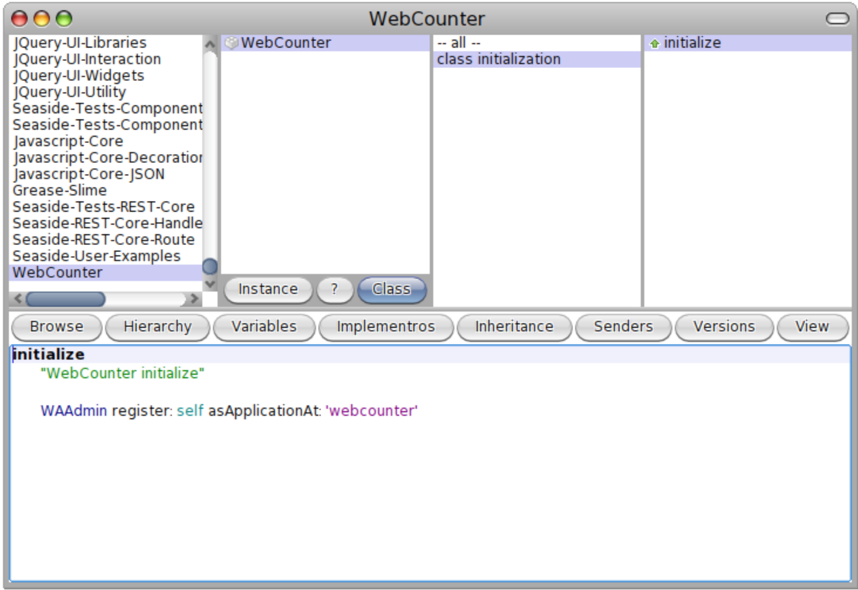


Figure 2-16 Adding the executable comment.

Listing 2-18 Add anchors and callbacks to your counter

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

Adding Behavior

Now we can add some actions to our component. We start with a very simple change; we let the user change the value of the count variable by defining callbacks attached to links (also known as anchors) displayed when the component is rendered in a web browser, as shown in Figure 2-17. Using callbacks allows us to define some code that will be executed when a link is clicked.

We modify the method `WAComponent>>renderContentOn:` as in script2-18. Don't forget that `WAComponent>>renderContentOn:` is on the *instance* side. Each callback is given a Pharo block: an anonymous method (strictly speaking, *alexical closure*) delimited by [and]. Here we send the message`callback:` (to the result of the anchor message) and pass the block as the argument. In other words, we ask Seaside to execute our callback block whenever the user clicks on the anchor.

Click on the links to see that the counter get increased or decreased as shown in Figure 2-19.

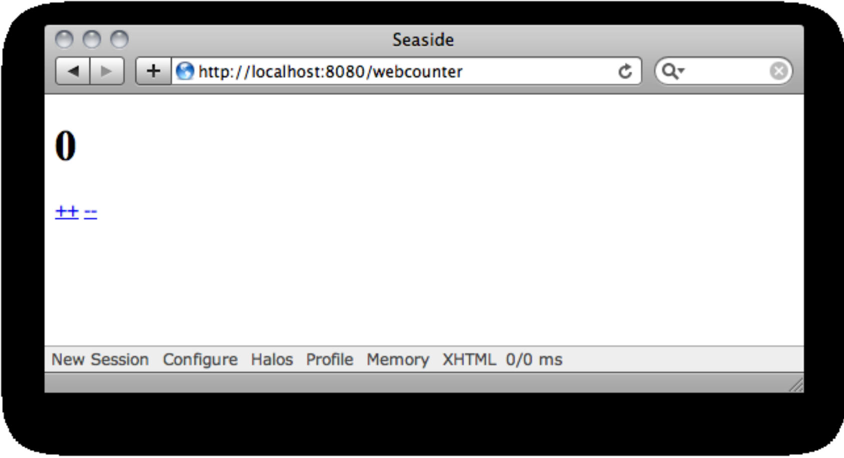


Figure 2-17 A simple counter with actions.

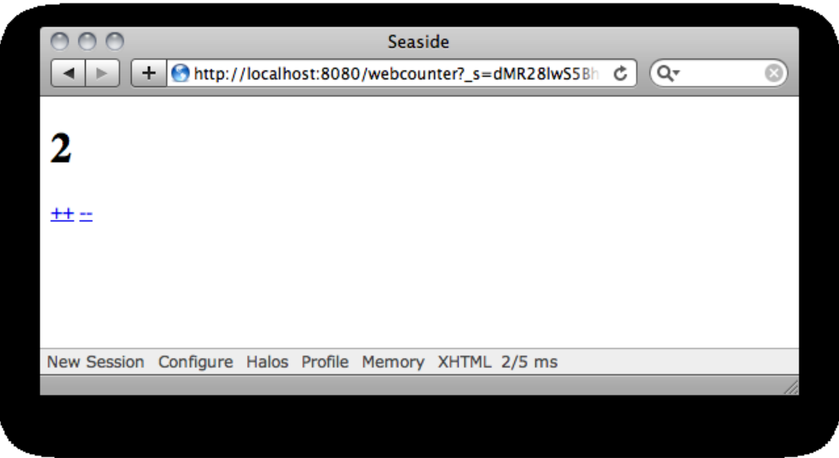


Figure 2-19 A simple counter with a different value.

Adding a Class Comment

A class comment (along with method comments) can help other developers understand a class. Do not forget to add a nice comment to your class and save it using your favorite version control system.

When you're studying a framework, class comments are a pretty good place to start reading. Classes that don't have them require a lot more developer effort to figure out so get in the habit of adding these comments to all of your classes.

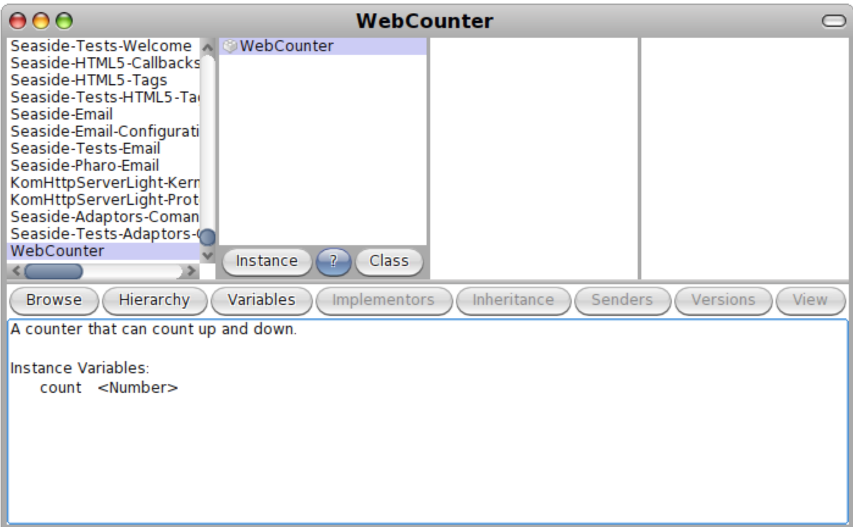


Figure 2-20 A class comment.

Now you are set to code in Seaside.

2.6 Summary

You have now learned how to define a component, register it and modify it. Now you are ready to proceed to Part II to learn all the details of using Seaside to build powerful web applications.

CHAPTER 3

Getting Started in Gemstone/S

In this chapter we show you how to develop a simple Seaside application in GemStone/S. There are two different ways to install and run a GemStone/S server: the GLASS Virtual Appliance (GLASS is an acronym for GemStone, Linux, Apache, Seaside, and Smalltalk) – a pre-built environment for running GemStone/S in VMware, and the GemStone/S Web Edition – a native version of GemStone/S for Linux or Mac OS X Leopard. Further information is available at http://seaside.gemstone.com/.

The identical development process is used for both the GLASS Virtual Appliance and the native GemStone/S Web Edition. Both are available from http://seaside.gemstone.com/downloads.html. For most developers we recommend using the appliance, since this avoids the intricacies of system administration tasks. All GemStone/S editions which run Seaside are fully 64-bit enabled, and require 64-bit hardware, a 64-bit operating system, and at least 1GB RAM. The GLASS Virtual Appliance will run on a 32-bit Windows OS as long as the underlying hardware supports running a VWware 64-bit guest operating system.

3.1 Using the GLASS Virtual Appliance

The GLASS Virtual appliance is a pre-built, ready-to-run, 64-bit VMware virtual appliance configured to start GemStone, Seaside, Apache, and Firefox when it is booted. It is a complete Seaside development environment, including:

- Seaside 2.8 running in a GemStone/S 2.2.5 64-bit multi-user Smalltalk virtual machine, application server and object database.
- A Squeak 3.9 VM and Squeak image configured as a development environment for the GemStone/S server running on the appliance.
- An Apache 2 web server configured to display Seaside applications running in GemStone/S.
- A Firefox web browser set to display the GemStone/S system status on its home page (although you can reach that same page from any browser on your network.)
- A toolbar menu to start, stop, or restart GLASS or Apache, start Squeak, and run GemStone/S backups.
- A toolbar icon which starts a terminal session on the appliance.

- The latest stable release of Xubuntu Linux – Version 7.10.

You start the GLASS Virtual Appliance from your VMware console, just as you would any other VMware virtual appliance. The first time it may take several minutes before the system is fully operational since it must boot Linux, start the GemStone/S server, three GemStone virtual machines, Apache, and Firefox. It's ready once you see the status page shown in **??**.

Figure 3-1 GLASS Virtual Appliance status page.

We recommend when you are ready to stop work, you suspend the appliance rather than shut it down. This will make the next startup much faster. You'll be able to start up just where you left off.

The status of your GemStone/S system is refreshed every 10 seconds. All the GemStone processes listed in the right sidebar should have a green OK status as shown in 3-2. If not, use the "GLASS Appliance" menu shown in 3-4 to start, stop, or restart GLASS or its individual components.

You should now be able to explore the Seaside components installed in the GLASS Virtual appliance by clicking on the "GLASS: Seaside" bookmark you can see in 3-3. You can also view that web page from another computer on your network by using the "eth0:" IP address listed under "Network Information".

Should you need to edit a file or perform other command line operations on the appliance, you can open a terminal session by clicking on the terminal icon in the toolbar. If you prefer, you can `ssh` to the appliance by using the IP address mentioned above and the username/password

GLASS Appliance Status

GemStone Processes

Stone: OK
cache: OK
Netldi: OK
Gem 9001: OK
Gem 9002: OK
Gem 9003: OK
Gem Maint: OK

Network Information

lo: 127.0.0.1
eth0: 192.168.77.128

System Information

Kernel: 2.6.22-14-generic
CPU: 2162.615 MHz
RAM: 500 MB

Figure 3-2 GLASS Virtual Appliance status.

Figure 3-3 GLASS Virtual Appliance Seaside page.

glass/glass. To copy files to/from the appliance use the scp command. Here's an example of using scp to copy a seaside log file from the appliance to your current directory.

```
scp glass@192.168.77.128:/opt/gemstone/log/seaside.log .
```

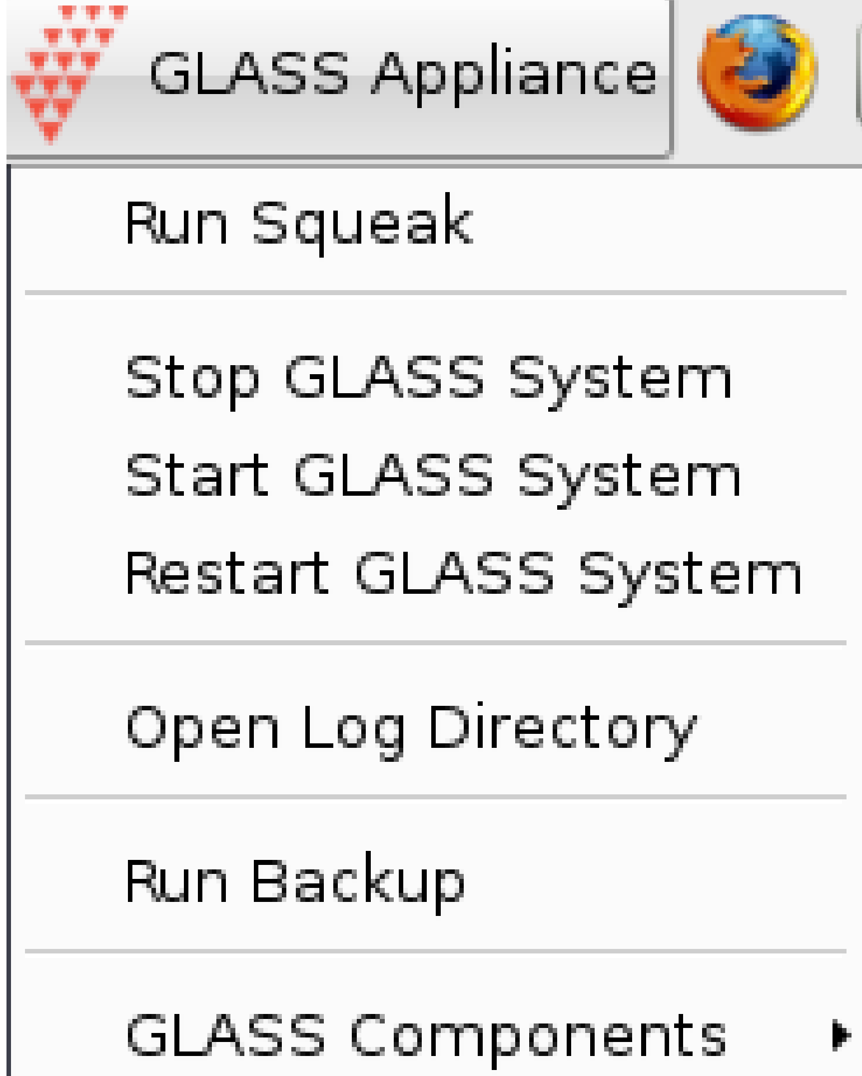


Figure 3-4 GLASS Virtual Appliance menu.

3.2 A first Seaside Component

3.3 Defining a Component

3.4 Defining Some Methods

3.5 Rendering a Counter

3.6 Registering the application

3.7 Adding Behavior

3.8 Keeping Up with the Latest Features

CHAPTER 4

Gemstone version

Part II

Fundamentals

In this part we will introduce you to the manipulation of basic elements such as texts, anchors and callbacks as well as forms. It presents the notion of *brushes* that is central to the Seaside API. Understanding these concepts will be fundamental to your use of Seaside.

CHAPTER 5

Rendering Components

In this chapter, you will learn the basics of displaying text and other information such as tables and lists with Seaside and its powerful XHTML manipulation interface. You will learn how to create a *component* which could include text, a form, a picture, or anything else that you would like to display in a web browser. Seaside's component framework is one of its most powerful features and writing an application in Seaside amounts to creating and manipulating components. You will learn how to use Seaside's API, which is based on the concept of "brushes", to generate valid XHTML.

5.1 Brush concept

One of the great features of Seaside is that you do not have to worry about manipulating HTML yourself and creating valid XHTML. Seaside produces valid XHTML: it automatically generates HTML markup using valid tags, it ensures that the tags are nested correctly and it closes the tags for you.

Let's look at a simple example: to force a line-break in HTML (for instance, to separate the lines of a postal address) you need to use a break tag: `<br/>`. Some people use `<br>` or `<br></br>`, and neither is valid in XHTML. Some browsers will accept these incorrect forms without a problem, but some will mark them as errors. If your content is getting passed on through RSS or Atom clients, it may fail in unexpected ways. You do not need to worry about any of this when using Seaside.

The basic metaphor used in Seaside for rendering HTML is one of painting on a *canvas* using *brushes*. Methods such as `renderContentOn:` that are called to render content are passed an argument (by convention named `html`) that refers to the current canvas object. To render content on this object you can call its methods (or to use the correct Smalltalk terminology, you can pass it messages). In the simple example given above, to add a line-break to a document you would use `html break`.

When you send a message to the canvas, you're actually asking it to start using a new *brush*. Each brush is associated with a specific type of HTML tag, and can be passed arguments defining more detail of what you want to be rendered. So to write out a paragraph of text, you would use `html paragraph with: 'This is the text'`. This tells the canvas to start using the paragraph brush (which causes `<p>` to be output), then output the text passed as the argument, and finally to finish using the brush (which causes `</p>` to be output).

Many brushes can be passed multiple messages before they are finished, by chaining the messages together with `;` (this is called *cascading* messages in Pharo). For example, a generic *heading* exists which can be used to generate HTML headings at various levels, by passing it a `level:` message with an argument specifying the level of heading required:

```
html heading
   level: 3;
   with: 'A third level heading'.
```

This will produce the HTML:

=<h3>A third level heading</h3>

You can cascade as many messages as you need to each brush object.

You can easily tell Seaside to nest tags by using Pharo blocks:

```
html paragraph: [
    html text: 'The next word is '.
    html strong: 'bold' ].
```

This will produce the HTML:

```
<p>The next word is <strong>bold</strong></p>
```

Note that we've used a very handy shortcut here: many of the brush methods have an equivalent method that can be called with a single argument so instead of typing `html paragraph with: 'text'` you need only type `html paragraph: 'text'`.

This is a very brief introduction that will allow you to begin to experiment with how these techniques can be combined into a larger piece of content, as you will see in the following sections.

5.2 First component: Hello World

Our first Seaside component will simply display *Hello world*. Begin by creating a package called 'SeasideBook-Hello' and then create the class `ScrapBook` as a subclass of `WAComponent` as shown below.

```
WAComponent subclass: #ScrapBook
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'SeasideBook-Hello'
```

When we use the term *component* in this text we generally mean an instance of a subclass of `WAComponent`. For now, just think of subclasses of `WAComponent` as "visual components". When it is time for a component to be displayed, Seaside sends it the message `WAComponent>>renderContentOn:` with a single argument (by convention called `html`) which is an instance of the class `WARenderCanvas` (the "canvas"). Think of the canvas as the medium on which you will paint your component. It provides a transparent interface to XHTML which makes it easy to produce text, anchors, images etc., in a modular way (i.e., attached to each component of your application). To start, we just want to show a simple text message. Fortunately the canvas supports a `text:` message for just this purpose, which we can use as shown below. Important Note that all the classes in Seaside are prefixed with `WA` which acts as a namespace. Do not use this prefix for your components. `WA` is intended for Seaside framework classes.

```
ScrapBook >> renderContentOn: html
   html text: 'Hello world'
```

Great, we have a component but how do we get Seaside to serve it? For now, evaluate the following code in a workspace:

```
WAAdmin register: ScrapBook asApplicationAt: 'hello'
```

Now open your web browser and go to http://localhost:8080/hello, and you should see something very like 5-1.

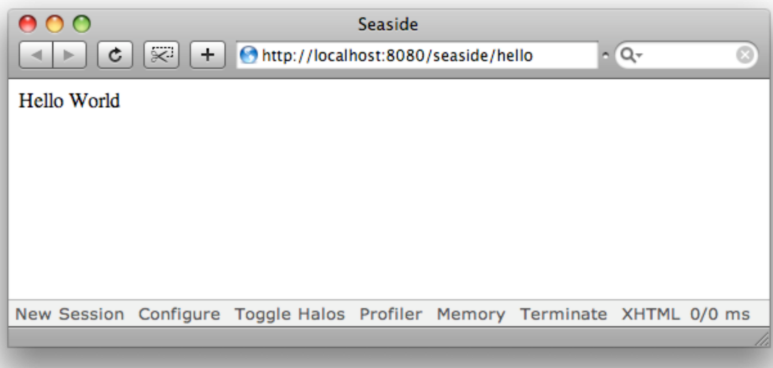


Figure 5-1 Hello World component.

Seaside added XHTML markup for the skeletal structure of an XHTML document (`html`, `head` and `body` tags). OK, so what is happening here? Grossly simplified: When we request this URI, Seaside creates a new instance of our class for us and then sends it `WAComponent>>renderContentOn:`. After being placed inside a skeleton XHTML document, the XHTML painted onto the canvas is then returned to the web browser to be displayed. Important Never invoke the method `renderContentOn:` directly, Seaside will do it for you. You will never need to send your component the message `WAComponent>>renderContentOn:` since the Seaside framework takes care of that for you. When it is time to paint your component, Seaside sends it `renderContentOn:`. This is very similar to models used in most GUI frameworks where a component (or window) is told to paint itself whenever the windowing system deems necessary. Also, keep this in mind as you work with Seaside: a rendering method is just for displaying a component not changing its state. Important Your rendering method is just for painting the current state of your component, it shouldn't be concerned with changing that state.

5.3 Fun with Seaside XHTML Canvas

Let's try making our `ScrapBook` component look a little more exciting. Redefine the method `renderContentOn:` as follows. Refresh your browser and you should see a situation similar to Figure *@ref:5-3).

```
ScrapBook >> renderContentOn: html
   html paragraph: 'Fun with Smalltalk and Seaside.'.
   html paragraph: [
        10 timesRepeat: [
            html image
```

```
                url: 'http://www.seaside.st/styles/logo-plain.png';
                width: 50.
            html horizontalRule ] ]
```

```
ScrapBook>>renderContentOn: html
    html paragraph: 'Fun with Smalltalk and Seaside.'.
    html paragraph: [
        10 timesRepeat: [
            html image
                url: 'http://www.seaside.st/styles/logo-plain.png';
                width: 50 ].
        html horizontalRule ]
```

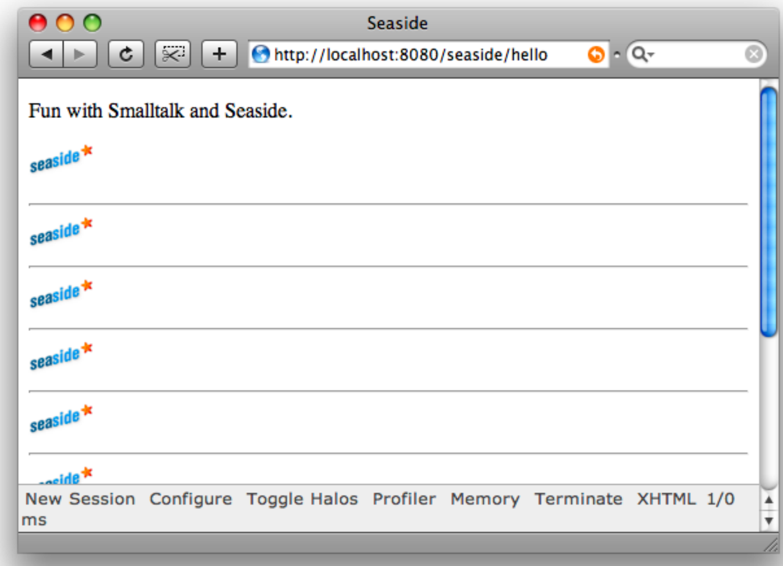


Figure 5-2 ScrapBook with vertical elements.

Using Seaside's canvas and brushes eliminates many of the errors that occur when manually manipulating tags.

5.4 Rendering Objects

Let's take a moment to step back and review what we have learnt. Consider the following method:

```
ScrapBook >> renderContentOn: html
    html paragraph: 'A plain text paragraph.'.
    html paragraph: [
```

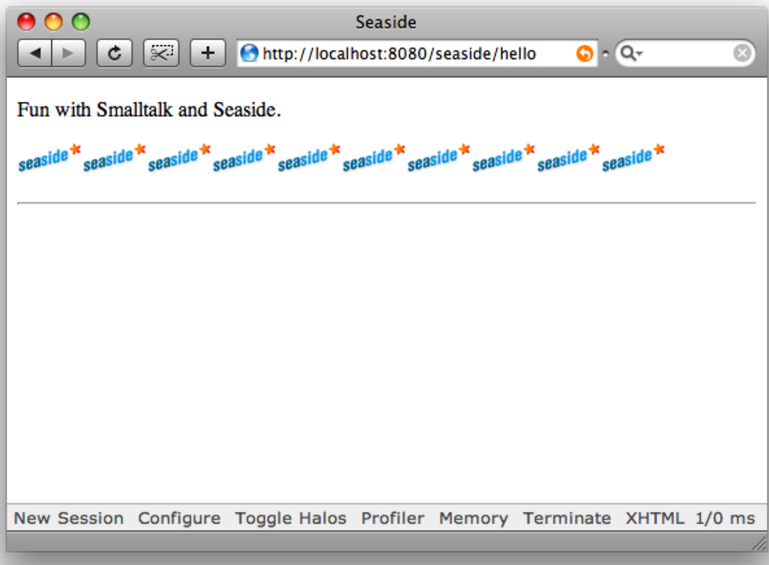


Figure 5-3 ScrapBook with horizontal elements.

```
    html render: 'A paragraph with plain text followed by a line
break. '.
    html break.
    html emphasis: 'Emphasized text '.
    html render: 'followed by a horizontal rule.'.
    html horizontalRule.
    html render: 'An image URI: '.
    html image url:
'http://www.seaside.st/styles/logo-plain.png' ]
```

There are four patterns that appear in this method.

1. `render: renderableObject`. This message adds the renderable object to the content of the component. It doesn't use any brushes, it just tells the object to render itself.
2. Message with zero or one argument. These create brushes. Just as a painter is able to use different tools to paint on a canvases, brushes represent the various tags we can use in XHTML, so `horizontalRule` will produce the tag `hr`. Some brushes take an argument such as `emphasis:` other don't. Section **??** will cover this in depth.
3. Composed messages. The expression `html image` creates an image brush, and then sends it a `url:` message to configure its attributes.
4. Block passed as arguments. Using a block (code delimited by `[` and `]`) allows us to say that the actions in the block are happening in the context of a given tag. In our example, the second paragraph takes an argument. It means that all the elements created within the block will be contained within the paragraph.

About the render: message. As you saw, we use the message `render:` to render objects. Modify the method `renderContentOn:` of your ScrapBook component as follows.

```
ScrapBook >> renderContentOn: html
    html paragraph: [
        html render: 'today: '.
        html render: Date today ]
```

Refresh your web browser, you should see a situation similar to Figure 5-4. The method`renderContentOn:` renders a string and the object representing the current date. It uses the method `render:`. Most of the time you will use the method `render:` to render objects or other components, see Chapter .

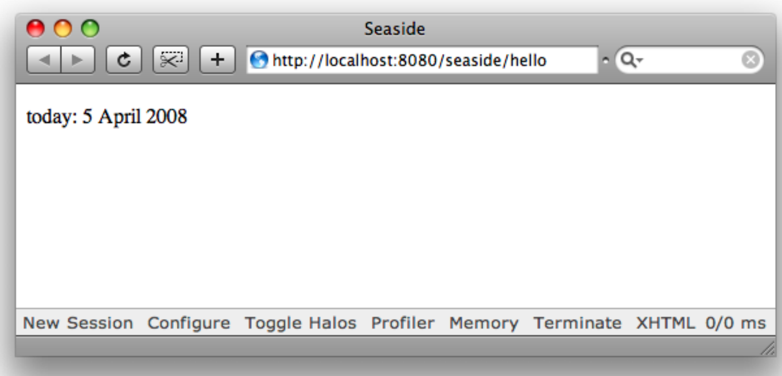


Figure 5-4 Rendering object with the `render:` method.

We use a block as the argument of the `paragraph:` because we want to specify that the string `'today '` and the textual representation of the current date are both within a paragraph. Seaside provides a shortcut for doing this. If you are sending only the message `render:` inside a block, just use the renderable object as a parameter instead of the block. The following two methods are equivalent and we suggest you to use the second form, see Figure 5-5.

Two equivalent methods.

```
ScrapBook >> renderContentOn: html
    html paragraph: [ html render: 'today: ' ]
```

```
ScrapBook >> renderContentOn: html
    html paragraph: 'today: '
```

About the method text:. You may see some Seaside code using the message `WAHtmlCanvas>>text:` as follow.

```
renderContentOn: html
    html text: 'some string'.
    html text: Date today.
```

The method `text:` produces the string representation of an object, as would be done in an inspector. You can pass any object and by default its textual representation will be rendered. In

html brush: [html render: anObject]

is equivalent to

html brush: anObject

Figure 5-5 Rendering object with the `render:` method.

Pharo and GemStone the method `text:` will call the method `Object>>asString`. In any case, the string representation is generated by sending the message `Object>>printOn:` to that object. `html text: anObject` is an efficient short hand for the expression `html render: anObject asString` in Pharo.

About the method renderOn:. If you browse the implementation of `WAHtmlCanvas>>render:` you will see that `render:` calls `renderOn:`. Do not conclude that you can send `renderOn:` in your `renderContentOn:` method. `Object>>renderOn:` is an internal method which is part of a double dispatch between the canvas and an object it is rendering. Do not invoke it directly!

5.5 Brush Structure

In the previous section you played with several brushes and painted a canvas with them. Now we will explain in detail how brushes work. A canvas provides a simple pattern for creating and using these brushes as shown in **??**.

1. Tell the canvas what type of brush you are using.
2. Configure the brush, specifying any special options that it may use.
3. Render the contents of this brush. This is often done by passing an object such as a string or a block to `with:`.

It is not always necessary to send a brush the `with:` message. Do so only if you want to specify the contents of the body of the XHTML tag. Since this message causes the XHTML tag to be rendered, it should be sent *last*.

Here is an example:

```
html heading level: 1; with: 'Hello world'.
```

produces the following XHTML

```
<h1>Hello world</h1>
```

In this example

1. We first specify the brush (tag) we are using with `html heading`.
2. We then send that brush the `level: 1` message to indicate that this should be a level 1 heading.
3. We tell the brush the contents of the heading and cause it to be rendered using `with:`.

```
renderContentOn: html
    ...
    html brush
        brushAttribute1: brushValue1;
        brushAttribute2: brushValue2;
        with: anObject
    ...
```

Figure 5-6 Select brush, configure it, and render it.

Here are some examples that show that it is not necessary to use `with:` if you do not specify the attributes of the brush.

Just a brush

```
html paragraph
```

produces

```
<p></p>
```

A brush with implicit content

```
html paragraph: 'foo'
```

produces

```
<p>foo</p>
```

Setting the content explicitly

```
html paragraph with: 'foo'
```

produces

```
<p>foo</p>
```

Setting attributes directly

```
html paragraph class: 'cool'; with: 'foo'
```

produces

```
<p class="cool">foo</p>
```

If no configuration of the brush is necessary, it is usually possible to specify it with a keyword parameter which becomes the contents of the tag. Thus the two following expressions are equivalent

```
html heading level: 1; with: 'Hello world'.
html heading: 'Hello World'.
```

since level 1 is the default level for a heading.

As you can see, there are two cases where you can write more compact code. These are summarized in 5-7. There is a style checking tool called Slime that checks for such cases. Slime is explained in Chapter .

```
html brush                    html brush
  with: [ ]                     with: [ html render: anObject ]

   is equivalent to                      is equivalent to

html brush                    html brush: anObject
```

Figure 5-7 Select brush, configure it, and render it.

The next section will show you what the other brushes are and how to find information about them within Seaside.

5.6 Learning Canvas and Brush APIs

As we proceed and introduce new brushes, we will provide a table describing the parts of the Brush API that are essential for this book. Once you've used the table to find the brush to use for a given tag, you can read the source code for that brush class to find out how to use it. For example, if you want to produce a heading such as `<h1>Something great</h1>` you'll see that you should use the `WAHtmlCanvas>>heading` message which produces a brush instance of `WAHeadingTag` which has the following API:

Methods in **`WAHeadingTag`**	Description
`level:`	Specify `anInteger` as the heading level for this brush.
`with:`	Render this heading tag with `anObject` as its body.

To help you to find the correct brush, the brushes are presented from the perspective of the HTML tags in the following table:

=HTML Factory Selector Brush Class ==HTML Factory Selector Brush Class ==HTML Factory Selector Brush Class =~~~=HTML Factory Selector Brush Class =~~~~~=HTML Factory Selector Brush Class =~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~~~~~~~~~~~=HTML Factory Selector Brush Class =~~~~~~~~~~~~~~~~~~~~~~~~~~~
=a anchor WAAnchorTag =a map WAImageMapTag =a popupAnchor WAPopupAnchorTag =abbr abbreviated WAGenericTag =acronym acronym WAGenericTag =address address WAGenericTag =big big WAGenericTag =blockquote blockquote WAGenericTag =br break WABreakTag =button button WAButtonTag =caption tableCaption WAGenericTag =cite citation WAGenericTag =code code WAGenericTag =col tableColumn WATableColumnTag =colgroup tableColumnGroup WATableColumnGroupTag =dd definitionData WAGenericTag =del deleted WAEditTag =dfn definition WAGenericTag =div div WADivTag =dl definitionList WAGenericTag =dt definitionTerm WAGenericTag =em emphasis WAGenericTag =fieldset fieldSet WAFieldSetTag =form form WAGenericTag =h1 heading WAHeadingTag =hr horizontalRule WAHorizontalRuleTag

=iframe iframe WAIframeTag =img image WAImageTag =input cancelButton WACancelButton-Tag =input checkbox WACheckboxTag =input fileUpload WAFileUploadTag =input hiddenInput WAHiddenInputTag =input imageButton WAImageButtonTag =input passwordInput WAPass-wordInputTag =input radioButton WARadioButtonTag =input submitButton WASubmitButton-Tag =input textInput WATextInputTag =ins inserted WAEditTag =kbd keyboard WAGenericTag =label label WALabelTag =legend legend WAGenericTag =li listItem WAGenericTag =object object WAObjectTag =ol orderedList WAOrderedListTag =optgroup optionGroup WAOptionGroupTag =option option WAOptionTag =p paragraph WAGenericTag =param parameter WAParameterTag =pre preformatted WAGenericTag =q quote WAGenericTag =rb rubyBase WAGenericTag =rbc rubyBaseContainer WAGenericTag =rp rubyParentheses WAGenericTag =rt rubyText WARuby-TextTag =rtc rubyTextContainer WAGenericTag =ruby ruby WAGenericTag =samp sample WA-GenericTag =script script WAScriptTag =select multiSelect WAMultiSelectTag =select select WAS-electTag =small small WAGenericTag =span span WAGenericTag =strong strong WAGenericTag =sub subscript WAGenericTag =sup superscript WAGenericTag =table table WATableTag =tag: tag: WAGenericTag =tbody tableBody WAGenericTag =td tableData WATableDataTag =textarea textArea WATextAreaTag =tfoot tableFoot WAGenericTag =th tableHeading WATableHeadingTag =thead tableHead WAGenericTag =tr tableRow WAGenericTag =tt teletype WAGenericTag =ul unorderedList WAUnorderedListTag =var variable WAGenericTag Pharo typically encourages explicit naming and avoids abbreviations – the few seconds per day you save by typing an abbreviated method or variable name may often come back much later to haunt you or someone else reading your code as minutes or even hours spent trying to debug code with poor readability. This book is not a complete Seaside reference. Once you're done reading it you will want to discover new brushes and brush options yourself. Let's take a few moments to describe how you would do that.

Suppose you know a specific XHTML tag you want to use and need to find the appropriate brush method. Some brush method names are the same as the corresponding XHTML tag name. For example you create a `div` tag using the `WAHtmlCanvas>>div` brush method. In other cases the brush name is the long form of the equivalent XHTML tag (`WAHtmlCanvas>>paragraph` creates a `p` tag, `WAHtmlCanvas>>unorderedList` creates a `ul` tag etc). This choice makes your methods a lot more readable than if the XHTML tags were used everywhere. Compare the following two code fragments.

```
This is not working code"
html p with: 'Hello world.'.
html ol with: [
   html li: 'Item 1'.
   html li: 'Item 2' ].
```

```
Working version"
html paragraph with: 'Hello world.'.
html orderedList with: [
   html listItem: 'Item 1'.
   html listItem: 'Item 2' ].
```

If you can't guess the brush method name just open a class browser on the canvas class `WARenderCanvas`. Keep in mind that the method you're interested in may be in a superclass, see the hierarchy below.

Normally, the brush configuration methods that set tag attributes, use the same name as the attribute. So, for example, to set the `altText` attribute for an `IMG` (image) tag you'd send the `image` brush the `WAImageTag>>altText:` message. If you don't know the tag attribute you need to open a class browser on the specific brush class. Once again, keep in mind that the method you're interested in may be in a superclass. In addition to tag attributes, many of the Seaside brushes support convenience methods and common Javascript hacks (like setting the focus of an input field). The best way to find these is to use your tools.

When you first begin using Seaside your canvas and brush vocabulary will be limited and it

might take you a few minutes to find what you're looking for. After a while you'll discover that there is a significant shared API (implemented in the abstract superclasses) and that you are already familiar with many of the brushes. Also helpful is the autocompletion mechanism in the development environment.

=WABrush

```
WACompound
WADateInput
WATimeInput
WATagBrush
WAAnchorTag
WAImageMapTag
WAPopupAnchorTag
WAAudioTag
WABasicFormTag
WAFormTag
WABreakTag
WACanvasTag
WACollectionTag
WADatalistTag
WAListTag
WAOrderedListTag
WAUnorderedListTag
WASelectTag
WAMultiSelectTag
WACommandTag
WADetailsTag
WADivTag
WAEventSourceTag
WAFieldSetTag
WAFormInputTag
WAAbstractTextAreaTag
WAColorInputTag
WAEmailInputTag
WASearchInputTag
WASteppedTag
WAClosedRangeTag
WANumberInputTag
WARangeInputTag
WATimeInputTag
WADateInputTag
WADateTimeInputTag
WADateTimeLocalInputTag
WAMonthInputTag
WAWeekInputTag
WATelephoneInputTag
WATextAreaTag
WATextInputTag
WAPasswordInputTag
WAUrlInputTag
```

```
WAButtonTag
WACheckboxTag
WAFileUploadTag
WAHiddenInputTag
WARadioButtonTag
WASubmitButtonTag
WACancelButtonTag
WAImageButtonTag
WAGenericTag
WAEditTag
WAHeadingTag
WAHorizontalRuleTag
WAIframeTag
WAImageTag
WAKeyGeneratorTag
WALabelTag
WAMenuTag
WAMeterTag
WAObjectTag
WAOptionGroupTag
WAOptionTag
WAParameterTag
WAProgressTag
WARubyTextTag
WAScriptTag
WASourceTag
WATableCellTag
WATableColumnGroupTag
WATableColumnTag
WATableDataTag
WATableHeadingTag
WATableTag
WATimeTag
WAVideoTag
```

5.7 Rendering Lists and Tables

We will modify our `ScrapBook` to display the site contents using a list. We want to use an ordered list so we'll ask the canvas for an `WAHtmlCanvas>>orderedList` brush, which answers with an instance of `WAListTag`. Inside the body of that tag we want a collection of list item tags (`li`) which we get with the canvas' `WAHtmlCanvas>>listItem:` method. We use the short form so we don't have to use `with:` for each list item.

```
ScrapBook >> renderContentOn: html
    html heading level: 1; with: 'Hello world'.
    html paragraph: 'Welcome to my Seaside web site.  In the
        future you will find all sorts of applications here
        such as:'.
    html orderedList with: [
        html listItem: 'Calendars'.
```

```
        html listItem: 'Todo lists'.
        html listItem: 'Shopping carts'.
        html listItem: 'And lots more...' ]
```

Let's use our earlier suggestions to write this code more succinctly. We'll use `heading:` instead of `html heading level1`, and we'll use the ordered list shortcut `WAHtmlCanvas>>orderedList:`. We can use this last shortcut since we aren't configuring the ordered list tag.

```
ScrapBook >> renderContentOn: html
    html heading: 'Hello world'.
    html paragraph: 'Welcome to my Seaside web site.  In the
        future you will find all sorts of applications here
        such as:'.
    html orderedList: [
        html listItem: 'Calendars'.
        html listItem: 'Todo lists'.
        html listItem: 'Shopping carts'.
        html listItem: 'And lots more...' ]
```

Open this component in your web browser and you should see something similar to 5-8.

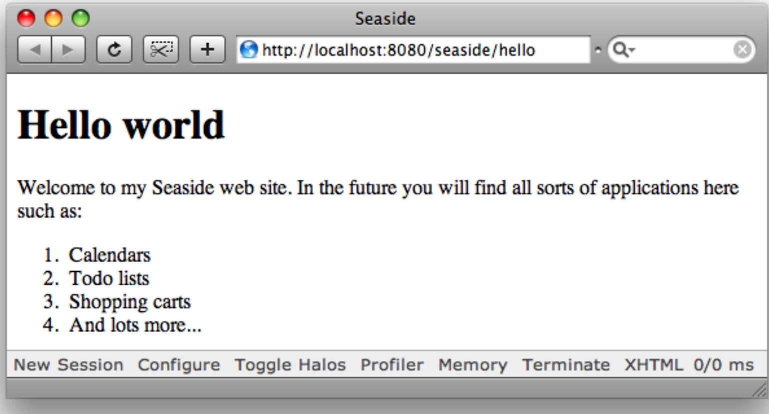


Figure 5-8 A list of items.

As good Pharoers following the DRY (Don't Repeat Yourself) principle, we can refactor this method to avoid an explicit enumeration as follows. This demonstrates the power of having a programmatic way to specify the component contents.

```
ScrapBook >> items
    ^ #('Calendars' 'Todo lists' 'Shopping carts' 'And lots more...')
```

```
ScrapBook >> renderContentOn: html
    html heading:  'Hello world'.
    html paragraph: 'Welcome to my Seaside web site.  In the
        future you will find all sorts of applications here
```

```
        such as:'.
    html orderedList: [
        self items do: [ :each | html listItem: each ] ]
```

Let's create a table of expected delivery dates. We suggest you perform a similar refactoring of the following method which illustrates `tableRow:` and `tableData:`.

```
ScrapBook >> renderContentOn: html
    html heading: 'Hello world'.
    html paragraph: 'Welcome to my Seaside web site.  In the
        future you will find all sorts of applications here
        such as:'.
    html table: [
        html tableRow: [
            html tableHeading: 'Calendars'.
            html tableData: '1/1/2006'.
            html tableData: 'Track events, holidays etc'] .
        html tableRow: [
            html tableHeading: 'Todo lists'.
            html tableData: '5/1/2006'.
            html tableData: 'Keep track of all the things to
    remember to do.' ].
        html tableRow: [
            html tableHeading: 'Shopping carts'.
            html tableData: '8/1/2006'.
            html tableData: 'Enable your customers to shop online.'
    ] ]
```

Notice that we generate table text entries in a fashion that is very similar to what we did in the list example. Note also that we used `WAHtmlCanvas>>tableHeading:` to designate a cell that represents a row header.

5.8 Style Sheets

The visual design of most modern web applications is controlled with Cascading Style Sheets (CSS). Seaside provides a simple method to add a style sheet to a component. Override the `WAComponent>>style` method in your component to return a CSS style sheet as a string, for example as follows:

```
ScrapBook>>style
    ^ 'h1 { text-align: center; }'
```

Now, refresh your browser and you should see a centered "Hello world". Bring up the halo on this component and click the pencil. Notice that you can edit the component's style method here. If you save your changes then the component's style method will be updated.

Use of the `WAComponent>>style` method is discouraged for anything but writing sample code and rapid development while experimenting with component styles. There are several reasons for this:

- It places too much emphasis on presentation at the component level and makes it difficult to reuse components in applications with a different "look". The same motivation for having XHTML separate from CSS applies to separating style documents from components.

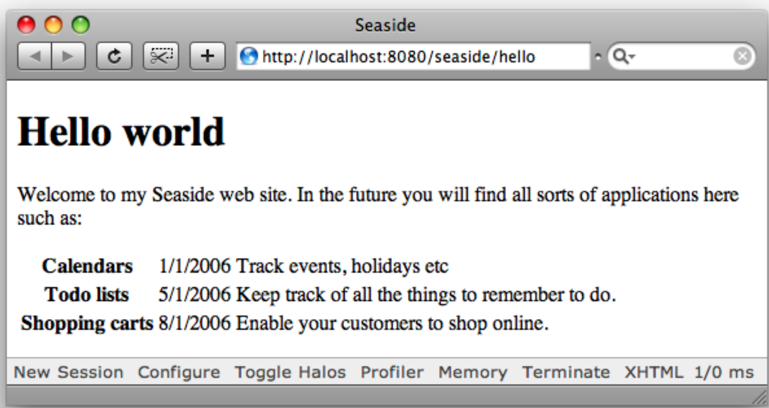


Figure 5-9 A table of items.

- Having many small `style` methods increases the load time of your pages. Each `WAComponent>>style` method is loaded as a separate document.
- Having a `style` method at the component level may give you the false impression that this style only applies to that component. In fact, CSS style sheets are loaded in the XHTML `head` section of the document and apply to the entire document, which means you have to be careful to get your CSS selectors right. It can be very difficult to track down an errant style selector when it is hidden in a component.
- It makes it more difficult to work with other non-Smalltalk developers to style your documents.

As you work through this book use the `style` method but keep in mind that a more complex application would use an external style sheet as described in Chapter **??** or file library style sheet as described in Chapter **??**.

If you've used CSS regularly then you're already familiar with using `div` (block elements) and `span` (inline elements) with the `class` attribute to help you select specific parts of a document to style.

Here's how one would, for example, give a red background to error text:

```
ScrapBook >> renderContentOn: html
    " code from previous example "
    html span
        class: 'error';
        with: 'This site does not work yet'
```

```
ScrapBook >> style
    ^ 'h1 { text-align: center; }
   span.error { background-color: red; }'
```

This book is not about XHTML or CSS but **??** provides a quick overview of CSS. We do have a few suggestions:

- Use span and div with CSS classes to mark content generated by your components. Quite often, components render themselves entirely within a div whose CSS class is the name of the component's Smalltalk class. This makes it easier for CSS developers to locate the XHTML elements that they want to style. Come up with conventions that work for you and stick to them.
- Your component's renderContentOn: method should be simple. Do not include style information, otherwise it will be difficult to customize the way in which your component is displayed.
- Your component's renderContentOn: method should be short. If your method gets longer than a couple of lines create intention-revealing helper methods following the naming pattern renderSomethingOn:. This is especially useful if you start to use conditional statements and loops. This technique will also allow you to override certain aspects of the rendering process in subclasses of your component.
- Use component style method only when the style elements are very specific to that component. Otherwise use style libraries as discussed later in Chapter .
- Try to use style sheets to structure your document's overall presentation, rather than XHTML tables. There are many good CSS references which show you how to lay out pages.

5.9 Summary

We have shown that you can specify the visual aspect of your component using the Seaside brush API. These brushes will make it easy to produce valid XHTML code. We have also demonstrated that you can use all the power of Smalltalk to specify your content, and that all the visual aspects of your application can be specified using CSS.

The method renderContentOn: is automatically invoked by Seaside. It allows you to specify the output of the application. Brushes are used to paint XHTML tags onto a canvas object. Blocks are used to create a scope within tags. For example:

```
html paragraph with: [ html render: 'today' ]
```

renders the string 'today' within <p>today</p>. A brush is an expression of the form:

```
html brush
    attributes1;
    attributes2;
    with: anObject
```

Code can be made more compact in two ways.

1. When the nested expression is a single object you can avoid blocks. The expression html paragraph with: [html render: 'today'] is equivalent to html paragraph with: 'today'.
2. When you don't need to configure the brush's attributes, you don't need to use with:. The expression html paragraph with: 'today' is equivalent to html paragraph: 'today'.

In conclusion the following three code snippets create exactly the same output. For readability and to avoid having to type unnecessary code, we usually choose the shortest version possible:

```
html paragraph with: [ html render: 'today' ].
html paragraph with: 'today'.
html paragraph: 'today'.
```

CHAPTER 6

CSS in a Nutshell

In this chapter we present CSS in a nutshell and show how Seaside helps you to use CSS in your applications. The goal of the chapter is not to replace CSS tutorials, many of which can be found on the web. Rather, the goal is to establish some basic principles and show how Seaside facilitates the decoupling of information and its visual presentation in web browsers. A clear separation between the page components and their rendering is really central to Seaside. Sometimes this frustrates newcomers because Seaside does not use template mechanisms for rendering. However, the Seaside approach allows the clear separation between the responsibilities of the web designer and the web developer. The developer is not responsible for rendering and layout of the application, this is the job of the web designer.

The idea behind CSS is to decouple the presentation from the document itself. The tags in a document are interpreted using a CSS (Cascading Style Sheet) which defines the layout and style of the rendered document. In the context of Seaside, the component rendering methods generate XHTML and the CSS associated with the application specifies how such components should be displayed and placed on the page.

6.1 CSS Principles

Basically, a CSS specification contains a set of rules. A rule is a description of a stylistic aspect of one or more elements. A rule is composed of a *selector* and a *declaration*. In the following rule `h1` is a selector which specifies that the following declaration `color: red` will be applied to all the first-level headings, see Figure 6-1. The rule has the effect that all the first level headings will be red.

```
h1 { color: red; }
```

A declaration is composed of two parts separated by a colon and ended with a semicolon. The first part is the *property* being specified, and the second is the *value* assigned to that property.

We can group multiple CSS selectors to share the same property. The following rules:

```
h1 { color: red; }
h2 { color: red; }
h3 { color: red; }
```

are equivalent to this single rule:

```
h1, h2, h3 { color: red; }
```

Similarly, it is possible to assign several values to a single selector. For example, the following rule changes the alignment and color of headings.

```
h1 { color: red; text-align: center; }
```

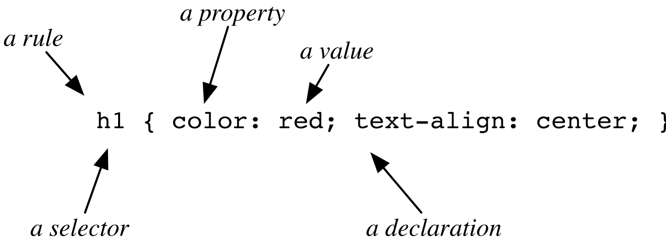


Figure 6-1 Essential CSS structural elements.

Most declarations are inherited from higher levels of your document tree. CSS property values assigned to one element are transferred down the tree to its descendants. For example, a property value set in the body of a document will be propagated to all its children, which may then redefine the value locally. This is true for color, font, etc., but not for other properties like width, height, and positioning, for which inheriting would not make sense.

While CSS declarations can be embedded in your document using a `style` tag, it is a good practice to have your CSS in a separate file. Then, if you were writing your XHTML by hand, you would add a reference to your CSS file in the head section of your document as follows.

```
<head>
<link rel="stylesheet" type="text/css" href="mystyle.css" />
</head>
```

It's not necessary to write this in Seaside though. The CSS will be served by using either the Seaside file library or with Apache, see Chapter **??**. For rapid prototyping, you can define the CSS of a component by overriding its `WAComponent>>style` method.

6.2 CSS Selectors

CSS allows you to select individual elements of an XHTML document, or groups of elements that share some property. Let's look at the different kind of selectors and how they can be used.

Tag Selector

The tag selector applies to specific XHTML tags, as we saw in the previous examples. The selector consists simply of the tag name as it appears in the XHTML source, but without the angle brackets. The following example removes the underlining from all anchor elements and changes the base font-size of the text within the page to 20 points. The `body` tag is one of the top-level tags automatically created by Seaside enclosing the whole page.

```
a { text-decoration: none; }
body { font-size: 20pt; }
```

Class Selector

The class selector is by far the most often used CSS selector. It starts with a period and usually defines a visual property that can be added to XHTML tags. The following CSS fragment defines the `center` and the `highlight` classes.

```
.center { text-align: center; }
.highlight { color: yellow; }
```

To use class selectors, simply set the `WAGenericTag>>class:` attribute of the tag you want to change. Here we associate the class selector `.center` to a given div element and `.highlight` to a given paragraph.

```
html div
    class: 'center';
    with: 'Seaside is cool'.
html paragraph
    class: 'highlight';
    with: 'Highlighted text'
```

The generated XHTML code looks like this:

```
<div class="center">Seaside is cool</div>
<p class="highlight">Highlighted text</p>
```

Multiple classes can be added to a single XHTML tag. So the following code will display a single text that is centered and highlighted:

```
html div
    class: 'center';
    class: 'highlight';
    with: 'Centered and Highlighted'
```

The generated XHTML code looks like this:

```
<div class="center highlight">Centered and Highlighted</div>
```

Often CSS classes are used to conditionally highlight certain elements of a page depending on the state of the application. Seaside provides the convenience method`WAGenericTag>>class:if:` to make this as concise as possible. The following code snippet creates ten `div` tags and adds the CSS class `even` only if the condition `index even` evaluates to `true`. This is shorter than to manually build an `ifTrue:ifFalse:` construct.

```
1 to: 10 do: [ :index |
    html div
        class: 'even' if: index even;
        with: index ]
```

Pseudo Class Selector

Pseudo classes are similar to CSS classes, but they don't appear in the XHTML source code. They are automatically applied to elements by the web browser, if certain conditions apply. These

conditions can be related to the content, or to the actions that the user is carrying out while viewing the page. To distinguish pseudo classes from normal CSS selectors they all start with a colon.

```
:focus { background-color: yellow; }
:hover { font-weight: bold; }
```

The first rule specifies that elements (typically input fields of a form) get a yellow background, if they have focus. The second rule specifies that all elements will appear in bold while the mouse cursor hovers over them.

The following table gives a brief overview of pseudo classes supported by most of today's browsers:

`:active`	Matches an activated element, e.g. a link that is clicked.
`:first-child`	Matches the first child of another element.
`:first-letter`	Matches the first character within an element.
`:first-line`	Matches the first line of text within an element.
`:focus`	Matches an element that has the focus.
`:hover`	Matches an element below the mouse pointer.
`:link`	Matches an unvisited link.
`:visited`	Matches a visited link.

Reference or ID Selector

A reference or ID is the name of *a particular* XHTML element on the page. Thus, the given style will affect only the element with the unique ID `error` (if defined). The ID selector is indicated by prefixing the ID with a # character:

```
#error { background-color: red; }
```

To create a tag with the given ID use the following Seaside code:

```
html div
    id: 'error';
    with: 'Some error message'
```

The generated XHTML code looks like this:

```
<div id="error">Some error message</div>
```

There are a couple of issues to be aware of when using IDs in your XHTML. IDs have to be unique on a XHTML page. If you use the same ID multiple times, some web browsers may not render your page as you expect, or may even refuse to render it at all. Furthermore some Javascript libraries dynamically apply their own IDs to identify page elements and these may override your carefully chosen IDs, causing your styling to fail in mysterious ways. So, to avoid invalid XHTML and conflicts with JavaScript code, do not use IDs for styling. Exclusively use CSS classes for styling, even if the particular style is used only once.

Composed Selectors

CSS selectors can be composed in various ways to give you more control over which page elements they select.

Conjunction. If you concatenate selectors without any spaces, it means that the matching element must satisfy all given selectors. This is generally used to refine the application of class or pseudo-class selectors. For example, we can write `div.error` in the style sheet and this will affect only the tags that also have the class `error`.

Similarly `a:hover` will only apply when the user moves their mouse over anchor tags. It might be tempting to specify multiple classes with `.error.highligth`. Even though this is part of the CSS standard, it does not work in older versions of Internet Explorer. **Descendant.** Another possibility is to combine selectors with a space. This finds all elements that match the first term, then searches within each for descendant elements that match the second term (and so on). For example, `div .error` selects all the elements *within* a tag that have the CSS class `error`. Elements that have the class `error` but no tag as one of its parents, are not affected.

Child. Yet another possibility is to combine two selectors with >. For example,`div > .error`. This selects all the tags with the class `error` that are direct children of a tag. Again this does not work in older versions of Internet Explorer.

There are a few more selectors available in modern web browsers to allow other criteria such as matching adjacent siblings. Since these selectors are not widely implemented in web browsers yet, we don't discuss them here.

CHAPTER 7

Anchors and Callbacks

In this chapter, you will learn to display anchors, also known as “links”. Seaside can generate traditional anchors linking to arbitrary URI’s but the most powerful use of anchors is to trigger callbacks (blocks of code) which perform actions in your applications.

You’ve already seen that Seaside uses the concepts of the *canvas* and *brushes* to insulate you from the complexities of generating valid XHTML. Similarly Seaside uses *callbacks* to hide the even greater complexities of allowing user interactions over the web.

Traditional web applications are *stateless*, that is, as soon as they have displayed a page to the user, they forget everything about that page. If the user then clicks a button on that page, the web application knows nothing about what was on the page the user was looking at, how the user got there, what choices they had made previously, and so on. If the web developers want to keep track of such information, they have to do so explicitly, by hiding information on the web page, or by saving records into a datastore every time they send a page to the user. Setting up, accessing and managing these structures takes up much of the time and energy of web developers.

In Seaside, you don’t have these problems: the current state of the program, all its variables and methods, and its history, are all stored automatically whenever a page is sent to the user, and this information is all restored for you behind the scenes if the user then performs any actions on that page.

This section will show you how to make use of all these features. We’ll introduce you to your first real web application, “iAddress”, which is a simple address book application to illustrate the points presented in this chapter and the following ones.

7.1 From Anchors to Callbacks

You can generate run-of-the-mill HTML anchors by creating an anchor brush (send `WAHtmlCanvas>>anchor` to the canvas), then configuring the anchor to be associated with a URL using `WAAnchorTag>>url:` and specifying the text for the anchor using `WAAnchorTag>>with:`. Here is a simple component that displays an anchor that displays a link to the Seaside web site.

```
WAComponent subclass: #SimpleAnchor
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'SeasideBook-Anchors'
```

```
SimpleAnchor >> renderContentOn: html
    html anchor
        url: 'http://www.seaside.st';
        with: 'Seaside Website'
```

Register this component as “simple-anchor” then view the component through your browser and you should see a page similar to 7-1.

Figure 7-1 A simple anchor.

Clicking on the *Seaside Website* anchor will bring you to the website.

7.2 Callbacks

In Seaside, anchors can be used for much more than simple links to other documents. An anchor can be assigned a *callback*, which is a block (similar to closures or anonymous methods in other languages). When the user clicks on the link, the user’s browser submits a request back to Seaside, which then runs the code in the block (it *executes* the block).

Here is an example of a component defining a callback which increases the value of the `count` variable of the component:

```
WAComponent subclass: #AnchorCallbackExample
    instanceVariableNames: 'count'
    classVariableNames: ''
    package: 'SeasideBook-Anchors'
```

```
AnchorCallbackExample >> initialize
    super initialize.
    count := 0.
```

```
AnchorCallbackExample >> anchorClicked
    count := count + 1
```

```
AnchorCallbackExample >> renderContentOn: html
    html text: count.
    html break.
    html anchor
        callback: [ self anchorClicked ];
        with: 'click to increment'
```

This method `renderContentOn:` creates an anchor element by sending `WAHtmlCanvas>>anchor` to the canvas object (html). The `anchor` method returns an instance of `WAAnchorTag` which is then used to set the callback (via the method `callback:`) and text for this anchor (via the message `with:`).

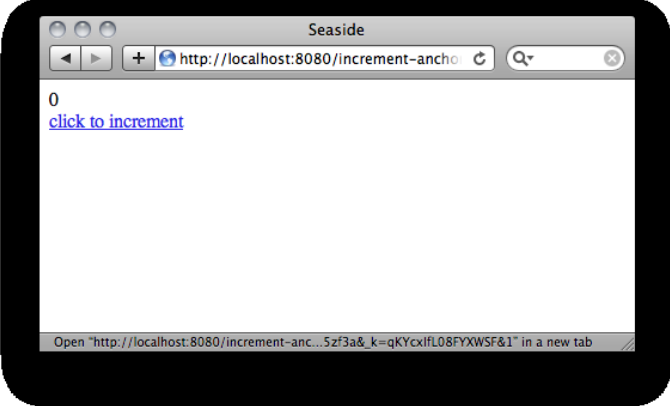


Figure 7-2 Using a callback.

When the user clicks on the anchor labelled "click to increment", the callback block is executed: it sends the message `anchorClicked` to the component which in turn increments the count. Once this callback is processed, Seaside renders our component (which will show the new count).

Register the above application as "anchor" and view it in your browser, see Figure 7-2. Clicking on the link will increment the count. Callback Processing. When Seaside receives a request, it processes all active callbacks and then asks the component to render itself. The order of this process is important. Only when it has completed processing callbacks will it move on to the rendering phase. It will become important to remember this as you build increasingly complicated applications.

Methods on `WAAnchorTag`	Description
`url: anUrl`	Specify a URL to visit when this link is clicked.
`callback: aBlock`	Specify a block that will be invoked when this link is clicked.
`with: anObject`	This specifies the anchor text.

7.3 About Callbacks

The contents of a callback are not limited to a single message send. A callback can contain any valid Smalltalk code. In a callback you can do anything you want, except that you should never use the render canvas from an outer scope. This canvas will be invalid at the time the callback is executed, as it has already been sent through the wire. Normally callbacks are not used for rendering, with the exception of AJAX. In this case AJAX passes you a new renderer into the callback, never use the old one from the outer scope. This means never refer to the `html` canvas argument in the rendering method in which the callback is declared.

It is sometimes better to put your callback code in a separate method so that you can use it from different callbacks or when subclassing your component. Some callbacks also take an argument that contains the input entered by the user (more on this later). Do not render within callbacks. Do not send any messages to the `html` canvas while processing callbacks. At the time the callback is evaluated the canvas is not active anymore. Another problem that new Seasiders might run into is that they try to change the state of their application while rendering. This will inevitably lead to confusing errors, so pay attention and memorise the following warning: Do not change state while rendering. Don't instantiate new components. Don't `call:` components. Don't `answer`. Don't add or remove decorations (`addDecoration:`, `isolate:`, `addMessage:`, etc.). Just produce output and define callbacks that do the fancy stuff you can't do while rendering.

7.4 Contact Information Model

In the next few chapters we will develop a simple application, named iAddress, which manages an email address book. We will begin by creating a new package `iAddress`, and creating in this package a class whose instances will represent the contacts in our address book.

```
Object subclass: #Contact
    instanceVariableNames: 'name emailAddress'
    classVariableNames: 'Database'
    package: 'iAddress'
```

On the instance side, add the following methods:

```
Contact >> emailAddress
    ^ emailAddress
```

```
Contact >> emailAddress: aString
    emailAddress := aString
```

```
Contact >> name
    ^ name
```

```
Contact >> name: aString
    name := aString
```

Next we provide an instance creation method and a method that creates a sample database of `Contact` instances. Note that these are *class side* methods, and should be put in an `initialization` method protocol.

```
Contact class >> name: nameString emailAddress: emailString
    ^ self new
          name: nameString;
          emailAddress: emailString;
          yourself
```

```
Contact class >> createSampleDatabase
    Database := OrderedCollection new
        add: (self name: 'Bob Jones' emailAddress:
    'bob@nowhere.com');
        add: (self name: 'Steve Smith' emailAddress:
    'sm@somewhere.com');
        yourself
```

Three more `accessing` methods should be added to the class side:

```
Contact class >> contacts
    "Answers an OrderedCollection of the contact information
    instances."

    Database isNil ifTrue: [ self createSampleDatabase ].
    ^ Database
```

```
Contact class >> addContact: aContact
    self contacts add: aContact
```

```
Contact class >> removeContact: aContact
    self contacts remove: aContact
```

We use the class variable named `Database` to store an `OrderedCollection` of `Contact` instances. Note that as you work with this class, you can always reset the database by evaluating the following expression.

```
Contact createSampleDatabase
```

7.5 Listing the Contacts

Let's create a component which displays a list of contacts:

```
WAComponent subclass: #ContactListView
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'iAddress'
```

```
ContactListView >> renderContact: aContact on: html
    html render: aContact name; render: ' '; render: aContact
    emailAddress
```

```
ContactListView >> renderContentOn: html
    html unorderedList: [
        Contact contacts do: [ :contact |
            html listItem: [ self renderContact: contact on: html ]
    ] ]
```

Notice how we split up the rendering method. This is common practice in Seaside. Register this component as "contacts" and then browse it at http://localhost:8080/contacts and you should see a window similar to the one shown in Figure 7-3.

```
WAAdmin register: ContactListView asApplicationAt: 'contacts'
```

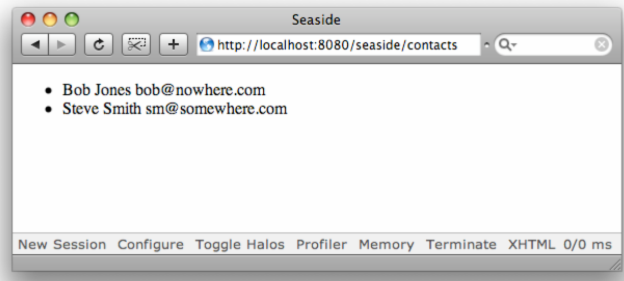


Figure 7-3 Displaying the contact database.

Already, with just a few lines of very readable code, you are able to load data from a (very simple) data store and list that data on a web page. Let's see how easy it is to start adding new records. It is actually bad design to refer to the `Contact` class directly. It would be better to add a model instance variable to our component, but for now, we will stick with what we have in the interest of simplicity.

7.6 Adding a Contact

We will modify the render method so that we can add a contact to our database, as follows. We add a callback associated with the text 'Add contact':

```
ContactListView >> renderContentOn: html
    html anchor
        callback: [ self addContact ];
        with: 'Add contact'.
    html unorderedList: [
        Contact contacts do: [ :contact |
            html listItem: [ self renderContact: contact on: html ]
    ] ]
```

```
ContactListView >> addContact
    | name emailAddress |
    name := self request: 'Name'.
    emailAddress := self request: 'Email address'.
    Contact addContact: (Contact name: name emailAddress:
    emailAddress)
```

You should now have the Add contact link as shown in 7-4.

Here we've made use of the `WAComponent>>request:` method to display a message for the user to enter a name, then another message for them to enter an email address. We will discuss the `request:` method in Chapter . Note that a real application would present a form with several fields to be filled up by the user.

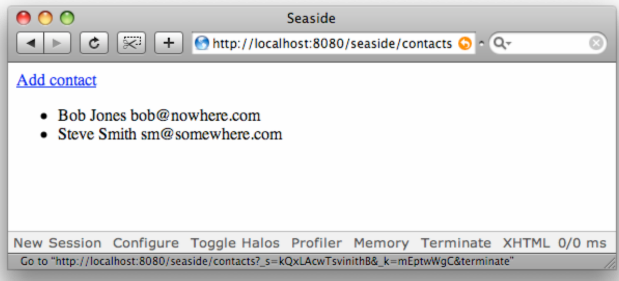


Figure 7-4 Contact list with Add contact link.

7.7 Removing a Contact

We would like to display an anchor which, when clicked, removes an item from this list. For this we just can redefine `renderContact:on:` as follows. We add another callback with the 'remove' text, see Figure 7-5.

```
ContactListView >> renderContact: aContact on: html
    html text: aContact name , ' ' , aContact emailAddress.
    html text: ' ('.
    html anchor
       callback: [ self removeContact: aContact ];
       with: 'remove'.
    html text: ')'
```

```
ContactListView >> removeContact: aContact
    Contact removeContact: aContact
```

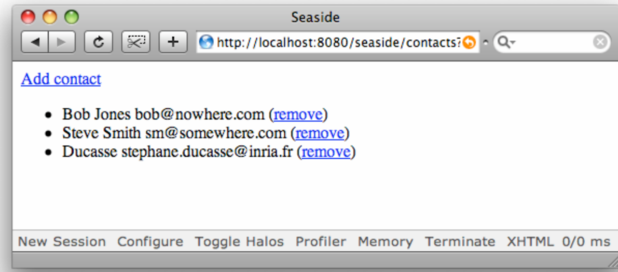


Figure 7-5 Contacts can now be removed.

Try it yourself. Click on the remove anchors. The corresponding contact entry will be removed

from the database. When you're done playing around be sure to reset the database as described at the end of Section 7.4.

It would be nice to get a confirmation before removing the item. The following method definition fixes that.

```
ContactListView >> removeContact: aContact
    (self confirm: 'Are you sure that you want to remove this
    contact?')
        ifTrue: [ Contact removeContact: aContact ]
```

We send the component (self) the message `WAComponent>>confirm:`, which displays a "Yes/No" confirmation dialog (See 7-6).

Figure 7-6 Contacts can now be removed.

The method `confirm:` returns `true` if the user answers "Yes" and `false` otherwise. This is very straightforward because Seaside handles all the complexity for you, which is amazing when you consider what a mess this kind of interaction is with many other web frameworks. Let's look at what happens:

1. The user clicks on the anchor, causing the web browser to submit a request to Seaside.
2. Seaside finds and evaluates the callback for the anchor (our block of code).
3. The callback sends `ContactListView>>removeContact:`, which in turn sends `WAComponent>>confirm:`.
4. The execution of `ContactListView>>removeContact` is **suspended**, and the confirmation page is returned to the user's web browser.
5. The user clicks the "Yes" or "No" button causing their browser to send a request to Seaside.
6. The confirmation component handles this request, "answering" `true` if the user clicked "Yes" and `false` otherwise.
7. When the confirmation component answers, the `ContactListView>>removeContact:` method **resumes** execution and processes the answer from `confirm:`, deleting the contact item if the answer was `true`.

So, in Seaside, it is easy for a method to display another component, wait for the user to interact with it, and then resume execution when that component has completed its job. This is akin to *modal dialogs* in Graphical User Interface (GUI) applications, see Chapter 9.

7.8 Creating a mailto: Anchor

In this section, we add “mailto:” links to our `ContactListView`. Users of our application can then simply click on the e-mail address to send an e-mail, assuming that their web browser is properly configured to respond to `mailto:` links. As discussed in , we can specify the URL for an anchor explicitly. Here is the modified version of our rendering method:

```
ContactListView >> renderContact: aContact on: html
    html text: aContact name.
    html space.
    html anchor
        url: 'mailto:' , aContact emailAddress;
        with: aContact emailAddress.
    html text: ' ('.
    html anchor
        callback: [ self removeContact: aContact ];
        with: 'remove'.
    html text: ')'
```

Test this new component in your browser to see that your mailto: links are working correctly, see Figure 7-7.

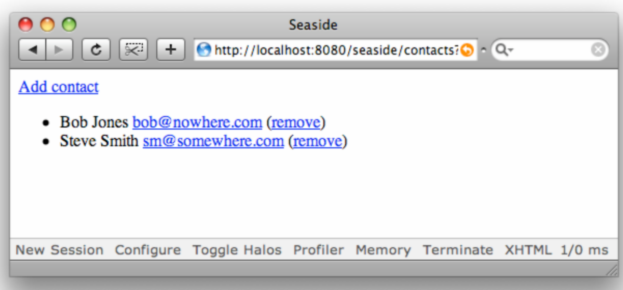


Figure 7-7 With mailto anchor.

Note that rather than manipulating strings in this way, experienced Pharoers might want to actually define an “email address” class to handle the different representations of email addresses. In fact, Seaside 3.0 already defines a class `WAEmailAddress` which may be used for this very purpose.

7.9 Summary

In this chapter you saw callbacks, a powerful feature of Seaside. Using a callback, we can attach an action or a small program to a link or button that will be executed only when the element is activated. What is really powerful is that you can write any Pharo code in a callback. In the next chapter, we will continue to enhance the iAddress application to show you how to handle forms.

CHAPTER 8

Forms

In this chapter, we describe how to use XHTML forms and controls in Seaside. Controls such as text input fields, popup lists, radio buttons, check boxes, date inputs, and buttons are always created within an XHTML element. In this chapter we show how to create this element and how to use these common controls.

8.1 Text Input Fields and Buttons

Let's continue with the same `Contact` domain model class that we used in Chapter . We wish to create a form that allows the user to update the name and email address for a `Contact`. Smalltalkers are generally very careful to separate presentation from the core of the data model, so we will create a component that will hold the user interface code to allow the user to edit the underlying `Contact` instance:

```
WAComponent subclass: #ContactView
    instanceVariableNames: 'contact'
    classVariableNames: ''
    package: 'iAddress'
```

Notice that we've specified a `contact` instance variable; this will be used to hold a reference to the instance of the `Contact` class that we want to edit. Now we create the accessors for the `contact` instance variable. Look carefully at the `contact` method below. Before returning the value of the `contact` instance variable, it checks that it has been set. If it hasn't been set, the code in the block is executed which assigns a sensible value to the variable. This *lazy initialisation* is a common idiom in Smalltalk code. At the moment we want to test this component as a stand alone component, so the accessor method will lazily load one of the contacts for us.

```
ContactView >> contact
    ^ contact ifNil: [ contact := Contact contacts first ]
```

```
ContactView >> contact: aContact
    contact := aContact
```

Next, we introduce our first new canvas message: the message `form`. This method returns an instance of `WAFormTag`. The only message in `WAFormTag` of interest to us right now is `with:`

which, as we've seen before, takes an argument and renders that argument between the open and close XHTML tags (i.e. the `<form>` and `</form>` tags in this case). Controls such as input fields, buttons, popups, list boxes, and so on must all be placed inside a `form` element. Forgetting to add the form element is a common mistake. Controls such as input fields, buttons, popups, list boxes etc., must all be placed inside a `form` tag. If not, they may not be rendered, or they may be rendered but then ignored by the browser. Our form will have three elements: two text boxes, one each for the name and the email address; and a button for the user to submit their changes.

Let's look first at the text fields for the name and e-mail address inputs. These fields are created by the canvas' `textInput` message which returns a `WATextInputTag`. For each brush we use two methods `value:` and `callback:` . The `value:` method determines what should be put into this field when it is displayed to the user; here we use the accessor methods on the `Contact` instance to give these values. The `callback:` method takes a block that has a single argument. When the user submits the form, the block will have the new contents of the field passed to it using this argument; here we use this to update the `Contact` instance (via its accessor methods).

Finally we would like our component to have a *Save* button. We create a button with the canvas `submitButton` method, which answers a `WASubmitButtonTag`. We assign a callback so that when the user presses this button the message `save` is sent.

Here's the rendering method which creates two text inputs and a submit button:

```
ContactView >> renderContentOn: html
    html form: [
        html text: 'Name: '.
        html textInput
            callback: [ :value | self contact name: value ];
            value: self contact name.
        html break.
        html text: 'Email address: '.
        html textInput
            callback: [ :value | self contact emailAddress: value ];
            value: self contact emailAddress.
        html break.
        html submitButton
            callback: [ self save ];
            value: 'Save']
```

```
ContactView >> save
    "For now let's just display the contact information"
    self inform: self contact name , '--' , self contact emailAddress
```

The brushes `submitButton` and `textInput` you can also use the message `value:` and `with:` interchangeably. They both define the contents of the button of text input field. When the user's browser submits this form, first all the input callbacks are processed, then the (single) submit button callback will be processed. The order is important because the input callbacks set the corresponding field in the `Contact` instance. The `save` method expects those fields to be set before it is invoked. Important You should remember that Seaside processes all input field callbacks before the submit button callback. Register this component as a new application called "contact", see Section **??** for details. Point your web browser to http://localhost:8080/contact and you should see the form as shown in Figure **??**. Try entering values and submitting the form.

Brush Message Summary The two following tables show a summary of the most useful `textInput` and `submitButton` brush methods.

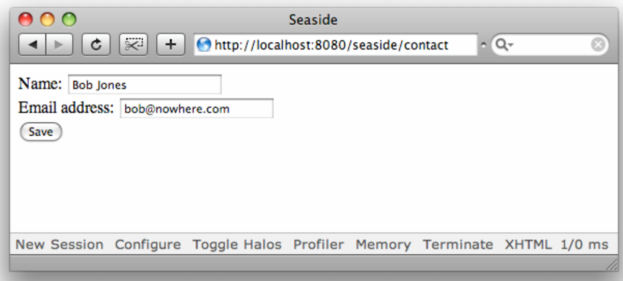


Figure 8-1 Filling up our contact view.

Methods on `WATextInputTag`	Description
`callback: aBlock`	Specify a single argument callback block which is passed the string value entere
`value: aString`	Specify an initial value for the text input.
`on: aSymbol of: anObject`	This is a convenience method explained in the next section.

Methods on `WASubmitButton`	Description
`callback: aBlock`	Specify a zero-argument callback block which is invoked when the user clicks th
`value: aString`	Specify a label for this submit button.
`on: aSymbol of: anObject`	This is a convenience method explained in the next section.

8.2 Convenience Methods

Seaside offers also some convenience methods that make your code shorter. Let's have a look at them.

Text input fields. The initial value of an input field often comes from an accessor method on some class (for example `self contact name`). Similarly your input field callbacks will often look like those in the previous example, and simply take the text that the user entered and store it using a similar method name (for example `self contact name: value`). Because this is such a common pattern, text input brushes provide the method `on:of:`, which does this automatically for you so you can write:

```
html textInput on: #name of: self contact
```

instead of:

```
html textInput
   callback: [ :value | self contact name: value ];
   with: self contact name
```

Buttons. Similarly, the label of a submit button can often be inferred from the name of the method it invokes. Submit button brushes provide the method `on:of:`, which does this automatically for you allowing you to write one line:

```
html submitButton on: #save of: self
```

instead of all of this:

```
html submitButton
   callback: [ self save ];
   value: 'Save'
```

The actual conversion from the selector name to the button label happens by sending `labelForSelector:` to the second argument. The default implementation of this method simply capitalizes the first letter of the selector and returns a string, but applications might decide to customize that method by overriding it. **Text fields.** For text fields, the `on:of:` method takes the (symbol) name of the property to be edited and the object which holds the property.

Specifying method names. Seaside generates method names from the property names using the usual Smalltalk accessor/mutator naming conventions. For example, a property called `#name` would use a method called `name` as an accessor and a method called `name:` as a mutator. The accessor is used to provide the starting value for the field and the mutator is used in a callback to set the value of the property.

Generating labels. For submit buttons, `on:of:` takes the name of the method to invoke and the object to which to send the message. It will use the method name to generate a label for the button with a bit of intelligence. The symbol `#save` becomes the label "Save", whereas the symbol `#youCanUseCamelCase` becomes "You Can Use Camel Case". If you don't like this translation, use the `callback:` and `value:` methods, as demonstrated in the last section.

So, putting all these techniques to work, our render method could be changed to:

```
ContactView >> renderContentOn: html
    html form: [
        html text: 'Name:'.
        html textInput on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textInput on: #emailAddress of: self contact.
        html break.
        html submitButton on: #save of: self ]
```

All of the Seaside input components support both the `on:of:` and the more primitive `callback:` and `value:` methods, so we will use whichever form makes our code the more readable. Anchors also support `on:of:`. As we mentioned above, controls such as input fields, buttons, popups, list boxes, and so on must all be placed inside a `form` tag. Typically only a single `form` tag is needed on a page. `form` tags *must not* be nested but multiple `form` tags can occur, one after another, on a single page. Only the contents of a single `form` will be submitted by the web browser though (normally determined by the form in which the user clicked a submit button).

8.3 Drop-Down Menus and List Boxes

XHTML provides a single element, `select`, which can be shown by web browsers as a drop-down menu or a list box, depending on the parameters of the element. In this section, we look at examples of each type. We'll start with a drop-down menu.

For the sake of an example, let's track the gender of each of our contacts. Change the `Contact` class definition to include the `gender` instance variable and add the methods which manipulate it, as shown below.

```
Object subclass: #Contact
    instanceVariableNames: 'name emailAddress gender'
    classVariableNames: 'Database'
    package: 'iAddress'
```

```
Contact >> gender
    ^ gender ifNil: [ gender := #Male ]
```

```
Contact >> isMale
    ^ self gender = #Male
```

```
Contact >> isFemale
    ^ self gender = #Female
```

```
Contact >> beMale
    gender := #Male
```

```
Contact >> beFemale
    gender := #Female
```

We would like to add a drop-down menu to our editor that allows the user to indicate the gender of someone in the contact list. The simplest way to do this is with the canvas' `select` method. This method returns a `WASelectTag`.

The following method shows how the `select` brush can be parametrized to render a list for gender selection.

```
ContactView >> renderContentOn: html
    html form: [
        html text: 'Name:'.
        html textInput on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textInput on: #emailAddress of: self contact.
        html break.

        "Drop-Down Menu"
        html text: 'Gender: '.
        html select
            list: #(#Male #Female);
            selected: self contact gender;
            callback: [ :value |
                value = #Male
                    ifTrue: [ self contact beMale ]
                    ifFalse: [ self contact beFemale ] ].
        html break.

        html submitButton on: #save of: self ]
```

Notice that `selected:` allows us to specify which item is selected by default (when the list is first displayed). Let's update the `save` method to display the gender as follows:

```
ContactView >> save
    self inform: self contact name ,
        '--' , self contact emailAddress ,
        '--' , self contact gender
```

Try the application now. You should see a drop-down menu to select the gender, as shown in Figure **??**.

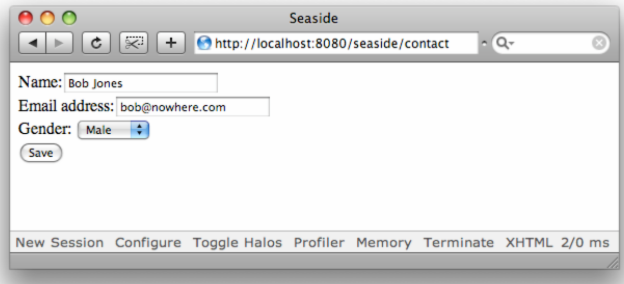


Figure 8-2 Gender as a drop-down menu.

Modify the gender input so that it specifies a list size:

```
ContactView >> renderContentOn: html
    html form: [
        html text: 'Name:'.
        html textInput on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textInput on: #emailAddress of: self contact.
        html break.

        "List Box"
        html text: 'Gender: '.
        html select
            size: 2;
            list: #(#Male #Female);
            selected: self contact gender;
            callback: [ :value |
                value = #Male
                    ifTrue: [ self contact beMale ]
                    ifFalse: [ self contact beFemale ] ].
        html break.
        html submitButton on: #save of: self]
```

Experienced Pharoers will be getting concerned at the length of this method by now. Generally it is considered good practice to keep your methods to a few lines at most. For the purposes of this exercise, we will be ignoring this good practice, but you may want to think about how you could split this method up. Now view the application in your browser. Most browsers will show a list rather than a drop-down menu, see **??**.

Select Brush Message Summary. The following table shows a summary of the most important message of the `select` brush.

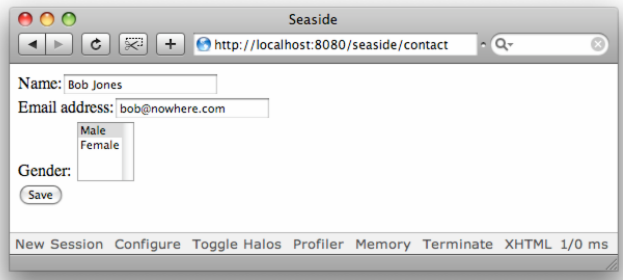


Figure 8-3 With radio buttons.

Methods on `WASelectTag`	Description
`list: aCollection`	Specify the list of options from which to select.
`selected: anObject`	Specify the object which should be shown as selected by default.
`callback: aBlock`	Specify a single-argument callback block which will be passed the object selected
`size: anInteger`	Specify the number of rows of the list that should be visible. Note, if you don't sp
`on: aSymbol of: anObject`	This is a convenience method as explained previously.

8.4 Radio Buttons

In our gender example above, the list is a bit of overkill. Let's present the user with radio buttons instead. We create radio buttons with the canvas' `radioButton` message, which returns a `WARadioButtonTag`. Radio buttons are arranged in groups, and radio buttons in a group are mutually exclusive, so only one can be selected at a time.

We will make two changes to our `renderContentOn:` method: declare a local variable named `group` and replace the list code with a radio button definition, as shown in the following method:

```
ContactView >> renderContentOn: html
    | group |
    html form: [
        html text: 'Name:'.
        html textInput on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textInput on: #emailAddress of: self contact.
        html break.

        "Radio Buttons"
        html text: 'Gender: '.
        group := html radioGroup.
        group radioButton
            selected: self contact isMale;
            callback: [ self contact beMale ].
        html text: 'Male'.
```

```
        group radioButton
            selected: self contact isFemale;
            callback: [ self contact beFemale ].
        html text: 'Female'.
        html break.

        html submitButton on: #save of: self ]
```

First, we ask the canvas to create a new group using `radioGroup`. We then ask the group for a new radio button using the message `radioButton`. The `selected:` message determines if the browser will render the page with that button selected. Notice in our example that we select the button if it corresponds to the current value of the `gender` variable. That way the form reflects the state of our component.

The `callback:` method should be a zero argument callback block which is executed when the page is submitted with that radio button selected. Note the callback block is not called for options that were not selected.

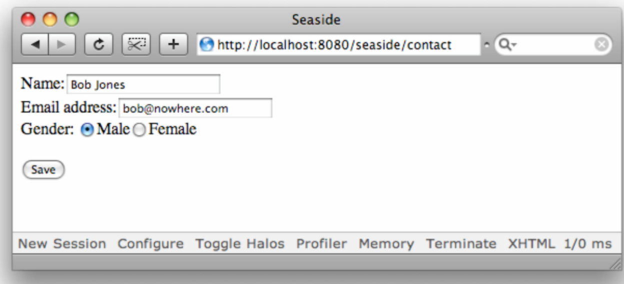


Figure 8-4 Filling up our contact view.

Radio Button Brush Summary. The following table gives a summary of the most important `radioButton` brush messages.

Methods on `WARadioButtonTag`	Description
`group: aRadioGroup`	Specify the radio group to which this button belongs.
`selected: aBoolean`	Specify a boolean value that indicates whether this radio button is initially selecte
`callback: aBlock`	Specify a zero argument callback block which is called if this button is selected wh

8.5 Check Boxes

Let's modify our model class again. This time we will add an instance variable (and accessors) for a Boolean property that indicates if a contact wants to receive e-mail updates from us.

```
Object subclass: #Contact
    instanceVariableNames: 'name emailAddress gender
    requestsEmailUpdates'
    classVariableNames: 'Database'
    package: 'iAddress'
```

```
Contact >> requestsEmailUpdates
    ^ requestsEmailUpdates ifNil: [ requestsEmailUpdates := false ]
```

```
Contact >> requestsEmailUpdates: aBoolean
    requestsEmailUpdates := aBoolean
```

We will use the canvas' `checkbox` method to produce a `WACheckboxTag` as shown in the following method. Checkboxes are useful for boolean inputs such as `requestsEmailUpdates`.

```
ContactView >> renderContentOn: html
    | group |
    html form: [
        html text: 'Name:'.
        html textInput on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textInput on: #emailAddress of: self contact.
        html break.
        html text: 'Gender: '.
        group := html radioGroup.
        group radioButton
            selected: self contact isMale;
            callback: [ self contact beMale ].
        html text: 'Male'.
        group radioButton
            selected: self contact isFemale;
            callback: [ self contact beFemale ].
        html text: 'Female'.
        html break.

        "Checkbox"
        html text: 'Send email updates: '.
        html checkbox
            value: self contact requestsEmailUpdates;
            callback: [ :value | self contact requestsEmailUpdates:
    value ].
        html break.

        html submitButton on: #save of: self ]
```

Next, update the display method to show the value of this flag. Figure 8-5 shows our new form.

```
ContactView >> save
    self inform: self contact name ,
        '--' , self contact emailAddress ,
        '--' , self contact gender ,
        '--' , self contact requestsEmailUpdates printString
```

Try the application now. Fill out the form and submit it to see that the checkbox is working.

Model adaptation. It often requires some work to get the model and the UI (web or graphical) to communicate with each other effectively. For example, we can't write this inside the render method:

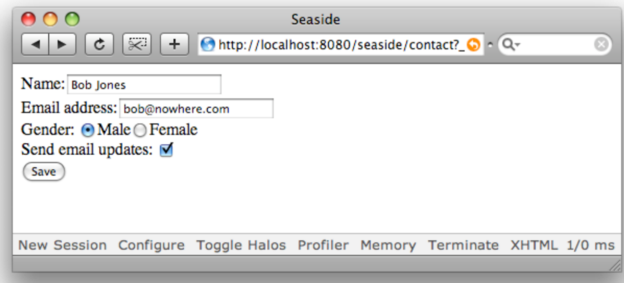


Figure 8-5 With Check boxes.

```
html textInput on: #gender of: model
```

This is because the `on:of:` method expects the property to have accessors and mutators. In this case we can either configure the brush with `callback:>>callback:` and `value:`, or use a go-between or adapter method in our view class. This is what we did in the example above. Callbacks are flexible and designed for specifying such an interface.

Checkbox Brush Message Summary. The following table shows a summary of the most important messages of the `checkbox` brush.

Methods on **WACheckboxTag**	Description
`callback: aBlock`	Specify a one-argument callback block which will be passed true or false.
`on: aSymbol of: anObject`	This is a convenience method.
`onTrue: aBlock onFalse: aBlock`	Specify two zero-argument blocks. One is performed if the box is checked wh
`value: aBoolean`	Specify the initial value for the checkbox.

8.6 Date Inputs

Using the canvas' `dateInput` method is the simplest way to provide a date editor. It produces a `WADateInput`, which understands the messages `callback:` and `value:`.

First, add a `birthdate` instance variable to the class `Contact` and produce accessor methods for it. You should be familiar with how to do this by now. If not, look back at the changes you've made in the previous sections.

Then add the following code to the `form:` block on your `renderContentOn:` method:

```
    html dateInput
        callback: [ :value | self contact birthdate: value ];
        with: self contact birthdate.
    html break.
```

For those who like to see the date presented in a different order, cascade-send `options: #(day month year)` to the brush.

Your `renderContentOn:` method should now look like this:

```
ContactView >> renderContentOn: html
    | group |
    html form: [
        html text: 'Name:'.
        html textInput on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textInput on: #emailAddress of: self contact.
        html break.
        html text: 'Gender: '.
        group := html radioGroup.
        group radioButton
            selected: self contact isMale;
            callback: [ self contact beMale ].
        html text: 'Male'.
        group radioButton
            selected: self contact isFemale;
            callback: [ self contact beFemale ].
        html text: 'Female'.
        html break.
        html text: 'Send email updates: '.
        html checkbox
            value: self contact requestsEmailUpdates;
            callback: [ :value | self contact requestsEmailUpdates:
    value ].
        html break.

        "Date Input"
        html text: 'Birthday: '.
        html dateInput
            callback: [ :value | self contact birthdate: value ];
            with: self contact birthdate.
        html break.

        html submitButton on: #save of: self ]
```

Finally update our display method, as below:

```
ContactView >> save
    self inform: self contact name ,
        '--' , self contact emailAddress ,
        '--' , self contact gender ,
        '--' , self contact requestsEmailUpdates printString ,
        '--' , self contact birthdate printString
```

Because the `birthdate` instance variable is `nil`, the date is displayed with the current date. Your final version of the application should look something like 8-6.

Date Message Summary. The following table shows the messages of the `dateInput` brush.

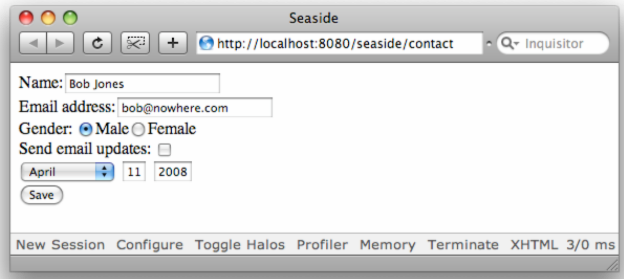


Figure 8-6 Full contact view.

Methods on `WADateInput`	Description
`callback: aBlock`	Specify a single argument callback block which is passed a `Date` object representing the
`with: aDate`	Specifies the date with which we will initialize our date editor. If `aDate` is `nil`, today's d
`options: anArray`	Specify the order in which the `#day`, `#month` and `#year` fields are rendered.

8.7 File Uploads

You can use a form to allow a user to upload a file. Here is how you would construct a simple form for a file upload.

```
UploadForm >> renderContentOn: html
    html form multipart; with: [
        html fileUpload
            callback: [ :value | self receiveFile: value ].
        html submitButton: 'Send File' ]
```

This code will produce a form, as shown in 8-7. Some browsers may also display a text input field so you can enter the file path.

Important You must send the `multipart` message to the form brush if you want to use the form for file uploads (as shown in this example). If you forget to do this, the upload won't work. Press the *Choose File* button to select the file, then press *Send File* to start the upload. When the upload completes, Seaside will invoke our callback which sends the `receiveFile:` message with a `WAFile` object that represents your file. As an example, the method below will save the file in a directory called `uploads` in Pharo. change fileDirectory....

```
UploadForm >> receiveFile: aFile
    | stream |
    stream := (FileDirectory default directoryNamed: 'uploads')
        assureExistence;
        forceNewFileNamed: aFile fileName.
    [ stream binary; nextPutAll: aFile rawContents ]
        ensure: [ stream close ]
```

Note that it is possible to press the *Send File* button before selecting a file. In this case the callback is not triggered.

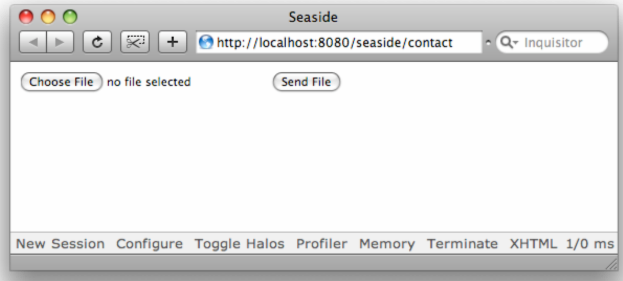


Figure 8-7 File uploads.

To make this method more robust, you could also check for empty files and devise a check that would prevent malicious persons from uploading many large files in an attempt to fill up your disk space. Note also that this method will quietly replace an existing file with the same name. You could easily change this method so that it checks for duplicate file names or tests the size of the file before saving it.

Alternatively you can offer to download the file again. Add an instance variable called `file` and change your code according to the following example:

```
UploadForm >> renderContentOn: html
    html form multipart; with: [
        html fileUpload
            callback: [ :value | file := value ].
        html submitButton: 'Send File' ].
    file notNil ifTrue: [
        html anchor
            callback: [ self downloadFile ];
            with: 'Download' ]
```

```
UploadForm >> downloadFile
    self requestContext respond: [ :response |
        response
            contentType: file contentType;
            document: file rawContents asString;
            attachmentWithFileName: file fileName ]
```

8.8 Summary

Seaside makes it easy to display forms with buttons, popup lists, checkboxes, etc. It is easy in part because of the callbacks that it uses to inform you of the value of these items. In , we saw how callbacks are used with anchors. Seaside uses the power of callbacks to make your job much easier.

Part III

Using Components

This part describes the core of Seaside: its component model. In Seaside, components are generally designed to have a direct relationship with the state of some part of the underlying model, and to take advantage of their state to change the way they display themselves and interact with the user. The fact that this state is encapsulated locally in the component, rather than stored globally as "session state", sets Seaside apart from most other web application development frameworks. For example, a list can be made responsible for holding the currently selected item or a calendar the currently selected date. In fact, you've already started building your applications this way: in the `ContactView` knew which contact it was editing. Understanding how best to use these stateful components, and how to build interactions between them, allows us to build widgets just as we would for desktop applications.

You have already seen how components can be created and how they can display themselves on the web page. This section will demonstrate how you can have more control over these processes and how components can interact with other components. You will see two forms of interaction. A component can *embed* content and functionality from other components into its own web page; alternatively, it can *call* other components, allowing them to take over its web page until they return a result to the main component. *Tasks* can be used to give more control over these interactions.

You will also see how a pre-defined component can be given different behaviour and appearance to allow it to be re-used in different ways. This reuse is achieved in Seaside via component *decoration*.

Finally, you will see a discussion of "Slime", which despite its name is an extremely useful library to check and validate your Seaside code.

CHAPTER 9

Calling Components

Seaside applications are based on the definition and composition of components. Each component is responsible for its rendering, its state and its own control flow. Seaside lets us freely compose and *reuse* such components to create advanced and dynamic applications. You have already built several components. In this chapter, we will show how to reuse these components by "calling" them in a modal way. Embedding components in other components will be discussed in the following chapter.

9.1 Displaying a Component Modally

Seaside components have the ability to specify that another component should be rendered (usually temporarily) in their place. This mechanism is triggered by the message `call:`. During callback processing, a component may send the message `call:` with another component as an argument. The component passed as an argument in this way can be referred to as the *delegate*. The `call:` method has two effects:

1. In subsequent rendering cycles, the delegate will be displayed in place of the original component. This continues until the delegate sends the message `WAComponent>>answer` to itself.
2. The current execution state of the calling method is suspended and does not return a value yet. Instead, Seaside renders the web page in the browser (showing the delegate in place of the original component).

The delegate may be a complex component with its own control flow and state. If the delegate component later sends the message `answer`, then execution of the (currently suspended) calling method is resumed at the site of the `call:`. We will explain this mechanism in detail after an example. Important From the point of view of a component, it calls another component and that component will (eventually) answer.

9.2 Example of Call/Answer

To illustrate that mechanism, let's use the `ContactListView` and `ContactView` components developed in earlier chapters. Our goal is simple: in a `ContactListView` component, we will

display a link to edit the contacts (as shown in ??), and when the user selects that link, display the `ContactView` on that `Contact`. We accomplish this using the `call:` message.

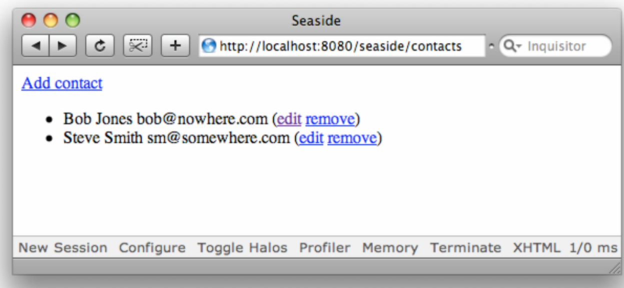


Figure 9-1 New version of the `ContactListView`.

The `editContact:` method is passed a contact as an argument. It creates a `ContactView` component for the contact and calls this new component by sending it the message `call:`.

```
ContactListView >> editContact: aContact
    self call: (ContactView new
        contact: aContact;
        yourself)
```

Next, we change the method `ContactListView>>renderContact:on:` to invoke the method we just defined when the edit link is selected, as below:

```
ContactListView >> renderContact: aContact on: html
    html text: aContact name , ' ' , aContact emailAddress.
    html text: ' ('.
    html anchor   " <-- added "
        callback: [ self editContact: aContact ];
        with: 'edit'.
    html space.
    html anchor
        callback: [ self removeContact: aContact ];
         with: 'remove'.
    html text: ')'
```

In the previous chapters, the `save` method of the `ContactView` component just displayed the contact values using a dialog. Now, using the message `answer`, we are able to return control from the newly created `ContactView` component to the `ContactListView` which created it and called it. Modify the `ContactView` so that when the user presses *Save* it returns to the caller (the `ContactListView`):

```
ContactView >> save
    self answer
```

Have a look at the way the method `editContact:` creates a new instance of `ContactView` and then passes this instance as an argument to the `call:` message. When you call a component,

you're passing control to that component. When that component is done (in this case the user pressed the *Save* button), it will send the message `answer` to return control to the caller.

Interact with this application now and follow the link. Fill out the resulting form and press the *Save* button. Notice that you're back to the `ContactListView` component. So, you `call:` another component and when it is done it should `answer`, returning control of the display to the caller. Important You can think of the call/answer pair as the Seaside component equivalent of raising and closing a modal dialog respectively.

9.3 Call/Answer Explained

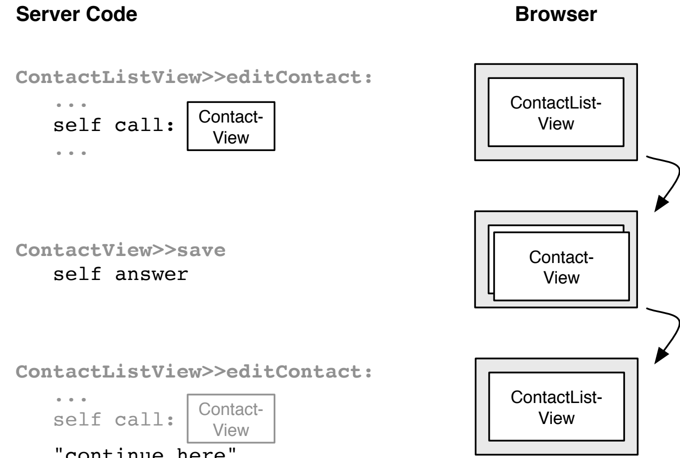


Figure 9-2 Call and Answer at Work.

9-2 illustrates the call/answer principle. The application is showing our `ContactListView` component.

When the user presses *edit* next to a contact name, the `ContactListView` component executes its callbacks until it reaches the line `self call: ...`, where it sends the message `call:` and passes it the `ContactView` component. This causes `ContactView` to take control of the browser region occupied by `ContactListView`. Note that the other component `A`, can continue to be active; this is an example of having multiple, simultaneous control flows in an application.

When the component `ContactView` reaches the line `self answer`, it sends the message `answer`, which has the effect of closing the `ContactView` component, giving back control to the `ContactListView` component, and possibly returning a value, as you will see in the next section. The returned value can be a complex object such as credit card information or a complete Contact object, or it can be as simple as a primitive object such an integer or a string. With Seaside you handle objects directly and there is no need to translate or marshall them in order to pass them around different components.

After the `answer` message send, the execution of the component `ContactListView` continues just after the call that opened the component `ContactView`. This is marked as `"continue here"` in the diagram.

9.4 Component Sequencing

As we just showed, calling is a *modal* interaction, that is, the method `call:` doesn't return until the component it called answers. That allows us to sequence component display.

```
ContactListView >> editContact: aContact
    | view |
    view := ContactView new.
    view contact: aContact.
    self call: view.
    self inform: 'Thanks for editing  ' , aContact name
```

Let's suppose that you have redefined the method `editContact:` as shown above. The method calls the view component and then, after the view answers, it displays a message. Here's something to wrap your brain around. What if the user fills in the form, presses the *Save* button, then presses *Back* and changes the values in the form and saves again? After the first save, the above method calls `WAComponent>>inform:` but when the user presses *Back* your method backs up into the `call:` of `ContactView`.

What Seaside does is the following: It snapshots the state of execution of your method so that it can jump back in response to the *Back* button. We'll go into much more detail about this later in . For now just try it and confirm that things work as you'd expect.

9.5 Answer to the Caller

There is a version of the `answer` message which takes an argument. This version, named `answer:` returns a value to the caller. One common use of this is to return a boolean to indicate if the user canceled or completed the operation. Since we don't have a cancel button in our `ContactView`, let's add one and answer appropriately, see 9-3.

But before doing that, we will refactor the `renderContentOn:` method. It's too long and overdue for refactoring. Using the refactoring capabilities of your favorite browser, extract methods so that it looks like this.

```
ContactView >> renderContentOn: html
    html form: [
        self renderNameOn: html.
        self renderEmailOn: html.
        self renderGenderOn: html.
        self renderSendUpdatesOn: html.
        self renderDateOn: html.
        self renderSaveOn: html ]
```

Now edit your new `renderSaveOn:` method to add a cancel button:

```
ContactView >> renderSaveOn: html
        html submitButton on: #cancel of: self. " <-- added "
        html submitButton on: #save of: self
```

Redefine the following methods to cancel and save the editing.

```
ContactView >> save
    self answer: true
```

```
ContactView >> cancel
    self answer: false
```

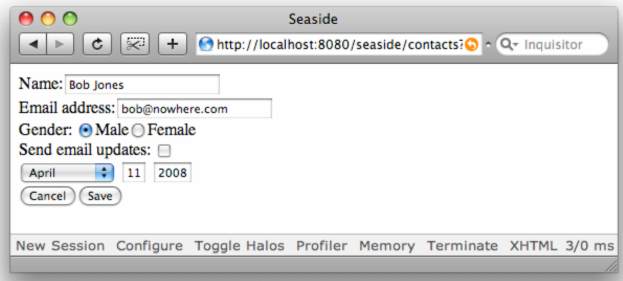


Figure 9-3 Contact edition with a cancel button.

Now we can change the method `ContactViewList>>editContact:` to use the returned value to avoid showing the final `inform:` dialog.

```
ContactListView >> editContact: aContact
    | view |
    view := ContactView new.
    view contact: aContact.
    (self call: view)
        ifFalse: [ ^ nil ].
    self inform: 'Thanks for editing ' , aContact name
```

If you try using the application as it currently stands, you may get a nasty surprise: if the user changes the name in the form and then presses *Cancel*, the underlying object will still be updated! So, rather than passing it the object we want to edit, we should instead pass it a copy of the contact to be edited and then, depending on the result passed by the `answer:`, decide whether to substitute the corresponding contact in the contact list.

```
ContactListView>>editContact: aContact
    | view copy |
    view := ContactView new.
    copy := aContact copy.
    view contact:  copy.
    (self call: view)
        ifTrue: [ Contact removeContact: aContact; addContact: copy ]
```

When you edit a user now, you'll notice that the user ends up moved to the end of the list of the users; this is the expected behaviour.

9.6 Don't call while rendering

One of the most common mistakes for first-time Seaside developers is to send the message `call:` a component from another component's rendering method, `renderContentOn:`. The rendering method's purpose is *rendering*. Its only job is to display the current state of the component. Callbacks are responsible for changing state, calling other components, etc. If you want

to render one component inside another one read . Important Don't `call:` a component from `renderContentOn:`, only call components from callbacks or from `WATask` subclasses.

9.7 A Look at Built-In Dialogs

Now it's time to look at the source code for the `WAComponent>>inform:` method from the `WAComponent` class. Do not type this code, it is already part of Seaside. The definition of the method `inform:` of the class `WAComponent` is the following one.

```
WAComponent >> inform: aString
    "Display a dialog with aString to the user until he clicks the
    ok button."

    ^ self wait: [ :cc | self inform: aString onAnswer: cc ]
```

The method `inform:` sends the message `wait:` to raise a newly created `WAFormDialog` component, exactly the same way as `call:` does.

How do you find related methods? Looking through `WAComponent` reveals:

- `WAComponent>>inform:` displays a dialog with a message to the user until he clicks the button.
- `WAComponent>>confirm:` displays a message and *Yes* and *No* buttons. Returns true if user selected *Yes*, false otherwise.
- `WAComponent>>request:`, `WAComponent>>request:default:`, `WAComponent>>request:label:`, and `WAComponent>>request:label:default:` display a message, an optional label and an input box. The string entered into the input box is returned. If the `default:` argument is specified it is used for the initial contents of the input box.
- `WAComponent>>chooseFrom:`, `WAComponent>>chooseFrom:caption:`, `WAComponent>>chooseFrom:default:`, and `WAComponent>>chooseFrom:default:caption:` display a drop-down list with different choices to let the user choose from. A default selection and a title can be given. The methods answer the selected item.

9.8 Handling The Back Button

Web browsers allow you to navigate your browsing history using the back button. The problem is that when you press the back button, the application interface and the underlying model can be out of sync. When you press the back button, only the browser is involved and not the server and the server has no way to know that you changed. Therefore your UI can be out of sync from its domain. Seaside offers you a way to control the back button effect.

There are two kinds of synchronization problems: UI state and model state. Seaside offers a good solution for UI state synchronization.

Experiment with the problem. In this section, we show the back button problem and show how Seaside makes it easy to handle. Perform the following experiment.

1. Browse the `WebCounter` application that we developed in the first chapter of this book.
2. Click on the ++ link to increase the value of the counter until the counter shows a value of 5.
3. Press the back button two times: you should see 3 now.
4. Click on the ++ link.

Your web browser does not show you 4 as you would expect, but instead displays 6. This is because the WebCounter component was not aware that you had pressed the back button. This situation can also arise if you open two windows that interact with the same application.

Solving the Problem. Seaside offers an elegant way to fix this problem. Define the method WAComponent>>states so that it returns an array that contains the objects that you want to keep in sync. In our WebCounter example we want to keep the count instance variable synchronized so we write the method as follows.

```
WebCounter>>states
    ^ Array with: count
```

This is not really what we want because the Seaside backtracking support is mostly intended for UI state and not model state. We want to backtrack the counter component, not the integer instance variable.

```
WebCounter>>states
    ^ Array with: self
```

Redo our back button experiment and you will see that after pressing the back button two times you can correctly increment the counter from 3 to 4.

9.9 Show/Answer Explained

This section explains the method show: in WAComponent. show: is a variation of call:. You may want to skip this section if you are new to Seaside. You will find it helpful later on if you need to have more control on how components replace each other.

The method show: passes the control from the receiving component to the component given as the first argument. As with call: the receiver will be temporarily replaced by aComponent. However, as opposed to call:, show: does not block the flow of control and immediately returns.

If we replace the call: in the method editContact: with show: the application does not behave the same way as before anymore:

```
ContactListView >> editContact: aContact
    | view |
    view := ContactView new.
    view contact: aContact.
    self show: view.
    self inform: 'Thanks for editing  ' , aContact name
```

The reason is that show: does not block and the confirmation is displayed immediately, effectively replacing the ContactView. Clicking on the *Ok* then reveals the ContactView. Of course this is not the intended behavior. We can fix this issue by assigning an answer handler to the view that displays the confirmation:

```
ContactListView >> editContact: aContact
    | view |
    view := ContactView new.
    view contact: aContact.
    view onAnswer: [ :answer |
        self inform: 'Thanks for editing  ' , aContact name ].
    self show: view
```

This solves our problem, but is arguably not very readable. Luckily there is `show:onAnswer:` that combines the two method calls:

```
ContactListView >> editContact: aContact
    | view |
    view := ContactView new.
    view contact: aContact.
    self show: view onAnswer: [ :answer |
        self inform: 'Thanks for editing  ' , aContact name ]
```

In fact, what we did above is continuation-passing style. Like this we can emulate the blocking behavior of `call:` by using `show:` and a block that defines what happens afterwards. Any code that uses `call:` can be transformed like this, however in case of loops that can become quite complicated (see).

9.10 Transforming a Call to a Show

Why is `show:` useful at all?

- First of all `show:` allows one to replace multiple components in one request. This is not possible with `call:` as it blocks the flow of execution and the developer has no possibility to display another component at the same time.
- Another reason to use `show:` is that it is more lightweight and that it uses fewer resources than `call:`. This means that if the blocking behavior is not needed, then `show:` is more memory friendly.
- Finally some Smalltalk dialects cannot implement `call:` due to limitations in their VM.

If you want or must get rid of the `call:` statements in a sequence of calls things are relatively simple. Transform code using `call:`

```
Task >> go
  | a b c |
  a := self call: A.
  b := self call: B.
  c := self call: C.
  ...
```

to the following using `show:onAnswer:`

```
Task >> go
  self show: A onAnswer: [ :a |
    self show: B onAnswer: [ :b |
      self show: C onAnswer: [ :c |
        ... ] ] ]
```

If you have a loop like the following one, things are slightly more complicated:

```
Task >> go
  [ self call: A ]
     whileTrue: [ self call: B ]
```

The example below shows an equivalent piece of code that uses recursion to implement the loop:

```
Task >> go
  self show: A onAnswer: [ :a |
    a ifTrue: [
      self show: B onAnswer: [ :b |
        self go ] ] ]
```

The transformation technique applied here is called *continuation-passing style* or short *CPS*. The `onAnswer:` block implements the continuation of the flow after the shown component answered. Unfortunately for more complicated flows CPS leads to messy code pretty quickly.

9.11 Summary

In this chapter we showed how to display component using the `call:` method. In the next chapter we will demonstrate how to embed components within each other.

CHAPTER 10

Embedding Components

Building reusable components and frameworks is the goal of all developers in almost all parts of their applications. The dearth of truly reusable (canned) component libraries for most of the existing web development frameworks is a good indication that this is difficult to do.

Seaside is among the few frameworks poised to change this. It has a solid component model giving one all of the mechanisms necessary to develop well encapsulated components and application development frameworks. We have seen in that components can be sequenced. In this chapter we show how to embed one component inside another component. In 10.6, we will see how to decorate a component to add functionality or change its appearance and as such reuse behavior. With a good component model, the possibility to display components and create new ones by reusing existing ones, writing Seaside applications is very similar to writing GUI applications.

We will start by writing an application which embeds one component, then refactor it into an application built out of two components. Finally we will discuss reuse and show how component decoration support this. We will show how components can be plugged together using events to maximize reuse.

10.1 Principle: Component Children

When we want to display several components on the same page, we can embed the components into each other. Usually a Seaside application consists of a main component which is usually called the root component. All the child components of the root component and their recursive children form a tree of nested components.

Child components are no different to other components or the root component. Note that the component tree of an application might change during the lifetime of a session. Through user interaction new components can be shown and old ones can be hidden.

Seaside requires that each component knows and declares all its visible child components using the method `WAComponent>>children`. This allows Seaside to know in advance what components will be visible when building the HTML and configure and trigger some event on these components before the actual rendering happens.

Note that Seaside does not require children to know their parents and the framework also does not provide this information. When instantiating the components such a link can be easily es-

tablished, but we do not suggest doing so as it would introduce strong coupling between the child component and its parent. For example, it would no longer be easily possible to use the same component in a different context.

Here are the steps that should be performed to embed components within another.

1. The parent component initializes the child components in the method `initialize`.
2. The parent component defines a method named `children` that returns all its direct child component instances, regardless of how and where they are store.
3. The parent component renders its child components in the method `renderContentOn:` using `html render: aComponent`.

10.2 Example: Embedding an Editor

We will build a new variation of our contact list manager. What we'd like to do is adapt our contact manager so that we see the item editor on the same page as the contact list item. That is, we want to embed the editor on the same page as the address list itself. While we could adapt the previous component to embed a component, we prefer to define a new component from scratch.

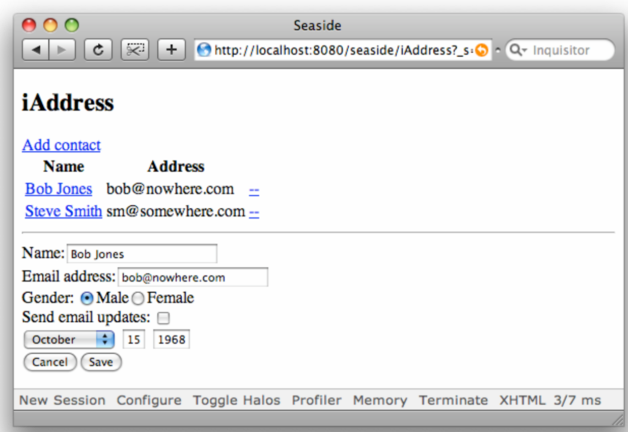


Figure 10-1 Embedding a ContactView into another component.

We already have a working editor component so let's just add it to a new `IAddress` component. That is, we're going to embed the `ContactView` component within the `IAddress` component.

Create the class. First we create the class of the new component `IAddress`.

```
WAComponent subclass: #IAddress
    instanceVariableNames: 'editor'
    classVariableNames: ''
    package: 'iAddress'
```

We add an instance variable, `editor`, to the class `IAddress` which is a reference to the editor that we will embed within our `IAddress` component. It is not always necessary to maintain a reference to an embedded component: we could also create the component on the fly (as soon as it is returned as part of the component' children). In our case, since the elements of our application are stateful objects, it is better to reuse components, taking advantage of the fact that they can store state, rather than recreating them. We will revisit this issue in **??**.

Initialize instances. The `initialize` method creates the editor and gives it a default contact to edit. We can then reuse the editor to edit other contacts, avoiding the need to create a new editor every time we want to edit something.

```
IAddress>>contacts
    ^ Contact contacts
```

```
IAddress >> initialize
    super initialize.
    editor := ContactView new.
    editor contact: self contacts first
```

Important **Specifying the component's children.** Any component that uses embedded children components should make Seaside aware of this by returning an array of those components. This is necessary because Seaside needs to be able to figure out what components are embedded within your component; Seaside needs to process all the callbacks for all of the components that may be displayed, before it starts any rendering of those components. Unless you add the children components to the parent's `children` method, the first Seaside will know about your children components is when you reference them in your rendering methods. **Specify the children.** Here is how to define the method `children` which returns an array containing the editor that is accessible via the instance variable `editor`.

```
IAddress >> children
    ^ Array with: editor
```

Specify some actions. Now define methods to create, add, remove and edit a contact.

```
IAddress >> addContact: aContact
    Contact addContact: aContact
```

```
IAddress >> askAndCreateContact
    | name emailAddress |
    name := self request: 'Name'.
    emailAddress := self request: 'Email address'.
    self addContact: (Contact name: name emailAddress: emailAddress)
```

```
IAddress >> editContact: aContact
    editor contact: aContact
```

```
IAddress >> removeContact: aContact
    (self confirm: 'Are you sure that you want to remove this
     contact?')
        ifTrue: [ Contact removeContact: aContact ]
```

Embedding a ContactView into another component with Halos.

Some rendering methods. We use a table to render the current contact list and let the user edit a contact by clicking on the name link.

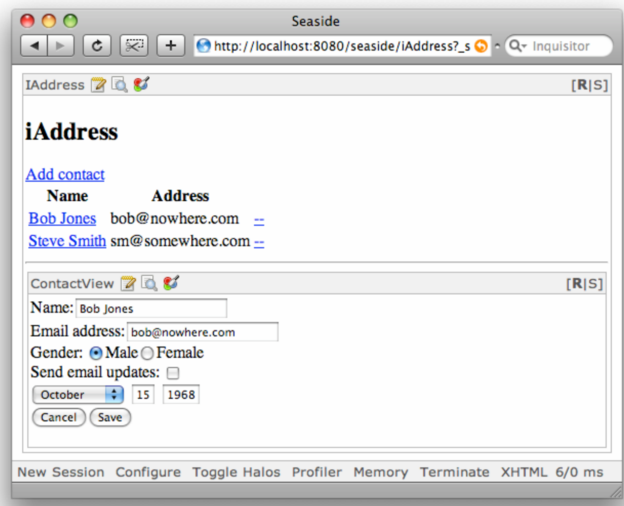


Figure 10-2 Embedding a ContactView into another component with Halos.

```
IAddress >> renderContentOn: html
    html form: [
        self renderTitleOn: html.
        self renderBarOn: html.
        html table: [
            html tableRow: [
                html tableHeading: 'Name'.
                html tableHeading: 'Address' ].
            self renderDatabaseRowsOn: html ].
        html horizontalRule.
        html render: editor ]
```

```
IAddress >> renderTitleOn: html
    html heading level: 2; with: 'iAddress'
```

```
IAddress >> renderBarOn: html
   html anchor
       callback: [ self askAndCreateContact ];
       with: 'Add contact'
```

```
IAddress >> renderDatabaseRowsOn: html
    self contacts do: [ :contact |
        html tableRow: [ self renderContact: contact on: html ] ]
```

```
IAddress >> renderContact: aContact on: html
    html tableData: [
        html anchor
            callback: [ self editContact: aContact ];
            with: aContact name].
    html tableData: aContact emailAddress.
    html tableData: [
        html anchor
            callback: [ self removeContact: aContact ];
            with: '--' ]
```

Register the application as ”iAddress” and try it out. Make sure that the editor is doing its job. Activate the halos. You’ll notice that there is a separate embedded halo around the editor component, see **??**. It is very helpful to inspect the state of a component in a running application (or view the rendered HTML.) Our simple implementation of `IAddress>>editContact:` will save changes even when you press *cancel*. See to understand how you can change this.

10.3 Components All The Way Down

The changes to your code in this section are presented purely to help you explore the embedding of components: they are not an example of good UI design, and are not required to progress with the following sections.

Let’s define a component that manages our list of contacts using components all the way down. Figure **??** shows that we will build our application out of two components: `PluggableContactListView` and `ContactView`. In addition, `PluggableContactListView` will be composed of several `ReadOnlyOneLinerContactView` components. We really get a tree of components. This exercise shows that a component should be designed to be pluggable. It also shows how to plug components together.

PluggableHalos

A Minimal Contact Viewer. We would like to have a compact contact viewer. First we will subclass the `ContactView` class to create the class `ReadOnlyOneLinerContactView`. This class has an instance variable `parent` which will hold a reference to the component that will contain it, since it should know how to invoke the contact editor.

```
ContactView subclass: #ReadOnlyOneLinerContactView
    instanceVariableNames: 'parent'
    classVariableNames: ''
    package: 'iAddress'
```

```
ReadOnlyOneLinerContactView >> parent: aParent
    parent := aParent
```

When the user clicks on the contact name, we want the associated user object to pass itself to the parent for editing. A similar action should occur when removing a contact from the database. Note that this component does not include a form. This is because only one form should be present on a page at any time, so a component is much more reusable if it does not define a form.

```
ReadOnlyOneLinerContactView >> renderContentOn: html
        html anchor
            callback: [parent editContact: self contact];
            with: self contact name.
        html space.
```

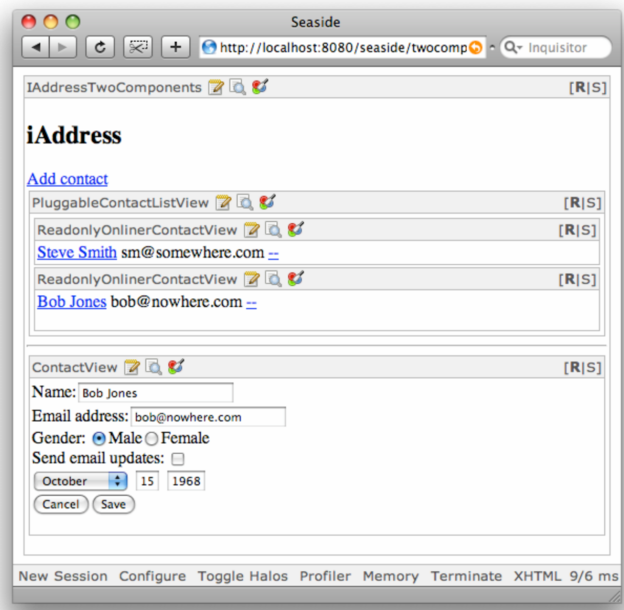


Figure 10-3 With components all the way down.

```
        html text: self contact emailAddress.
        html space.
        html anchor
            callback: [parent removeContact: self contact];
            with: '--'.
        html break.
```

Class Definition. Now we define the class `PluggableContactListView`. Since this component will embed all the contact viewer components, we add an instance variable `contactViewers`, that will refer to them. We also define an instance variable to refer to the editor that will show the detailed information of the currently selected contact.

```
ContactListView subclass: #PluggableContactListView
    instanceVariableNames: 'contactViewers editor'
    classVariableNames: ''
    package: 'iAddress'
```

We will use an identity dictionary to keep track of the contact viewer associated with each contact. Initializing our top level component consists of first creating the editor and then creating the viewers for the existing contacts.

```
PluggableContactListView >> contacts
    ^ Contact contacts
```

```
PluggableContactListView >> initialize
    super initialize.
    editor := ContactView new.
    contactViewers := IdentityDictionary new.
    self contacts do: [ :each | self addContactViewerFor: each ]
```

```
PluggableContactListView >> addContactViewerFor: aContact
    contactViewers
        at: aContact
        put: (ReadOnlyOneLinerContactView new
            contact: aContact;
            parent: self; " <-- added "
            yourself)
```

```
PluggableContactListView >> askAndCreateContact
     | name emailAddress |
     name := self request: 'Name'.
     emailAddress := self request: 'Email address'.
     self addContact: (Contact name: name emailAddress: emailAddress)
```

```
PluggableContactListView >> editor: anEditor
    editor := anEditor
```

Children accessing and rendering. We have now to specify that the contact viewers are embedded within the `PluggableContactListView` and how to render them.

```
PluggableContactListView >> children
    ^ contactViewers values
```

```
PluggableContactListView >> renderContentOn: html
    contactViewers values
        do: [ :each | html render: each. html break ]
```

We define a couple of methods to manage contacts.

```
PluggableContactListView >> addContact: aContact
    Contact addContact: aContact.
    self addContactViewerFor: aContact
```

```
PluggableContactListView>>removeContact: aContact
    contactViewers removeKey: aContact.
    Contact removeContact: aContact
```

Plugging everything together. Now we are ready to define a new version of `IAddress`. We simply subclass `IAddress` and pay attention to the fact that the list is now a component. So we initialize it, add it as part of the children of the component and render it.

```
IAddress subclass: #IAddressTwoComponents
    instanceVariableNames: 'list'
    classVariableNames: ''
    package: 'iAddress'
```

We pass the list component to the editor which has already been initialized in `IAddress`. We must also invoke the list's `askAndCreateContact` method, since it is the list that manages the creation of contacts. The rendering of the component includes a form in which the other components are embedded.

```
IAddressTwoComponents >> initialize
    super initialize.
    list := PluggableContactListView new.
    list editor: editor  " <-- added "
```

```
IAddressTwoComponents>>children
    ^ super children , (Array with: list)
```

```
IAddressTwoComponents>>renderBarOn: html
    html anchor
        callback: [ list askAndCreateContact ];
        with: 'Add contact'
```

```
IAddressTwoComponents >> renderContentOn: html
    html form: [
        self renderTitleOn: html.
        self renderBarOn: html.
        html break.
        html render: list.
        html horizontalRule.
        html render: editor ]
```

Note of course that embedding such an editor under the list of contacts is not a really good UI design. We just use it as a pretext to illustrate component embedding.

10.4 Intercepting a Subcomponent's Answer

Components may be designed to support both standalone and embedded use. Such components often produce answers (send `self answer:`) in response to user actions. When the component is standalone the answer is returned to the caller, but if the component is embedded the answer is ignored unless the parent component arranges to intercept it. In our example application the editor provides an answer when the user presses the "Save" button (i.e. in `ContactView>>save`) but this answer is ignored. It is easy to change our application to make use of this information; let's say we want to give the user confirmation that their data was saved. To accomplish this, change `IAddress>>initialize` and add the `WAComponent>>onAnswer:` behaviour:

```
IAddress >> initialize
    super initialize.
    editor := ContactView new.
    editor contact: self contacts first.
    editor onAnswer: [ :answer | self inform: 'Saved' ]   " <--
    added "
```

Now restart your application (press "New Session") and try it out. When you press the save button in the editor you should get a dialog tersely notifying you that your data is saved. Note that the component answer is passed into the block (although we didn't use it in this example).

The `onAnswer:` method is an important protocol for handling components and their answer.

10.5 A Word about Reuse

Suppose you wanted a component that shows only the name and email of our `ContactView` component. There are no special facilities in Seaside for doing this, so you may be tempted to

use template methods and specialize hooks in the subclasses. This may lead to the definition of empty methods in subclasses and may force you to define as many subclasses as situations you want to cover, for example if you want to create a `SimpleContactView` and a `ReadyonlyContactView`. See Figure 10-4 below.

An alternative approach is to build more advanced components using the messages `perform:` or `perform:with:` with a list of method selectors to be sent:

```
setOutlineForm
    "should be called during action phase"
    methods :=  #(renderNameOn: renderEmailOn:)
```

```
renderContentOn: html
   methods do: [ :each | self perform: each with: html ]
```

You can also define a component whose rendering depends on whether it is embedded. Here is an example where the rendering method does not wrap its content in a form tag when the component is in embedded mode (i.e., when it would expect its parent to have already created a form in which to embed this component). A better way of doing this would be to use a `FormDecorator` as shown in .

```
EmbeddableFormComponent>>renderContent: html body: aBlock
    self embedded
        ifTrue: [ aBlock value ]
        ifFalse: [ html form: aBlock ]
```

```
EmbeddableFormComponent>>renderContentOn: html
    self renderContent: html body: [
        "lots of form elements get rendered here"
        self renderSaveOn: html ]
```

This would then be embedded by another component using code like:

```
ContainingComponent >> renderContentOn: html
    html form: [
        "some form elements here, followed by our embeddable
     Component:"
        html render: (EmbeddableFormComponent new beEmbedded;
     yourself) ]
```

If you need more sophisticated dynamic control over the rendering of your component, you may want to use *Magritte* with Seaside. Magritte is a framework which allows you to define *descriptions* for your domain objects. It then uses these descriptions to perform automatic actions (loading, saving, generating SQL...). The Magritte/Seaside integration allows one to automatically generate forms and Seaside components from domain object described with Magritte descriptions, see . Magritte also offers ways to construct different views on the same objects and so the possibility to create multiple varieties of components: either by selecting a subset of fields to display, or by offering read-only or editable components. As such, it is an extremely useful addition to plain Seaside.

10.6 Decorations

Sometimes we would like to reuse a component while adding something to it, such as an information message or extra buttons. Seaside has facilities for doing this. In Seaside parlance, this is called “decorating” a component. Note that this is *not* implemented using the design pattern of the same name, but rather as a *Chain of Responsibility*. This means that decorations form a chain

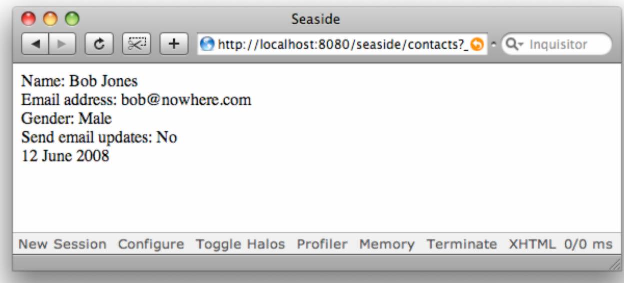


Figure 10-4 A readonly view of the ContactView.

of special components into which your component is inserted, and that a given message pass through the chain of decorators.

Decorations can be added to any component by calling `WAComponent>>addDecoration:`. Decorations are used to change the behavior or the look of the decorated component.

A component decoration is static in the sense that it should not change after the component has been rendered. Thus, a decoration should be attached to a component either just after it (the decorator) is created, or just before the component is passed as argument of a `call:` message.

```
self call: (aComponent
    addDecoration: aDecoration;
    yourself)
```

There are three kinds of decorations:

- **Visual Decorations.** These change a visual aspect of the decorated component: `WAMessageDecoration` renders a heading above the component; `WAFormDecoration` renders a form with buttons around the component; and `WAWindowDecoration` renders a border with a close widget around the component.
- **Behavioral Decorations.** These allow you to add some common behaviours to your components: `WAValidationDecoration` allows you to add validation of the answer-argument and the display of an error message.
- **Internal Decorations.** These support internal logic that you will use when building complex applications: `WADelegation` is used to implement the `call:` message; `WAAnswerHandler` is used to handle the `answer:` message.

Visual Decorations

Message Decorations. `WAMessageDecoration` renders a heading above the component using the message `WAComponent>>addMessage:`. As an example we add a message to the `ContactView` component by sending it `addMessage:`, see 10-5.

```
IAddress >> initialize
    super initialize.
    editor := ContactView new.
    editor contact: self contacts first.
```

```
    editor addMessage: 'Editing...'.  " <-- added "
    editor onAnswer: [ :answer | self inform: 'Saved', answer
    printString ]
```

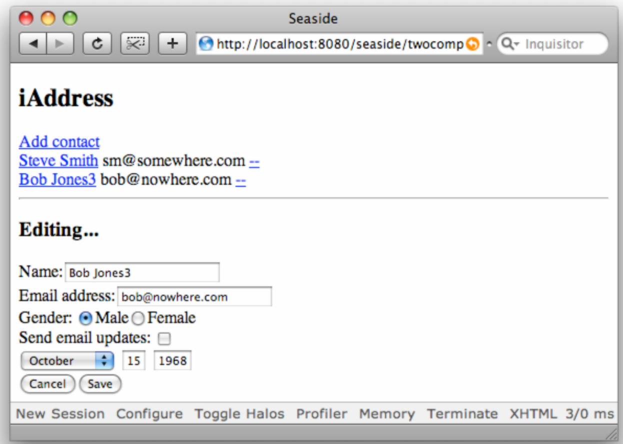


Figure 10-5 Adding a message around a component.

Note that the `WAComponent>>addMessage:` returns the decoration, so you may want to also use the `yourself` message if you need to refer to the component itself:

```
SomeComponent >> renderContentOn: html
    html render: (AnotherComponent new addMessage: 'Another
    Component'; yourself)
```

Window Decoration. `WAWindowDecoration` renders a border with a close widget around the component.

The following example adds a window decoration to the `ContactView` component. To see it in action, use the contacts application implemented by the `ContactList` component (probably at http://localhost:8080/contacts. The result of clicking on an edit link is shown in 10-6.

```
ContactView >> initialize
    super initialize.
    self addDecoration: (WAFormDecoration new buttons: #(cancel
    save)).
    self addDecoration: (WAWindowDecoration new title: 'Zork
    Contacts')
```

You may see that your *Save* and *Cancel* buttons are duplicated: you can remove this duplication by commenting out the `self renderSaveOn: html` line from `ContactView>>renderContentOn:`. It is much more common to add a window decoration when calling a component rather than when initializing it. The following example illustrates a common idiom that Seaside programmers use to decorate a component when calling it. It uses a decoration to open a component on a new page.

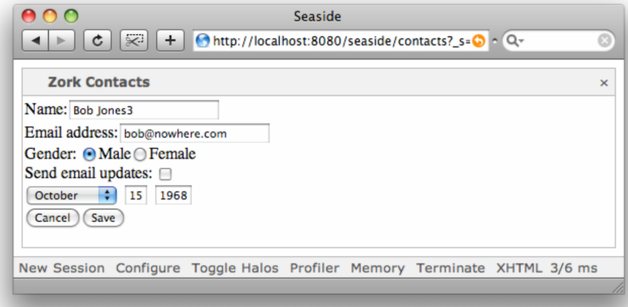


Figure 10-6 Decorating a component with a window.

```
WAPlugin >> open: aComponent
    "Replace the current page with aComponent."

    WARenderLoop new
        call: (aComponent
            addDecoration: (WAWindowDecoration new
                cssClass: self cssClass;
                title: self label;
                yourself);
            yourself)
        withToolFrame: false
```

Form Decoration. A `WAFormDecoration` places its component inside an HTML form tag. The message `WAFormDecoration>>buttons:` should be used to specify the buttons of the form. The button specification is a list of strings or symbols where each string/symbol is the label (first letter capitalized) for a button and the name of the component callback method when button is pressed.

The component that a `WAFormDecoration` decorates must implement the method `WAFormDecoration>>defaultButton`, which returns the string/symbol of the default button (the one selected by default) of the form. For each string/symbol specified by the `WAFormDecoration>>buttons:` method, the decorated component must implement a method of the same name, which is called when the button is selected. Important Be sure not to place any decorators between `WAFormDecoration` and its component, otherwise the `defaultButton` message may fail. You can examine the source of `WAFormDialog` and its subclasses to see the use of a FormDecoration to manage buttons:

```
WAFormDialog >> addForm
    form := WAFormDecoration new buttons: self buttons.
    self addDecoration: form
```

```
WAFormDialog >> buttons
    ^ #(ok)
```

Using Decorations in the Contacts application. You can add a `WAFormDecoration` to `ContactView` as follows: define an `initialize` method to add the decoration, and remove the su-

perfluous rendering calls from `renderContentOn:`, to leave simpler code and an unchanged application (see 10-7).

```
ContactView>>initialize
    super initialize.
    self addDecoration: (WAFormDecoration new buttons: #(cancel
    save))
```

```
ContactView>>renderContentOn: html
    self renderNameOn: html.
    self renderEmailOn: html.
    self renderGenderOn: html.
    self renderSendUpdatesOn: html.
    self renderDateOn: html.
```

We chose `cancel` and `save` as our button names since these methods were already defined in the class, but we could have used any names we wanted as long as we implemented the corresponding methods.

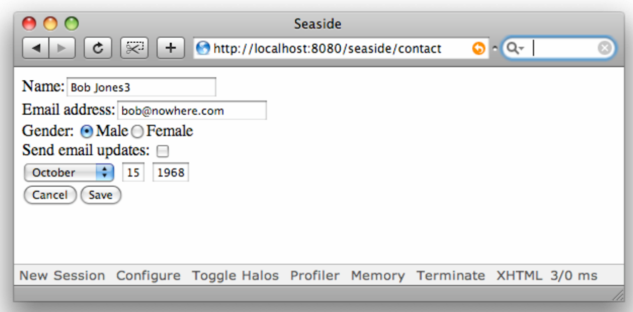


Figure 10-7 Using a decoration to add buttons and form to a ContactView.

Behavioral Decorators

Validation. A `WAValidationDecoration` validates its component form data when the component returns using `WAComponent>>answer` or `WAComponent>>answer:`. This decoration can be added to a component via the method `WAValidationDecoration>>validateWith:` as shown below.

```
SampleLoginComponent >> initialize
    super initialize.
    form := WAFormDecoration new buttons: self buttons.
    self addDecoration: form.
    self validateWith: [ :answer | answer validate ].
    self addMessage: 'Please enter the following information'.
```

If the component returns via `answer:`, the `answer:` argument is passed to the validate block. If the component returns using `answer` the sender of `answer` is passed to the validate block. Talk about `WADelegation` and `WAAnswerHandler`. **Accessing the component.** To access the

component when you have only a reference to its decoration you can use the message `WADecoration>>decoratedComponent`.

10.7 Component Coupling

Here is an interesting question that often comes up when writing components, and one which we faced when embedding our components: “How do the components communicate with each other in a way that doesn’t bind them together explicitly?” That is, how does a child component send a message to its parent component without explicitly knowing who the parent is? Designing a component to refer to its parent (as we did) is not always an ideal solution, since the interfaces of different parents may be different, and this would prevent the component from being reused in different contexts.

Another approach is to adopt a solution based on explicit dependencies, also called the *change/update mechanism*. Since the early days of Smalltalk, it has provided a built-in dependency mechanism based on a change/update protocol–this mechanism is the foundation of the MVC framework itself. A component registers its interest in some event and that event triggers a notification.

Announcements. Perhaps the most flexible and powerful approach is to use a more recent framework called *Announcement*. While the original dependency framework relied on symbols for the event registration and notification, *Announcement* promotes an object-oriented solution; i.e. events are standard objects. Originally developed by Vassili Bykov, this framework has been ported to several Smalltalk implementations, and is popular with Seaside developers.

The main idea behind the framework is to set up common announcers and to let clients register to send or receive notifications of events. An *event* is an object representing an occurrence of a specific event. It is the place to define all the information related to the event occurrence. An *announcer* is responsible for registering interested clients and announcing events. In the context of Seaside, we can define an announcer in a session. For more information on sessions see .

An Example. Here is an example taken from Ramon Leon’s very good Smalltalk blog (at http://onsmalltalk.com/). This example shows how we can use announcements to manage the communication between a parent component and its children as for example in the context of a menu and its menu items.

First we add a reference to a new `Announcer` to our session:

```
MySession >> announcer
    ^ announcer ifNil: [ announcer := Announcer new ]
```

Second a subclass of an `Announcement` is created for each event of interest, here child removal:

```
Announcement subclass: #RemoveChild
    instanceVariableNames: 'child'
    classVariableNames: ''
    package: 'iAddress'
```

Each subclass can have additional instance variables and accessors added to hold any extra information about the specific announcement such as a context, the objects involved etc. This is why announcement objects are both more powerful and simpler than using symbols.

```
RemoveChild class >> child: aChild
    ^ self new
        child: aChild;
        yourself
```

```
RemoveChild >> child: anChild
    child := anChild
```

```
RemoveChild >> child
    ^ child
```

Any component interested in this announcement registers its interest by sending the announcer the message `on: anAnnouncementClass do: aBlock` or `on: anAnnouncementClass send: aSelector to: anObject`. You can also ask an announcer to `unsubcribe:` an object. The messages `on:do:` and `on:send:to:` are strictly equivalent to the messages `subscribe: anAnnouncementClass do: aValuable` (an object understanding `value`) and `subscribe: anAnnouncementClass send: aSelector to: anObject`. In the following example, when a parent component is created, it expresses interest in the `RemoveChild` event and specifies the action that it will perform when such an event happens.

```
Parent >> initialize
    super initialize.
    self session announcer on: RemoveChild do: [:it | self
     removeChild: it child]
```

```
Parent >> removeChild: aChild
    self children remove: aChild
```

And any component that wants to fire this event simply announces it by sending in an instance of that custom announcement object:

```
Child >> removeMe
    self session announcer announce: (RemoveChild child: self)
```

advanced Note that depending on where you place the announcer, you can even have different sessions sending events to each other, or different applications. **Pros and cons.** Announcements are not always the best way to establish communication between components and you have to decide the exact design you want. On one hand, announcements let you create loosely coupled components and thus maximize reusability. On the other hand, they introduce additional complexity when you may be able solve your communication problem with a simple message send.

10.8 Summary

In this chapter we have seen how to embed components to build up complex functionality. In particular, we have learned:

- To embed a component in another one, the parent component should just answer the component as one of its children. Its `children` method should return the direct children components.
- Each component may render its immediate children in its own render method by calling various methods and possibly the `render:` method.
- A component may be reused with decorations. Decorations are components which add visual aspects or change component behavior.

CHAPTER 11

Tasks

In Seaside, it is possible to define components whose responsibility is to represent the flow of control between existing components. These components are called tasks. In this chapter, we explain how you can define a task. We also show how Seaside supports application control flow by isolating certain paths from others. We will start by presenting a little game, the number guessing game. Then, we will implement two small hotel registration applications using the calendar component to illustrate tasks.

11.1 Sequencing Components

Tasks are used to encapsulate a process or control flow. They do not directly render XHTML, but may do so via the components that they call. Tasks are defined as subclasses of `WATask`, which implements the key method `WATask>>go`, which is invoked as soon as a task is displayed and can call other components.

Let's start by building our first example: a number guessing game (which was one of the first Seaside tutorials). In this game, the computer selects a random number between 1 and 100 and then proceeds to repeatedly prompt the user for a guess. The computer reports whether the guess is too high or too low. The game ends when the user guesses the number. Those of you who remember learning to program in BASIC will recognise this as one of the common exercises to demonstrate simple user interaction. As you will see below, in Seaside it remains a simple exercise, despite the addition of the web layer. This comes as a stark contrast to other web development frameworks, which would require pages of boilerplate code to deliver such straightforward functionality. We create a subclass of `WATask` and implement the `go` method:

```
WATask subclass: #GuessingGameTask
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'SeasideBook'
```

```
GuessingGameTask >> go
    | number guess |
    number := 100 atRandom.
        [ guess := (self request: 'Enter your guess') asNumber.
        guess < number
```

```
            ifTrue: [ self inform: 'Your guess was too low' ].
        guess > number
            ifTrue: [ self inform: 'Your guess was too high' ].
        guess = number ] whileFalse.
    self inform: 'You got it!'
```

The method go randomly draws a number. Then, it asks the user to guess a number and gives feedback depending on the input number. The methods `request:` and `inform:` create components (`WAInputDialog` and `WAFormDialog`) on the fly, which are then displayed by the task. Note that unlike the components we've developed previously, this class has no `renderContentOn:` method, just the method go. Its purpose is to drive the user through a sequence of steps.

Register the application (as 'guessinggame') and give it a go. Figure 11-1 shows a typical execution.

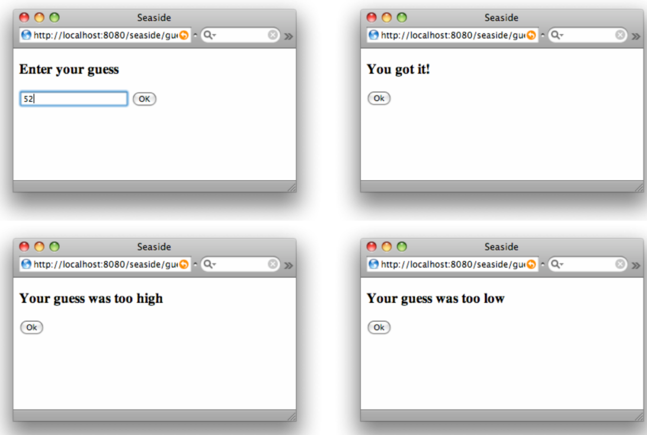


Figure 11-1 Guessing Game interaction.

Why not try modifying the game to count the number of guesses that were needed?

This example demonstrates that with Seaside you can use plain Smalltalk code (conditionals, loops, etc.,) to define the control flow of your application. You do not have to use yet another language or build a scary XML state-machine, as required in other frameworks. In some sense, tasks are simply components that start their life in a callback.

Because tasks are indeed components (`WATask` is a subclass of `WAComponent`), all of the facilities available to components, such as `call:` and `answer:` messages, are available to tasks as well. This allows you to combine components and tasks, so your `LoginUserComponent` can call a `RegisterNewUserTask`, and so on. Important Tasks do not render themselves. Don't override `renderContentOn:` in your tasks. Their purpose is simply to sequence through other views.Important If you are reusing components in a task – that is, you store them in instance variables instead of creating new instances in the go method – be sure to return these instances in the #children method so that they are backtracked properly and you get the correct control flow.

11.2 Hotel Reservation: Tasks vs. Components

To compare when to use a task or a component, let's build a minimal hotel reservation application using a task and a component with children. Using a task, it is easy to reuse components and build a flow. Here is a small application that illustrates how to do this. We want to ask the user to specify starting and ending reservation dates. We will define a new subclass of `WATask` with two instance variables `startDate` and `endDate` of the selected period.

```
WATask subclass: #HotelTask
   instanceVariableNames: 'startDate endDate'
   classVariableNames: ''
   package: 'Calendars'
```

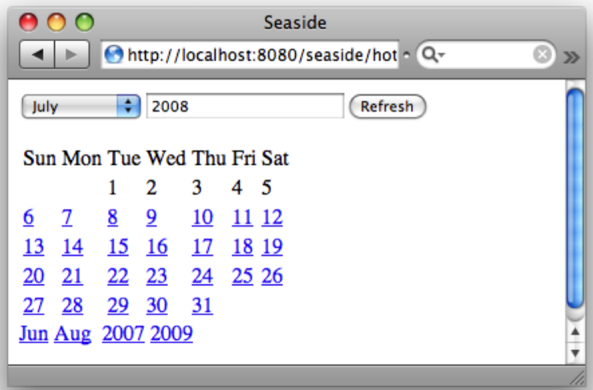
We define a method go that will first create a calendar with selectable dates after today, then create a second calendar with selectable days after the one selected during the first interaction, and finally we will display the dates selected as shown in 11-2.

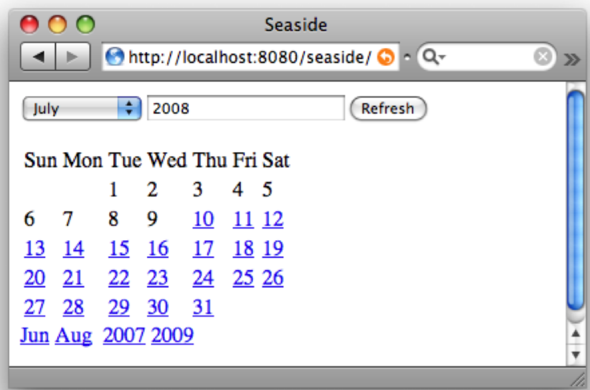
```
HotelTask >> go
    startDate := self call: (WAMiniCalendar new
        canSelectBlock: [ :date | date > Date today ]).
    endDate := self call: (WAMiniCalendar new
        canSelectBlock: [ :date | startDate isNil or: [ startDate <
    date ] ]).
    self inform: 'from ' , startDate asString , ' to ' ,
        endDate asString , ' ' , (endDate - startDate) days asString
    ,
        ' days'
```

Note that you could add a confirmation step and loop until the user is OK with his reservation.

Now this solution is not satisfying because the user cannot see both calendars while making his selection. Since we can't render components in our task, it's not easy to remedy the situation. We could use the message `addMessage: aString` to add a message to a component but this is still not good enough. This example demonstrates that tasks are about flow and not about presentation.

```
HotelTask >> go
    startDate := self call: (WAMiniCalendar new
        canSelectBlock: [ :date | date > Date today ];
        addMessage: 'Select your starting date';
        yourself).
    endDate := self call: (WAMiniCalendar new
        canSelectBlock: [ :date | startDate isNil or: [ startDate <
    date ] ];
        addMessage: 'Select your leaving date';
        yourself).
    self inform: (endDate - startDate) days asString , ' days: from
    ' ,
        startDate asString , ' to ' , endDate asString , ' '
```





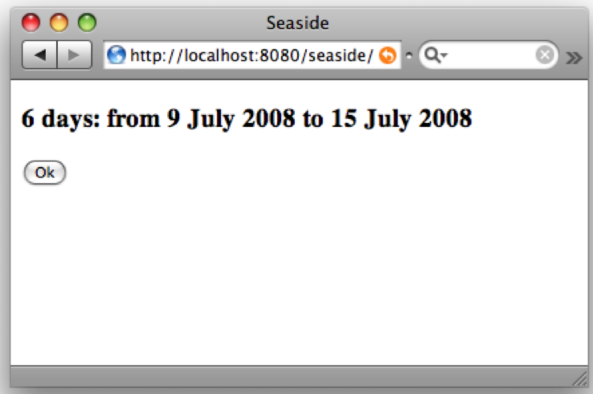


Figure 11-2 A simple reservation based on task.

11.3 Mini Inn: Embedding Components

Let's solve the same problem using component embedding. We define a component with two calendars and two dates. The idea is that we want to always have the two mini-calendars visible on the same page and provide some feedback to the user as shown by 11-3.

```
WAComponent subclass: #MiniInn
    instanceVariableNames: 'calendar1 calendar2 startDate endDate'
    classVariableNames: ''
    package: 'Calendars'
```

Since we want to show the two calendars on the same page we return them as children.

```
MiniInn >> children
    ^ Array with: calendar1 with: calendar2
```

We initialize the calendars and make sure that we store the results of their answers.

```
MiniInn >> initialize
    super initialize.
    calendar1 := WAMiniCalendar new.
    calendar1
        canSelectBlock: [ :date | Date today < date ];
        onAnswer: [ :date | startDate := date ].
    calendar2 := WAMiniCalendar new.
    calendar2
        canSelectBlock: [ :date | startDate isNil or: [ startDate <
    date ] ];
        onAnswer: [ :date | endDate := date ]
```

Finally, we render the application, and this time we can provide some simple feedback to the user. The feedback is simple but this is just to illustrate our point.

```
MiniInn >> renderContentOn: html
    html heading: 'Starting date'.
    html render: calendar1.
    startDate isNil
        ifFalse: [ html text: 'Selected start: ' , startDate
    asString ].
    html heading: 'Ending date'.
    html render: calendar2.
    (startDate isNil not and: [ endDate isNil not ]) ifTrue: [
        html text: (endDate - startDate) days asString ,
           ' days from ' , startDate asString , ' to ' ,
            endDate asString , ' ' ]
```

11.4 Summary

In this chapter, we presented tasks, subclasses of Task. Tasks are components that do not render themselves but are used to build application flow based on the composition of other components. We saw that the composition is expressed in plain Pharo.

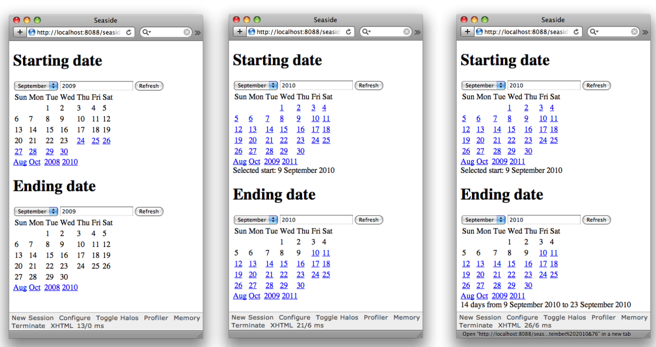


Figure 11-3 A simple reservation with feedback.

CHAPTER 12

Writing good Seaside Code

This short chapter explains how you can improve the quality of your code and your programming skills in general by running *Slime*, a Seaside-specific code critics tool. For example, Slime can detect if you do not follow the canonical forms for brush usage that we presented in Chapter . It will also help you identify other potential bugs early on, and help you produce better code. Furthermore, if you work on a Seaside library, it is able to point out portability issues between the different Smalltalk dialects.

12.1 A Seaside Program Checker

Slime analyzes your Seaside code and reveals potential problems. Slime is an extension of *Code Critics* that is shipped with the *Refactoring Browser*. Code Critics, also called *SmallLint*, is available with the Refactoring Browser originally developed by John Brant and Don Roberts. Lukas Renggli and the Seaside community extended Code Critics to check common errors or bad style in Seaside code. The refactoring tools and Slime are available in the One-Click Image and we encourage you to use them to improve your code quality.

Pay attention that the rules are not bulletproof and by no means complete. It could well be that you encounter false positives or places where it misses some serious problems, but it should give you an idea where your code might need some further investigation.

Here are some of the problems that Slime detects:

Possible Bugs. This group of rules detects severe problems that are most certainly serious bugs in the source code:

- The message `with:` is not last in the cascade,
- Instantiates new component while generating HTML,
- Manually invokes `renderContentOn:`,
- Uses the wrong output stream,
- Misses call to super implementation,
- Calls functionality not available while generating output, and
- Calls functionality not available within a framework callback.

Bad style. These rules detect some less severe problems that might pose maintainability problems in the future but that do not cause immediate bugs.

- Extract callback code to separate method,
- Use of deprecated API, and
- Non-standard object initialization.

Suboptimal Code. This set of rules suggests optimization that can be applied to code without changing its behavior.

- Unnecessary block passed to brush.

Non-Portable Code. While this set of rules is less important for application code, it is central to the Seaside code base itself. The framework runs without modification on many different Smalltalk platforms, which differ in the syntax and the libraries they support. To avoid that contributors from a specific platform accidentally submit code that only works with their platform we've added some rules that check for compatibility. The rules in this category include:

- Invalid object initialization,
- Uses curly brace arrays,
- Uses literal byte arrays,
- Uses method annotations,
- Uses non-portable class,
- Uses non-portable message,
- ANSI booleans,
- ANSI collections,
- ANSI conditionals,
- ANSI convertor,
- ANSI exceptions, and
- ANSI streams.

12.2 Summary

In this chapter, we presented tasks, subclasses of Task. Tasks are components that do not render themselves but are used to build application flow based on the composition of other components. We saw that the composition is expressed in plain Pharo.

Part IV

Seaside in Action

CHAPTER 13

A Simple ToDo Application

The objective of this chapter is to highlight the important issues when building a Seaside application: defining a model, defining a component, rendering the component, adding callbacks, and calling other components. This chapter will repeat some elements already presented before but within the context of a little application. It is a kind of summary of the previous points.

13.1 Defining a Model

It is a good software engineering practice to clearly separate the domain from its views. This is a common practice which allows one to change the rendering or even the rendering framework without having to deal with the internal aspects of the model. Thus, we will begin by presenting a simple model for a todo list that contains todo items as shown by Figure 13-1.

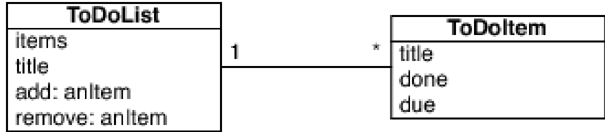


Figure 13-1 A simple model with items and an item container.

`ToDoItem` class. A todo item is characterized by a title, a due date and a status which indicates whether the item is done.

```
Object subclass: #ToDoItem
    instanceVariableNames: 'title due done'
    classVariableNames: ''
    package: 'ToDo-Model'
```

It has accessor methods for the instance variables `title`, `due` and `done`.

```
ToDoitem >> title
    ^ title
```

```
ToDoitem >> title: aString
    title := aString
```

```
ToDoItem >> due
    ^ due
```

```
ToDoItem >> due: aDate
    due := aDate asDate
```

```
ToDoItem >> done
    ^ done
```

```
ToDoItem >> done: aBoolean
    done := aBoolean
```

We specify the default values when a new todo item is created by defining a method `initialize` as follows:

```
ToDoItem >> initialize
    self title: 'ToDo Item'.
    self due: Date tomorrow.
    self done: false.
```

A word about `initialize` and `new`. Pharo and Squeak are the only Smalltalk dialects that perform automatic object initialization. This greatly simplifies the definition of classes. If you have defined an `initialize` method, it will be automatically called when you send the message `new` to your classes. In addition, the instance method `initialize` is defined in the class `Object` so you can (and are encouraged) to invoke potential `initialize` methods of your superclasses using `super initialize` in your own `initialize` method. If you want to write code that is portable between dialects, you should redefine the method `new` in all your root classes (subclasses of `Object`) as shown below and you should **not** invoke `initialize` via a super call in your root classes.

```
ToDoItem class >> new
    ^ self basicNew initialize
```

In this book we follow this convention and this is why we have not added `super initialize` in the methods `ToDoItem>>initialize` and `ToDoList>>initialize`.

We also add two testing methods to our todo item:

```
ToDoItem >> isDone
    ^ self done
```

```
ToDoItem >> isOverdue
    ^ self isDone not and: [ Date today > self due ]
```

`ToDoList` Class. We now create a class that will hold a list of todo items. The instance variables will contain a title and a list of items. In addition, we define a *class variable* `Default` that will refer to a singleton of our class.

```
Object subclass: #ToDoList
   instanceVariableNames: 'title items'
    classVariableNames: 'Default'
    package: 'ToDo-Model'
```

You should next add the associated accessor methods `title`, `title:`, `items` and `items:`.

The instance variable `items` is initialized with an `OrderedCollection` in the `initialize` method:

```
ToDoList >> initialize
   self items: OrderedCollection new
```

We define two methods to add and remove items.

```
ToDoList >> add: aTodoItem
    self items add: aTodoItem
```

```
ToDoList >> remove: aTodoItem
   ^ self items remove: aTodoItem
```

Now we define the *class-side* method `default` that implements a lazy initialization of the singleton, initializes it with some examples and returns it. The class-side method `reset` will reset the singleton if necessary.

```
ToDoList class >> default
    ^ Default ifNil: [ Default := self new ]
```

```
ToDoList class >> reset
   Default := nil
```

Finally, we define a method to add some todo items to our application so that we have some items to work with.

```
ToDoList class >> initializeExamples
    "self initializeExamples"

    self default
        title: 'Seaside ToDo';
        add: (ToDoItem new
            title: 'Finish todo app chapter';
            due: '11/15/2007' asDate;
            done: false);
        add: (ToDoItem new
            title: 'Annotate first chapter';
            due: '04/21/2008' asDate;
            done: true);
         add: (ToDoItem new
            title: 'Watch out for UNIX Millenium bug';
            due: '01/19/2038' asDate;
            done: false)
```

Now evaluate this method (by selecting the `self initializeExamples` text and selecting `do it` from the context menu). This will populate our model with some default todo items.

Now we are ready to define our seaside application using this model.

13.2 Defining a View

First, we define a component to see the item list. For that, we define a new component named `ToDoListView`.

```
WAComponent subclass: #ToDoListView
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'ToDo-View'
```

We can register the application by defining the class method initialize as shown and by executing `ToDoListView>>initialize`.

```
ToDoListView class >> initialize
    "self initialize"
    WAAdmin register: self asApplicationAt: 'todo'
```

You can see that the todo application is now registered by pointing your browser to http://localhost:8080/config/ as shown in 13-2.

Figure 13-2 The application is registered in Seaside.

If you click on the todo link in the config list you will get an empty browser window. This is to be expected since so far the application does not do any rendering. Now if you click on the halo you should see that your application is present on the page as shown in 13-3.

Figure 13-3 Our application is there, but nothing is rendered.

Now we are ready to work on the rendering of our component.

13.3 Rendering and Brushes

We define the method `model` to access the singleton of `ToDoList` as follows.

```
ToDoListView >> model
    ^ ToDoList default
```

A word about design. Note that directly accessing a singleton instead of using an instance variable is definitively not a good design since it favors procedural-like global access over encapsulation and distribution of knowledge. Here we use it because we want to produce a running application quickly. The singleton design pattern looks trivial but it is often misunderstood: it should be used when you want to ensure that there is never more than one instance; it does **not** limit *access* to one instance at a time. In general, if you can avoid a singleton by adding an instance variable to an object, then you do not need the singleton. The method `renderContentOn:` is called by Seaside to render a component. We will now begin to implement this method. First we just display the title of our todo list by defining the method as follows:

```
ToDoListView >> renderContentOn: html
   html text: self model title
```

If you refresh your browser you should obtain 13-4.

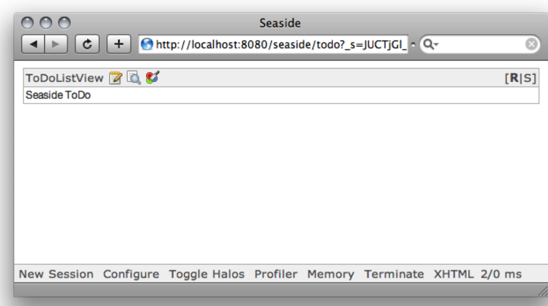


Figure 13-4 Our todo application simply displaying its title.

Now we will make some changes that will help us render the list and its elements. We will define a CSS style so we redefine the method `renderContentOn:` to use the brush `heading`.

```
ToDoListView >> renderContentOn: html
    html heading: self model title
```

Refresh your browser to see that you did not change much, except that you will get a bigger title. To render a list of items we define a method `renderItemsOn:` that we will invoke from `renderContentOn:`. To render an individual item we define a method called `renderItem:on:`.

```
ToDoListView >> renderItem: anItem on: html
    html listItem
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: anItem title
```

```
ToDoListView >> renderItemsOn: html
    self model items
        do: [ :each | self renderItem: each on: html ]
```

```
ToDoListView >> renderContentOn: html
    html heading: self model title.
    html unorderedList: [ self renderItemsOn: html ]
```

As you see, we are rendering the todo items as an unordered list. We also conditionally assign CSS classes to each list item, depending on its state. To do this, we will use the handy method `class:if:` since it allows us to write the condition and the class name in the cascade of the brush. Each item will get a class that indicates whether it is completed or overdue. The CSS will cause each item to be displayed with a color determined by its class. Because we haven't defined any CSS yet, if you refresh your browser now, you will see the plain list.

Next, we edit the style of this component either by clicking on the halos and the pencil and editing the style directly, or by defining the method `style` on the class `ToDoListView` in your code browser. Check Chapter 6 to learn more about the use of style-sheets and CSS classes.

```
ToDoListView >> style
    ^ 'body {
    color: #222;
    font-size: 75%;
    font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}
h1 {
    color: #111;
    font-size: 2em;
    font-weight: normal;
    margin-bottom: 0.5em;
}
ul {
    list-style: none;
    padding-left: 0;
    margin-bottom: 1em;
}
li.overdue {
    color: #8a1f11;
}
li.done {
    color: #264409;
}'
```

Refresh your browser and you should see the list of items and the todo list title as shown in 13-5.

13.4 Adding Callbacks

As we saw in Chapter 7, Seaside offers a powerful way to define a user action: *callbacks*. We can use callbacks to make our items editable. We will extend the method `renderItem:on:` with *edit* and *remove* actions. To do this, we render two additional links with every item.

Figure 13-5 Our todo application, displaying its title and a list of its items colored according to status.

```
ToDoListView >> renderItem: anItem on: html
    html listItem
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: [
            html text: anItem title.
            html space.
            html anchor
                callback: [ self edit: anItem ];
                with: 'edit'.
            html space.
            html anchor
                callback: [ self remove: anItem ];
                with: 'remove' ]
```

We use an `anchor` brush and we attach a callback to the anchor. Thus, the methods defined below are invoked when the user clicks on an anchor. Note that we haven't implemented the edit action yet. For now, we just display the item title to see that everything is working. The remove action is fully implemented.

```
ToDoListView >> edit: anItem
    self inform: anItem title
```

```
ToDoListView >> remove: anItem
    (self confirm: 'Are you sure you want to remove ' , anItem title
    printString , '?')
        ifTrue: [ self model remove: anItem ]
```

You should now be able to click on the links attached to an item to invoke the `edit` and `remove` methods as shown in 13-6.

You can have a look at the generated XHTML code by turning on the halos and selecting the *source* link. You will see that Seaside is automatically adding lots of information to the links on the page. This is part of the magic of Seaside which frees you from the need to do complex request parsing and figure out what context you were in when defining the callback.

Now it would be good to allow users to add a new item. The following code will just add a new

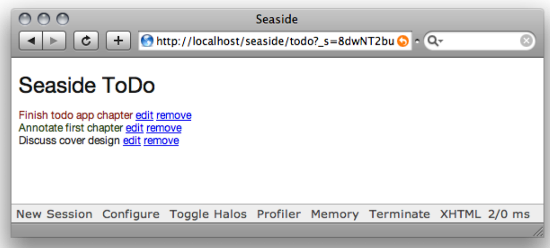


Figure 13-6 Our todo application with anchors.

anchor under the title (see 13-7):

```
ToDoListView >> renderContentOn: html
    html heading: self model title.
    html anchor
        callback: [ self add ];
        with: 'add'.
    html unorderedList: [ self renderItemsOn: html ]
```

For now, we will define a basic version of the addition behavior by simply defining add as the addition of the new item in the list of items. Later on we will open an editor to let the user define new todo items in place.

```
ToDoListView >> add
    self model add: ToDoItem new
```

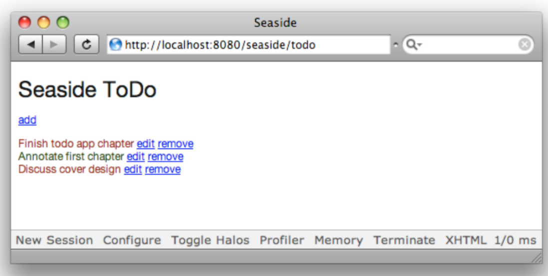


Figure 13-7 Our todo application with add functionality.

13.5 Adding a Form

We would like to have a *Save* button so that we can save our changes. We need to wrap our component in a form in order for this to work correctly (see). Here is our updated renderCon-

tentOn: method:

```
ToDoListView >> renderContentOn: html
   html heading: self model title.
    html form: [
        html anchor
             callback: [ self add ];
             with: 'add'.
        html unorderedList: [ self renderItemsOn: html ].
        html submitButton: 'Save' ]
```

Now we can add a checkbox to change the status of a todo item, see 13-8.

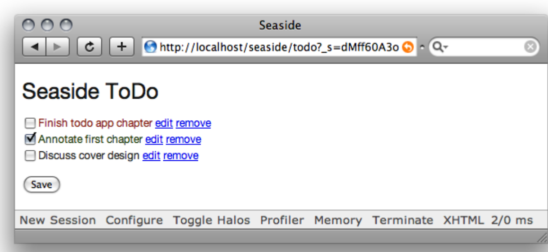


Figure 13-8 Our todo application with checkboxes and save buttons.

Note that the value of the checkbox is passed as an argument of the checkbox callback. The callback uses this value to change the status of the todo item. Notice the use of the submitButton to add a submit button in the form.

```
ToDoListView >> renderItem: anItem on: html
    html listItem
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: [
            html checkbox
                value: anItem done;
                callback: [ :value | anItem done: value ].
            html render: anItem title.
            html space.
            html anchor
                callback: [ self edit: anItem ];
                with: 'edit'.
            html space.
            html anchor
                callback: [ self remove: anItem ];
                with: 'remove' ]
```

13.6 Calling other Components

We are ready to create another component and call it. We create a component called ToDoItemView that is used to represent a specific todo item. Let's create a new class that will refer to the item it represents via an instance variable named model.

```
WAComponent subclass: #ToDoItemView
    instanceVariableNames: 'model'
    classVariableNames: ''
    package: 'ToDo-View'
```

We define the corresponding accessor methods.

```
ToDoItemView >> model
    ^ model
```

```
ToDoItemView >> model: aModel
    model := aModel
```

Now we can define the rendering method for our new component. Note that this is a nice example showing the diversity of brushes since we use a different brush for each entity we manipulate.

```
ToDoItemView >> renderContentOn: html
    html heading: 'Edit'.
    html form: [
        html text: 'Title:'; break.
        html textInput
            value: self model title;
            callback: [ :value | self model title: value ].
        html break.
        html text: 'Due:'; break.
        html dateInput
            value: self model due;
            callback: [ :value | self model due: value ] ]
```

Finally, we make sure that this new component is used when we edit an item. To do this, we redefine the method edit: of the class ToDoListView so that it calls the new component on the item we want to edit. Note that the method call: takes a component as a parameter and that this component will be displayed in place of the calling component, see Chapter .

```
ToDoListView >> edit: anItem
    self call: (ToDoItemView new model: anItem)
```

If you click on the edit link of an item you will be able to edit the item. You will notice one tiny problem with the editor: we do not yet let users save or commit their changes! We will correct this in the next section.

In the meantime, add a style sheet to make the editor look nice:

```
ToDoItemView >> style
    ^ 'body {
    color: #222;
    font-size: 75%;
    font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}
```

```
h1 {
    color: #111;
    font-size: 2em;
    font-weight: normal;
    margin-bottom: 0.5em;
}'
```

13.7 Answer

We just saw how one component can call another and that the other component will appear in place of the one calling it. How do we give back control to the caller or return a result? The method `answer:` performs this task. It takes an object that we want to return as a parameter.

Let's demonstrate how to use `answer:`. We will add two buttons to the interface of the `ToDoItemView`: one for cancelling the edit and one to return the modified item. Note that in one case we use a normal `submitButton`, and in the other case we use `cancelButton`. Pay attention since the cancel button *looks* exactly the same as a submit button, but it avoids processing the input callbacks of the form that would modify our model. This means we don't need to copy the model as we did in .

```
ToDoItemView >> renderContentOn: html
    html heading: 'Edit'.
    html form: [
        html text: 'Title:'; break.
        html textInput
            value: self model title;
            callback: [ :value | self model title: value ].
        html break.
        html text: 'Due:'; break.
        html dateInput
            value: self model due;
            callback: [ :value | self model due: value ].
        html break.
        html submitButton
            callback: [ self answer: self model ];
            text: 'Save'.
        html cancelButton
            callback: [ self answer: nil ];
            text: 'Cancel' ]
```

Working directly on the model. Now the use of the cancel button does solve the problem in the above example, but generally this approach isn't sufficient by itself: when a component returns an answer, you often want to do some additional validation on the potentially invalid object before updating your model.

Therefore, we should also modify the method `edit:` to edit a *copy* of the item and, depending on the returned value of the editor, we should replace the current item with its modified copy.

```
ToDoListView >> edit: anItem
    | edited |
    edited := self call: (ToDoItemView new model: anItem copy).
    edited isNil
        ifFalse: [ self model replace: anItem with: edited ]
```

Add the following method to ToDoList:

```
ToDoList >> replace: aTodoItem with: anotherItem
    self items at: (self items indexOf: aTodoItem) put: anotherItem
```

Magritte Support. Replacing a copied object works well in our example, but does not if there are other references to the object (because you end up with a new object). One of the advanced features of Magritte is that it uses a *Memento* to support the automatic cancellation of edited objects: in other words, it copies the whole object during the edit operation into an internal data-structure and then edits only this object. As soon as the changes are saved, it walks over the Memento and pushes the changes to the real object.

13.8 Embedding Child Components

So far, we have seen how a component displays itself and how a component can invoke another one. This component invocation has behaved like a modal interface in which you can interact only with one dialog at a time. Now, we will demonstrate the real power of Seaside: creating an application by simply plugging together components which may have been independently developed. How do we get several components to display on the same page? By simply having a component identify its subcomponents. This is done by implementing the `children` method.

Figure 13-9 Getting an editor to edit new item.

Suppose that we would like to add an item to our list. Normally a web application developer would use a single form which would be used both to edit and to add a todo item, but for demonstration purposes we take a different approach. We would like to display the editor below the list. That is, we want to *embed* a `ToDoItemView` in a `ToDoListView`. Our solution is to allow the user to add an item by pressing a button which will display an editor for the new item, as seen in 13-9.

We begin by adding an instance variable named `editor` to the `ToDoListView` class as follows:

```
WAComponent subclass: #ToDoListView
   instanceVariableNames: 'editor'
   classVariableNames: ''
   package: 'ToDo-View'
```

Then, we define the method `children` that returns an array containing all the subcomponents of our component. This array contains just the element `editor` since list items are rendered by the list component itself. Note that Seaside automatically ignores component children that are nil, so we don't have to worry if it is not initialized.

```
ToDoListView >> children
    ^ Array with: editor
```

We modify `renderContentOn:` to add an *Add* button and to trigger the add action. Note that when the value of the instance variable `editor` is nil the rendering does not show anything.

```
ToDoListView >> renderContentOn: html
    html heading: self model title.
    html form: [
        html unorderedList: [ self renderItemsOn: html ].
        html submitButton
            text: 'Save'.
        html submitButton
            callback: [ self add ];
            text: 'Add' ].
    html render: editor
```

Figure 13-10 With an item added.

Next we redefine the method `add` to add a new component. It first creates an instance of `ToDoItemView` whose model is a newly created todo item.

```
ToDoListView >> add
    editor := ToDoItemView new model: ToDoItem new
```

Notification of `answer:` messages. How do we update the todo list model? Suppose the user cancels the editing. How do we handle that situation? We need a way to know when a subcomponent executed the method. You can get notified of `answer:` execution by using the method `onAnswer:`. Using `onAnswer:` involves modifying the `initialize` method of the parent component to specify what should be done when one of the subcomponents executes `answer:`. The method `onAnswer:` requires a block whose argument represents the object that got answered (`parent onAnswer: [ :object | ... ]`).

The `onAnswer:` block will be executed with the answered object as its argument. Since the editor will return `nil` when the user cancels editing, we need to check the value passed in. We modify the `add` method as follows:

```
ToDoListView >> add
    editor := ToDoItemView new model: ToDoItem new.
    editor onAnswer: [ :value |
       value isNil
           ifFalse: [ self model add: value ].
       editor := nil ]
```

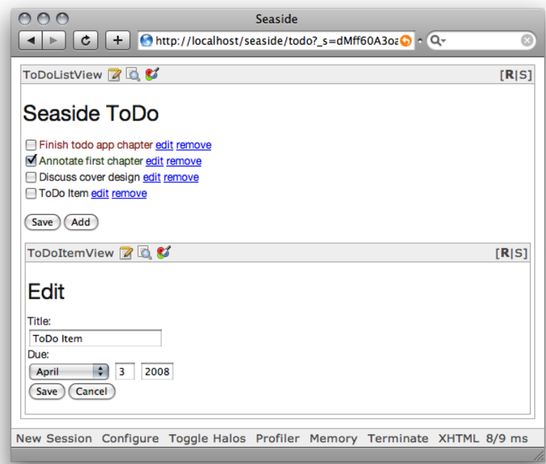


Figure 13-11 Our final todo App

Note that the *Save* button is different from the *Add* button since the *Save* button (so far) does nothing but submit the form. In the AJAX chapter, we will see that this situation can be avoided altogether (see Chapter).

If you get the error "Children not found while processing callbacks", check that the `children` method returns all the direct subcomponents. The halos are another good tool for understanding the nesting and structure of components. We suggest you turn on the halos while developing your applications, as seen in 13-11.

13.9 Summary

You have briefly reviewed all the key mechanisms offered by Seaside to build a complex dynamic application out of reusable components. You can either invoke another component or compose a component out of existing ones. Each component has the responsibility to render itself and return its subcomponents.

CHAPTER 14

A Web Sudoku Player

In this chapter we will build a Sudoku web application as shown in Figure 14-1. This gives us another opportunity to revisit how we build a simple application in Seaside.

	1	2	3	4	5	6	7	8	9
A	23456789	23456789	23456789	456789	1	456789	23456789	23456789	23456789
B	13456789	13456789	13456789	456789	2	456789	13456789	13456789	13456789
C	12456789	12456789	12456789	456789	3	456789	12456789	12456789	12456789
D	12356789	12356789	12356789	123789	4	123789	12356789	12356789	12356789
E	12346789	12346789	12346789	123789	5	123789	12346789	12346789	12346789
F	12345789	12345789	12345789	123789	6	123789	12345789	12345789	12345789
G	123456789	123456789	123456789	123456789	789	123456789	123456789	123456789	123456789
H	123456789	123456789	123456789	123456789	789	123456789	123456789	123456789	123456789
I	123456789	123456789	123456789	123456789	789	123456789	123456789	123456789	123456789

Solve

Figure 14-1 Just started playing.

14.1 The solver

For the Sudoku model we use the ML-Sudoku package developed by Martin Laubach which is available on SqueakSource. We thank him for allowing us to use this material. To load the package, open a Monticello browser and click on the *+Repository* button. Select HTTP as the type of repository and specify it as follows:

```
MCHttpRepository
    location: 'http://www.squeaksource.com/MLSudoku'
    user: ''
    password: ''
```

Click on the *Open* button, select the most recent version and click *Load*.

ML-Sudoku is composed of 4 classes: `MLSudoku`, `MLBoard`, `MLCell`, and `MLPossibilitySet`. The class responsibilities are distributed as follows:

- `MLSudoku` is the Sudoku solver. It knows how to solve a Sudoku.
- `MLCell` knows its neighbors and their location on the game board. A cell does not know its possibility set, see `MLPossibilitySet` below.
- `MLBoard` contains the cells and their possibility sets.
- `MLPossibilitySet` is a list of possible numbers between 1 and 9 that can go into a cell. These are the values that are possible without violating the Sudoku rule that each row, column and 3-by-3 sub-grid contains each number once.

14.2 Sudoku

First we define the class `WebSudoku` which is the Sudoku UI. We will create this class in the new `ML-WebSudoku` category:

```
WAComponent subclass: #WebSudoku
    instanceVariableNames: 'sudoku'
    classVariableNames: ''
    package: 'ML-WebSudoku'
```

This component will contain a Sudoku solver (an instance of `MLSudoku`) which we will refer to using the instance variable `sudoku`. We initialize this variable by defining the following `WebSudoku>>initialize` method.

```
WebSudoku >> initialize
    super initialize.
    sudoku := MLSudoku new
```

Describing and registering the application. Now we add a few methods to the *class* side of our component. We describe the application by defining the `description` method. We register our component as an application by defining the class `initialize` method and declare that the component `WebSudoku` can be a standalone application by having `canBeRoot` return true:

```
WebSudoku class >> description
    ^ 'Play Sudoku'
```

```
WebSudoku class >> initialize
    WAAdmin register: self asApplicationAt: 'sudoku'
```

```
WebSudoku class >> canBeRoot
    ^ true
```

Finally we define a CSS style for the Sudoku component:

```
WebSudoku >> style
    ^ '#sudokuBoard .sudokuHeader {
        font-weight: bold;
        color: white;
        background-color: #888888;
}

#sudokuBoard .sudokuHBorder {
        border-bottom: 2px solid black;
}

#sudokuBoard .sudokuVBorder {
        border-right: 2px solid black;
}

#sudokuBoard .sudokuPossibilities {
        font-size: 9px;
}

#sudokuBoard td {
        border: 1px solid #dddddd;
        text-align: center;
        width: 55px;
        height: 55px;
        font-size: 14px;
}

#sudokuBoard table {
        border-collapse: collapse
}

#sudokuBoard a {
        text-decoration: none;
        color: #888888;
}

#sudokuBoard a:hover {
        color: black;
}'
```

14.3 Rendering the Sudoku Grid

What we need to do is to render a table that looks like a Sudoku grid. We start by defining a method that creates a table and uses style tags for formatting.

```
WebSudoku >> renderHeaderFor: aString on: html
    html tableData
        class: 'sudokuHeader';
        class: 'sudokuHBorder';
        class: 'sudokuVBorder';
        with: aString
```

```
WebSudoku >> renderContentOn: html
    self renderHeaderFor: 'This is a test' on: html
```

Make sure that you invoked the class side `initialize` method and then run the application by visiting http://localhost:8080/sudoku. Your browser should display the string *This is a test*.

We need two helper methods when creating the labels for the `x` and `y` axis of our Sudoku grid. You don't need to actually add these helper methods yourself, they were already loaded when you loaded the Sudoku package:

```
Integer >> asSudokuCol
    "Label for columns"
    ^ ($1 to: $9) at: self
```

```
Integer >> asSudokuRow
    "Label for rows"
    ^ ($A to: $I) at: self
```

First we print a space to get our labels aligned and then draw the label for each column.

```
WebSudoku >> renderBoardOn: html
    html table: [
        html tableRow: [
            self renderHeaderFor: '' on: html.
            1 to: 9 do: [ :col |
                self renderHeaderFor: col asSudokuCol on: html ] ] ]
```

We make sure that the method `WebSudoku>>renderContentOn:` invokes the board rendering method we just defined.

```
WebSudoku >> renderContentOn: html
    self renderBoardOn: html
```

1 2 3 4 5 6 7 8 9

Figure 14-2 The column labels look like this.

If you run the application again, you should see the column labels as shown in Figure 14-2.

We now draw each row with its label, identifying the cells with the product of its row and column number.

```
WebSudoku >> renderBoardOn: html
    html table: [
        html tableRow: [
            self renderHeaderFor: '' on: html.
            1 to: 9 do: [ :col | self renderHeaderFor: col asString
    on: html ] ].
        1 to: 9 do: [ :row |
            html tableRow: [
                self renderHeaderFor: row asSudokuRow on: html.
                1 to: 9 do: [ :col |
                    html tableData: (col * row) asString ] ] ] ]
```

If you have entered everything correctly, your page should now look like Figure 14-3.

	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	9
B	2	4	6	8	10	12	14	16	18
C	3	6	9	12	15	18	21	24	27
D	4	8	12	16	20	24	28	32	36
E	5	10	15	20	25	30	35	40	45
F	6	12	18	24	30	36	42	48	54
G	7	14	21	28	35	42	49	56	63
H	8	16	24	32	40	48	56	64	72
I	9	18	27	36	45	54	63	72	81

Figure 14-3 Row labels are letters, column labels are numbers.

Now we define the method `renderCellAtRow:col:on:` that sets the style tags and redefine the `renderContentOn:` as follows so that it uses our style sheet.

```
WebSudoku >> renderContentOn: html
    html div
        id: 'sudokuBoard';
        with: [ self renderBoardOn: html ]
```

```
WebSudoku >> renderCellAtRow: rowInteger col: colInteger on: html
    html tableData
        class: 'sudokuHBorder' if: rowInteger \\ 3 == 0;
        class: 'sudokuVBorder' if: colInteger \\ 3 == 0
```

You also need to change `WebSudoku>>renderBoardOn:` so that it uses our new method `WebSudoku>>renderCellAtRow:col:on:`.

```
WebSudoku >> renderBoardOn: html
    html table: [
        html tableRow: [
            self renderHeaderFor: '' on: html.
            1 to: 9 do: [ :col | self renderHeaderFor: col asString
    on: html ] ].
        1 to: 9 do: [ :row |
            html tableRow: [
                self renderHeaderFor: row asSudokuRow on: html.
                1 to: 9 do: [ :col | self renderCellAtRow: row col:
    col on: html ] ] ] ]
```

If you refresh your page again, you should finally see a styled Sudoku grid as show in 14-4.

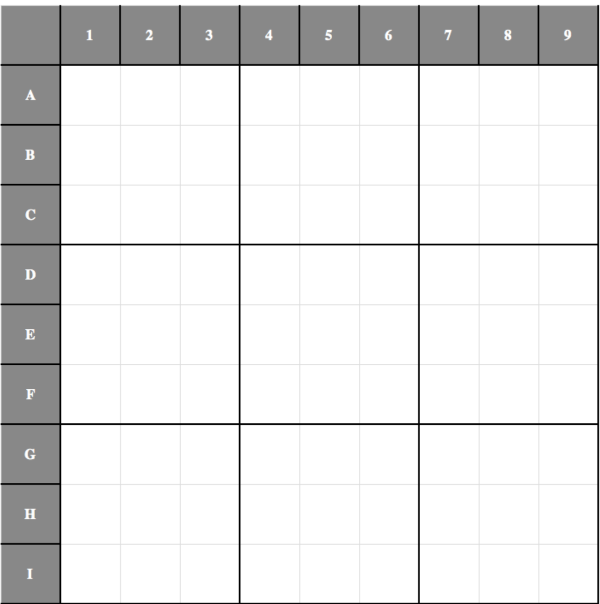


Figure 14-4 The Sudoku board with the style sheet applied.

Next we will use a small helper method `asCompactString` that given a collection returns a string containing all the elements printed one after the other without spaces. Again, you do not need to type this method, it was loaded with the ML-Sudoku code.

```
Collection >> asCompactString
    | stream |
    stream := WriteStream on: String new.
    self do: [ :each | ws nextPutAll: each printString ].
    ^ stream contents
```

We define a new method `renderCellContentAtRow:col:on:` that uses `asCompactString` to display the contents of a cell. Each cell displays its possibility set. These are the values that may

legally appear in that cell.

```
WebSudoku >> renderCellContentAtRow: rowInteger col: colInteger on:
    html
    | currentCell possibilites |
    currentCell := MLCell row: rowInteger col: colInteger.
    possibilites := sudoku possibilitiesAt: currentCell.
    possibilites numberOfPossibilities = 1
        ifTrue: [ html text: possibilites asCompactString ]
        ifFalse: [
            html span
                class: 'sudokuPossibilities';
                with: possibilites asCompactString ]
```

We make sure that the `renderCellAtRow:col:on:` invokes the method rendering cell contents.

```
WebSudoku >> renderCellAtRow: rowInteger col: colInteger on: html
    html tableData
        class: 'sudokuHBorder' if: rowInteger \\ 3 = 0;
        class: 'sudokuVBorder' if: colInteger \\ 3 = 0;
        with: [ self renderCellContentAtRow: rowInteger col:
    colInteger on: html ]
```

Refresh your application again, and your grid should appear as in 14-5.

	1	2	3	4	5	6	7	8	9
A	5	3	1247	124679	14789	1246789	146789	12469	124678
B	6	1247	1247	1234579	1345789	1245789	1345789	123459	123478
C	1247	9	8	1234567	13457	124567	134567	123456	123467
D	1234789	124578	1234579	14579	6	14579	1345789	123459	123478
E	12479	124567	1245679	8	14579	3	145679	124569	12467
F	134789	145678	1345679	14579	2	14579	13456789	134569	134678
G	13479	14567	1345679	1345679	134579	145679	2	8	1346
H	1234789	124678	1234679	1234679	134789	1246789	1346	1346	5
I	12348	124568	123456	123456	13458	124568	1346	7	9

Figure 14-5 The Sudoku grid is showing the possible values for each cell.

14.4 Adding Input

Now we will change our application so that we can enter numbers into the cells of the Sudoku grid. We define the method `setCell:to:` that changes the state of a cell and we extend the method `renderCellContentAtRow:col:on:` to use this new method.

```
WebSudoku >> setCell: aCurrentCell to: anInteger
    sudoku atCell: aCurrentCell removeAllPossibilitiesBut: anInteger
```

```
WebSudoku >> renderCellContentAtRow: rowInteger col: colInteger on:
    html
    | currentCell possibilities |
    currentCell := MLCell row: rowInteger col: colInteger.
    possibilities := sudoku possibilitiesAt: currentCell.
    possibilities numberOfPossibilities = 1
        ifTrue: [ ^ html text: possibilities asCompactString ].
    html span
        class: 'sudokuPossibilities';
        with: possibilities asCompactString.
    html break.
    html form: [
        html textInput
            size: 2;
            callback: [ :value |
                | integerValue |
                integerValue := value asInteger.
                integerValue isNil ifFalse: [
                (possibilities includes: integerValue)
                     ifTrue: [ self setCell: currentCell to:
    integerValue ] ] ] ]
```

The above code renders a text input box within a form tag, in each cell where there are more than one possibilities. Now you can type a value into the Sudoku grid and press return to save it, as seen in 14-6. As you enter new values, you will see the possibilities for cells automatically be automatically reduced.

Now we can also ask the Sudoku model to solve itself by modifying the method `renderContentOn:`. We first check whether the Sudoku grid is solved and if not, we add an anchor whose callback will solve the puzzle.

```
WebSudoku>>renderContentOn: html
    html div id: 'sudokuBoard'; with: [
        self renderBoardOn: html.
        sudoku solved ifFalse: [
            html break.
            html anchor
                callback: [ sudoku := sudoku solve ];
                with: 'Solve' ] ]
```

Note that the solver uses backtracking, i.e., it finds a missing number by trying a possibility and if it fails to find a solution, it restarts with a different number. To backtrack the solver works on copies of the Sudoku grid, throwing away grids that don't work and restarting. This is why we need to assign the result of sending the message `solve` since it returns a new Sudoku grid. Figure 14-7 shows the result of clicking on Solve.

	1	2	3	4	5	6	7	8	9
A	5	3	1247	124679	14789	1246789	146789	12469	124678
B	6	1247	1247	1234579	1345789	1245789	1345789	123459	123478
C	1247	9	8	1234567	13457	124567	134567	123456	123467
D	1234789	124578	1234579	14579	6	14579	1345789	123459	123478
E	12479	124567	1245679	8	14579	3	145679	124569	12467
F	134789	145678	1345679	14579	2	14579	13456789	134569	134678
G	13479	14567	1345679	1345679	134579	145679	2	8	1346
H	1234789	124678	1234679	1234679	134789	1246789	1346	1346	5
I	12348	124568	123456	123456	13458	124568	1346	7	9

Figure 14-6 A partially filled Sudoku grid.

	1	2	3	4	5	6	7	8	9
A	5	3	4	6	7	8	9	1	2
B	6	7	2	1	9	5	3	4	8
C	1	9	8	3	4	2	5	6	7
D	8	5	9	7	6	1	4	2	3
E	4	2	6	8	5	3	7	9	1
F	7	1	3	9	2	4	8	5	6
G	9	6	1	5	3	7	2	8	4
H	2	8	7	4	1	9	6	3	5
I	3	4	5	2	8	6	1	7	9

Figure 14-7 A solved Sudoku grid.

14.5 Back Button

Now let's play a bit. Suppose we have entered the values 1 through 6 as shown in Figure 14-1 and we want to replace the 6 with 7. If we press the *Back* button, change the 6 to 7 and press return, we get 6 instead of 7. The problem is that we need to copy the state of the Sudoku grid before making the cell assignment in `setCell:to:`.

```
WebSudoku >> setCell: currentCell to: anInteger
    sudoku := sudoku copy
                atCell: currentCell
                removeAllPossibilitiesBut: anInteger
```

If you change the definition of `setCell:to:` and try to replace 6 with 7 you will get a stack error similar to that shown in 14-8.

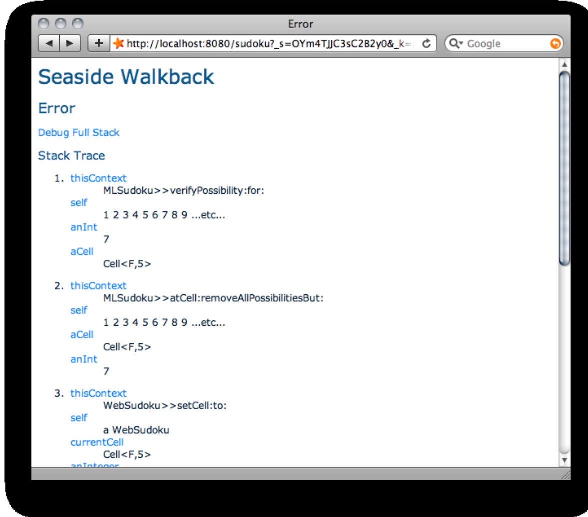


Figure 14-8 Error when trying to replace 6 by 7.

If you click on the debug link at the top of the stack trace and then look at your Pharo image, you will see that it has opened a debugger. Now you can check the problem. If you select the expression

```
self possibilitiesAt: aCell
```

and print it, you will get a possibility set with 6, which is the previous value you gave to the cell. The code of the method `MLSudoku>>verifyPossibility:for:` raises an error if the new value is not among the possible values for that cell.

```
MLSudoku >> verifyPossibility: anInteger for: aCell
    ((self possibilitiesAt: aCell) includes: anInteger)
        ifFalse: [ Error signal ]
```

In fact when you pressed the *Back* button, the Sudoku UI was refreshed but its model was still holding the old values. What we need to do is to indicate to Seaside that when we press the *Back* button the state of the model should be kept in sync and rollback to the corresponding older

version. We do so by defining the method `states` which returns the elements that should be kept in sync.

```
WebSudoku >> states
   ^ Array with: self
```

14.6 Summary

While the Sudoku solver introduces some subtleties because of its backtracking behavior, this application shows the power of Seaside to manage state.

Now you have a solid basis for building a really powerful Sudoku online application. Have a look at the class MLSudoku. Extend the application by loading challenging Sudoku grids that are defined by a string.

CHAPTER 15

Serving Files

Most web-based applications make heavy use of static resources. By "static" we mean resources whose contents are not sensitive to the context in which they are used. These resources are not dependent on the user or session state and while they may change from time to time they typically don't change during the time span of a single user's session. Static resources include for example images, style sheets and JavaScript files.

Using these resources in a Seaside application need be no different from using them in any other web application development framework: when deploying your application you can serve these resources using a web server and reference them in your Seaside application, as described in Chapter **??**.

In addition, Seaside supports a more tightly integrated file serving technique, called *FileLibrary*, which has some advantages over using a separate web server. In this chapter we will cover how to reference external resources and how to use the integrated FileLibrary to serve them from your Smalltalk image. Note that using FileLibrary to serve static resources is often slower than using a dedicated web server. In Chapter we explain how to serve static files in a more efficient way using Apache.

15.1 Images

We illustrate the inclusion of static resources by displaying an external picture within an otherwise empty component as shown in Figure 15-1. Create a component and use the method `WAImageTag>>url:` to add a URL to an image as follows:

```
ComponentWithExternalResource >> renderContentOn: html
    html image url: 'http://www.seaside.st/styles/logo-plain.png'
```

If you have many static files that all live in the same location, it is annoying to have to repeat the base-path over and over again. In this case you should use `WAImageTag>>resourceUrl:` to provide the tail of the URL.

```
ComponentWithExternalResource >> renderContentOn: html
    html image resourceUrl: 'styles/logo-plain.png'
```

Figure 15-1 Including an external picture into your components.

To tell Seaside about the part of the URL that you left out in your rendering code you have to go to the application configuration page (at http://localhost:8080/config) and specify the *Resource Base URL* in the server settings. Just enter `http://www.seaside.st` as shown in Figure 15-2. Seaside will automatically prepend this string to all URLs specified using `resourceUrl:>>resourceUrl:`. This reduces your code size and can be very useful if you want to move the resource location during deployment.

Be careful where you put the slash. Normally directories in URLs end with a slash, that's why we specified the resource base URL ending with a slash. Thus, you should avoid putting a slash at the beginning of the URL fragments you pass to `resourceUrl:`.

Serving a generated image

Another interesting way to serve a picture is to use a dynamically generated picture from within your image (see Figure 15-3). It is possible to use `WAImageTag>>form:` to pass a Pharo `Form` directly to the image brush.

```
ComponentWithForm >> renderContentOn: html
    html image form: aForm
```

That works reasonably well for simple graphics, however most visual things in Pharo are made using morphs. Luckily it is simple to convert a morph to a form:

```
ComponentWithForm >> renderContentOn: html
    html image form: (EllipseMorph new
       color: Color orange;
       extent: 200 @ 100;
       borderWidth: 3;
       imageForm)
```

You can also use `WAImageTag>>document:` as follows:

```
html image document: EllipseMorph new
```

Have a look at the example implemented in the class `WAScreenshot`. It demonstrates a much more sophisticated use of `WAImageTag>>form:` and presents the Pharo desktop as part of a web application. Furthermore it allows basic interactions with your windows from the web browser.

Server

Resourc

Server P

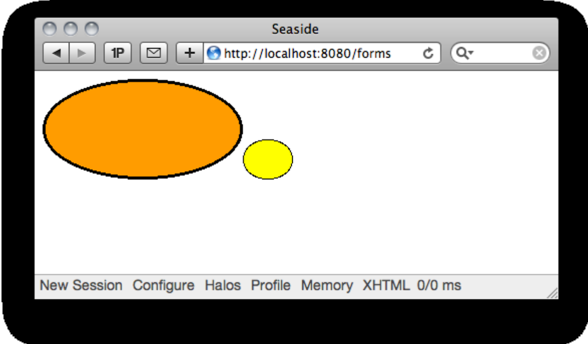


Figure 15-3 Displaying Pharo graphical object..

15.2 Including CSS and Javascript

So far, we've been including style information for our components by implementing the `style` method on our components. This is great for dynamic development, but there are a number of problems with this approach:

- Seaside is generating a style sheet file each time your component is rendered. This takes time to generate.
- Each generated stylesheet has the session key embedded in its URL, and so is seen as a unique file by your browser, and so loaded again.
- As you integrate more components in your page, each is generating its own stylesheet, so you can end up with many resources to be downloaded for each page.

Once your application's look and feel has begun to stabilise, you will want to think about using static stylesheets. These are typically included by using `link` tags in the `head` section of the XHTML document. This presents us with a problem: by the time your component gets sent `renderContentOn:`, the canvas has already generated the `head` section.

Fortunately, Seaside provides a hook method called `WAComponent>>updateRoot:` which is sent to all components which are reachable directly or indirectly through children or a `call:` message – which means basically to all visible components. This message is sent during the generation of the body of the `head` tag and can be extended to add elements to this tag. The argument to `updateRoot:` is an instance of `WAHtmlRoot` which supports the access to document elements such as `<title>`, `<meta>`, `<javascript>` and `<stylesheet>` with their corresponding messages (`WAHtmlRoot>>title`, `WAHtmlRoot>>meta`, `WAHtmlRoot>>javascript` and `WAHtmlRoot>>stylesheet`). It also allows you to add attributes to the `<head>` or `<body>` tags using the messages `WAHtmlRoot>>headAttributes`, `WAHtmlRoot>>bodyAttributes`.

In particular, `WAHtmlRoot` offers the possibility to add new styles or script using the messages `WAHtmlRoot>>addScript:` and `WAHtmlRoot>>addStyles:`.

The object returned by both `stylesheet` and `javascript` understands `url:` which allows you to specify the URL of the stylesheet or JavaScript file. Suppose we have a stylesheet being served from http://seaside.st/styles/main.css. We could adopt this style in our document by extending `updateRoot:` as follows:

```
WAComponent subclass: #ComponentWithStyle
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Serving-Files'
```

```
ComponentWithStyle >> updateRoot: anHtmlRoot
    super updateRoot: anHtmlRoot.
    anHtmlRoot stylesheet url: 'http://seaside.st/styles/main.css'
```

```
ComponentWithStyle >> renderContentOn: html
    html heading level: 1; with: 'Seaside'.
    html text: 'This component uses the Seaside style.'
```

Running the example should give you the following 15-4:

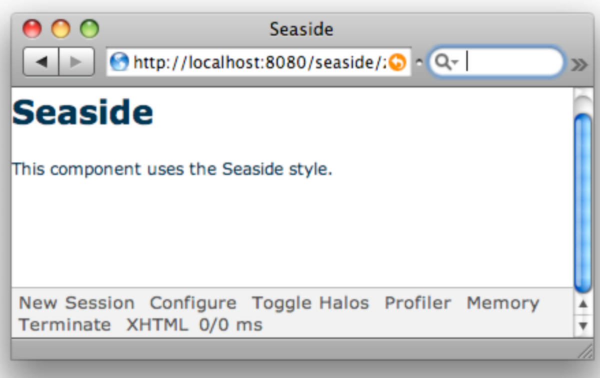


Figure 15-4 Application with enabled style sheet.

Now we will show how you can replace the stylesheet using the FileLibrary.

15.3 Working with File Libraries

Seaside includes a library for serving files called *FileLibrary*. This solution is handy for rapid application development and is suitable for deployed applications which only make use of a small number of small files. It has the advantage that all of the resources are contained in your Smalltalk image and can be versioned with your favorite Smalltalk version management tools. However this also means that these resources are **not** reachable where most of your operating system's tools are accustomed to find things.

FileLibrary has the primary advantage that it is a portable way to serve static contents directly from Seaside without the need to setup a standalone web server. See Chapter to read about Apache configuration for static file serving.

Creating a File Library

Setting up a file library is easy. Here are the steps you need to follow.

1. Put your static files in a directory. The location of the directory is not significant. From within the directory, the files can reference each other using their file names.
2. Create a file library by subclassing `WAFileLibrary`. For the rest of this text we assume its name is `MyFileLibrary`.
3. Add files to your file library. There are three ways to add files to your file library:

- Programmatically. - Via the web interface. - By editing your `MyFileLibrary` directly in your image.

Adding files programmatically. You can add files programmatically by using the class side methods `addAllFilesIn:` and `addFileAt:` in `MyFileLibrary`. For example:

```
MyFileLibrary addAllFilesIn: '/path/to/directory'
MyFileLibrary addFileAt: '/path/to/background.png'
```

Adding files via the config interface. Open the config application at http://localhost:8080/config and click the "configure" link for file libraries as shown in Figure 15-5. This shows which file libraries are available.

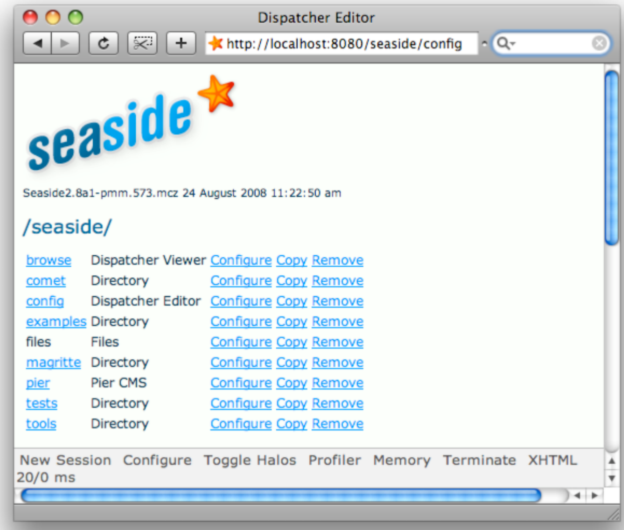


Figure 15-5 Configuring file libraries through the web interface: clicking on files - configure.

Click the configure link for `MyFileLibrary` as shown in 15-6 right.

There you can add a file by uploading it (select the file, then click the *Add* button as shown by 15-7).

When you add a file to a file library, Seaside creates a method with the file contents. If you find that there is an unusually long wait after pressing the *Add* button, make sure that the system (Squeak/Pharo) isn't waiting for you to type your initials to confirm that you want to create a new method. **Adding a file by editing the class.** File libraries are just objects and "files" in the

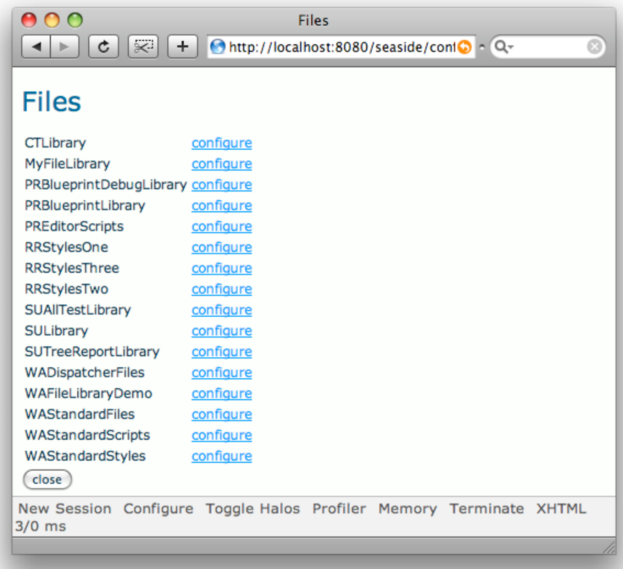


Figure 15-6 File libraries.

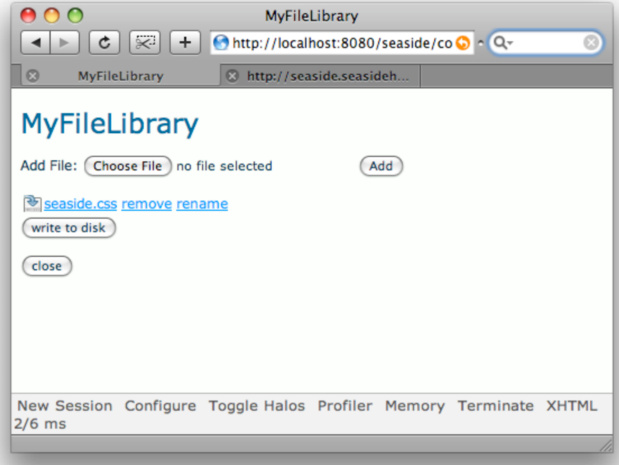


Figure 15-7 File libraries.

file library are just methods so you can always add and modify FileLibrary entries using your normal class browser but be sure to follow the method naming convention mentioned above. You'll probably find it pretty inconvenient to edit images within the browser though.

Adding a file to a file library either programmatically or using the configuration interface defines a corresponding method in the file library class, with the file name determining the name of the method. The dot is removed and the first letter of the suffix is capitalized. For example, the file main.css becomes the method `MyFileLibrary>>mainCss`. This puts certain limitations on the allowed file names. For example, the main part of the file name may not be all digits.

Once your files have been imported into the file library they are maintained independently from the files on your computer's file system. If you modify your files you will have to re-add them to the file library.

Once your files are stored in a FileLibrary they will be available to be served through Seaside.

Referencing FileLibrary files by URL

How you use a file library depends on what you want to do with the files in it. As you've seen in the previous sections, using image, music, style sheets and JavaScript files requires knowing their URL. You can find the URL of any document in your file library by sending the class `WAFileLibrary class>>urlOf:`. For example, if you had added the file `picture.jpg` to your library and you want to display it in a component you would write something like:

```
MyClass>>renderContentOn: html
    html image url: (MyFileLibrary urlOf: #pictureJpg)
```

The URL returned by `urlOf:` is relative to the current server. It does not contain the http://servername.com/ - the so-called "method *and "host* - portion of the URL. Note that WAFileLibrary implements a class method called `/`, so the expression `MyFileLibrary / #pictureJpeg` is equivalent to `MyFileLibrary urlOf: #pictureJpeg`.

Once you know the URL of the FileLibrary resources you can use them to include style sheets and JavaScript in your components as we have already discussed.

15.4 Example of FileLibrary in Use

We've gone on long enough without a working hands-on example. To illustrate how to use a file library, we will show how to add some resources to the WebCounter application we defined in the first chapter of this book (http://localhost:8080/webcounter) or can also use the version that comes with Seaside (http://localhost:8080/examples/counter). First we create a new subclass of `WAFileLibrary` named `CounterLibrary` as follows:

```
WAFileLibrary subclass: #CounterLibrary
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Test'
```

First as you can see in Figure 15-8 the counter library is empty.

We follow the steps presented in the previous section and associate two resources to our library (see Figure 15-9). One is an icon named `seaside.png` and the other is a CSS file named `seaside.css` – you can download the ones from the Seaside website we mentioned before:

seaside.pnghttp://www.seaside.st/styles/logo-plain.png (rename once downloaded).

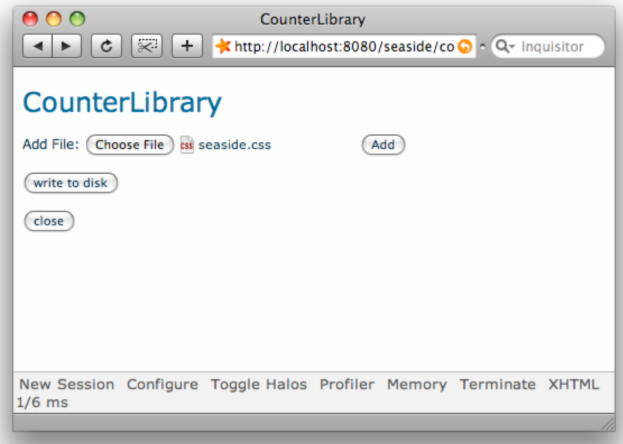


Figure 15-8 An empty CounterLibrary.

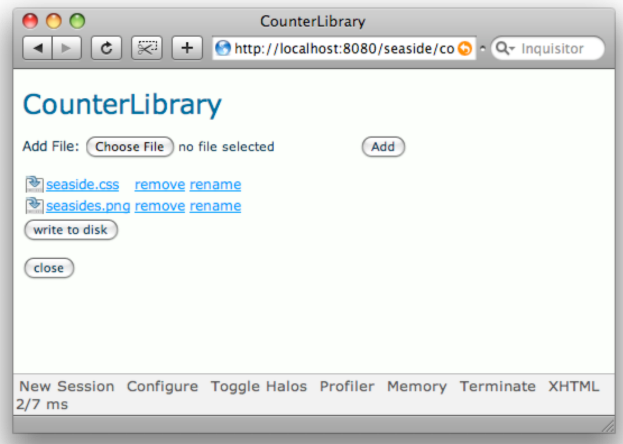


Figure 15-9 Adding files to the CounterLibrary.

seaside.csshttp://seaside.st/styles/main.css Pay attention that the file name of your resources does not contain non-alphabetic characters since it may cause problems. Now we change the method `renderContentOn:` – this shows how we access resources using the `urlOf:`.

```
WebCounter >> renderContentOn: html
    html image url: (CounterLibrary urlOf: #seasidePng).
    html heading: count.
    html anchor
        callback: [ self increase ];
        with: '++'.
    html space.
    html anchor
        callback: [ self decrease ];
        with: '--'
```

Next we implement `updateRoot:` so that our component contains a link to our style sheet:

```
WebCounter >> updateRoot: anHtmlRoot
   super updateRoot: anHtmlRoot.
   anHtmlRoot stylesheet url: (CounterLibrary urlOf: #seasideCss)
```

This causes the look of our application to change. It now uses the CSS file we added to our file library as shown by 15-10.

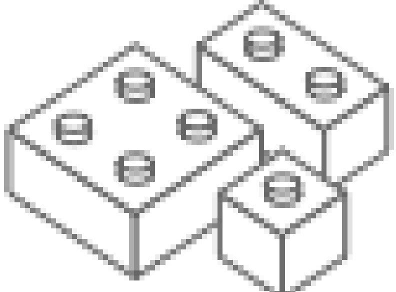
Have a look at the XHTML source generated by Seaside by using your browser's View Source option. You will see that the links are added to the head section of the HTML document as shown below:

```
...
<link rel="stylesheet" type="text/css"
    href="/files/CounterLibrary.css"/>
</head>
<body onload="onLoad()" onkeydown="onKeyDown(event)">
  <img alt="" src="/files/CounterLibrary/seaside.png"/>
  <h1>0</h1>
  <a
    href="http://localhost:8080/WebCounter?_s=UwGcN6vwGVmj9icD&amp;_k=D6Daqxer&amp;1"
  <a
    href="http://localhost:8080/WebCounter?_s=UwGcN6vwGVmj9icD&amp;_k=D6Daqxer&amp;2"
...
```

15.5 Which method should I use?

You have the following choices for serving static files with your Seaside application:

- The default answer is pretty simple: if you don't know anything about web servers, use `FileLibrary`.
- If you want to have your static resources versioned inside your Smalltalk image and don't have too many (or too large) resources, use `FileLibrary`.
- If you prefer to keep your static resources on your file system where you can edit and version them with your favorite file-based tools but you don't want to run a separate web server, go read about how to serve static content from your image in .
- Otherwise read Chapter about Apache file serving and configuration.



1

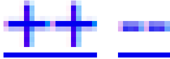


Figure 15-10 Counter with the updateRoot: method defined.

15.6 A Word about Character Encodings

Character encoding is an area that we programmers tend to avoid as much as possible, often fixing problems by trial and errors. With web-development you will sooner or later be bitten by character encoding bugs, no matter how you try to escape them. As soon as you are getting inputs from the user and displaying information in your web-browser, you will be confronted with character encoding problems. However, the basic concepts are simple to understand and the difficulty often lies in the extra layers that typically a web developer does not have to worry about such as the web-rendering engine, the web server and the input keyboard.

In this section we'll present the two basic concepts you have to understand - *character sets* and *character encodings*. This should help you avoid most problems. Then we will tell you how these are supported in Seaside. In addition, we strongly suggest to read the Chapter about encodings written by Sven van Caekenberghe in the "Entreprise Pharo: a Web Perspective" book.

Historically the difference between character sets and character encoding was minor, since a standard specified what characters were available as well as how they encoded. Unicode and ISO 10646 (Universal Character Set) changed this situation by clearly separating the two concepts. Such a separation is essential: on one hand you have the character sets you can manipulate and on the other hand you have how they are represented physically (encoded).

Character Sets

A character set is really just that, a set of characters. These are the characters of your alphabet. For practical reasons each character is identified by a *code point* e.g. `$A` is identified by the code point 65.

Examples of character sets are ASCII, ISO-8859-1, Unicode or UCS (Universal Character Set).

- **ASCII** (American Standard Code for Information Interchange) contains 128 characters. It was designed following several constraints such that it would be easy to go from a lowercase character to its uppercase equivalent. You can get the list of characters at http://en.wikipedia.org/wiki/Ascii. ASCII was designed with the idea in mind that other countries could plug their specific characters in it but it somehow failed. ASCII was extended in Extended ASCII which offers 256 characters.
- **ISO-8859-1** (ISO/IEC 8859-1) is a superset of ASCII to which it adds 128 new characters. Also called **Latin-1** or **latin1**, it is the standard alphabet of the latin alphabet, and is well-suited for Western Europe, Americas, parts of Africa. Since ISO-8859-1 did not contain certain characters such as the Euro sign, it was updated into ISO-8859-15. However, ISO-8859-1 is still the default encoding of documents delivered via HTTP with a MIME type beginning with "text/". http://www.utoronto.ca/webdocs/HTMLdocs/NewHTML/iso_table.html shows in particular ISO-8859-1.
- **Unicode** is a superset of Latin-1. To accelerate the early adoption of Unicode, the first 256 code points are identical to ISO-8859-1. A character is not described via its glyph but identified by its code point, which is usually referred to using "U+" followed by its hexadecimal value. Note that Unicode also specifies a set of rules for normalization, collation bi-directional display order and much more.
- **UCS** – the 'Universal Character Set' specified by the ISO/IEC 10646 International Standard contains a hundred thousand characters. Each character is unambiguously identified by a name and an integer also called its code point.

http://www.fileformat.info/info/charset/index.htm shows several character sets.

In Pharo.

Now let us see the concepts exist in Pharo. The `String`, `ByteString`, `WideString` class hierarchy is roughly equivalent to the `Integer`, `SmallInteger`, `LargeInteger` hierarchy. The class

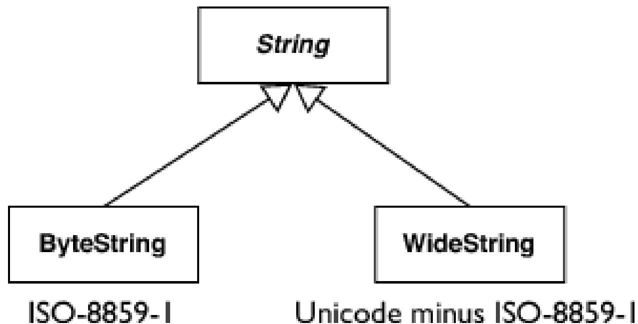


Figure 15-11 The Pharo String Library.

`Integer` is the abstract superclass of `SmallInteger` which represents number with ranges between -1073741824 and 1073741823, and `LargeInteger` which represents all the other numbers. In Pharo, the class `String` is the abstract superclass of the classes `ByteString` (ISO-8859-1) and `WideString` (Unicode minus ISO-8859-1) as shown in Figure 15-11. Such classes are about character sets and not encodings.

Encodings

An encoding is a mapping between a character (or its code point) and a sequence of bytes, and vice versa.

Simple Mappings. The mapping can be a one-to-one mapping between the character and the byte that represents it. If *and only if* your character set has 255 or less entries you can directly map each character by its index to a single byte. This is the case for ASCII and ISO-8859-1.

In the latest version of Pharo, the `Character` class represents a character by storing its Unicode. Since Unicode is a superset of latin1, you can create latin1 strings by specifying their direct values. When a `String` is composed only of ASCII or latin1 characters, it is encoded in a `ByteString` (a collection of bytes each one representing a character).

```
String with: (Character value: 65) with: (Character value: 66)
>>> 'AB'
```

```
'AB' class.
>>> ByteString
```

```
String with: (Character value: 16r5B) with: (Character value: 16r5D)
>>> '[]'
```

```
String with: (Character value: 16rA9)
>>>  the copyright character &copy;
```

```
Character value: 16rFC.
>>> the u-umlaut character &uuml;
```

The characters `Character value: 16r5B` (`[`) and `Character value: 65` (`A`) are both available in ASCII and ISO-8859-1. Now `Character value: 16rA9` displays © the copyright sign which is only available in ISO-8859-1, similarly `Character value: 16rFC` displays ü.

Other Mappings. As we already mentioned Unicode is a large superset of Latin-1 with over hundred thousand of characters. Unicode cannot simply be encoded on a single byte. There exist several character encodings for Unicode: the Unicode Transformation Format (UTF) encodings, and the Universal Character Set (UCS) encodings.

The number in the encodings name indicates the number of bits in one code point (for UTF encodings) or the number of bytes per code point (for UCS) encodings. UTF-8 and UTF-16 are probably the most commonly used encodings. UCS-2 is an obsolete subset of UTF-16; UCS-4 and UTF-32 are functionally equivalent.

- **UTF-8** (8-bits UCS/Unicode Transformation Format) is a variable length character encoding for Unicode. The Dollar Sign ($) is Unicode U+0024. UTF-8 is able to represent any character of the Unicode character sets, but it is backwards compatible with ASCII. It uses 1 byte for all ASCII characters, which have the same code values as in the standard ASCII encoding, and up to 4 bytes for other characters.
- **UCS-2** which is now obsolete used 2 bytes for all the characters but it could not encode all the Unicode standard.
- **UTF-16** extends UCS-2 to encode character missing from UCS-2. It is a variable size encoding using two bytes in most cases. There are two variants – the little endian and big endian versions: `16rFC 16r00` `16r00 16rFC` are variant representations of the same encoded character.

If you want to know more on character sets and character encodings, we suggest you read the Unicode Standard book, currently describing the version 5.0.

In Seaside and Pharo

update this to refer and use Zinc Now let us see how these principles apply to Pharo. The Unicode introduction started with version 3.8 of Squeak and it is slowly consolidated. You can still develop applications with different encodings with Seaside. There is an important rule in Seaside about the encoding: "do unto Seaside as you would have Seaside do unto you". This means that if you run an encoded adapter web server such as `WAKomEncoded`, Seaside will give you strings in the specified encoding but also expect from you strings in that encoding. In Squeak encoding, each character is represented by an instance of `Character`. If you have non-Latin-1 characters, you'll end up with instances of `WideString`. If all your Characters are in Latin-1, you'll have `ByteStrings`.

WAKomEncoded. `WAKomEncoded` takes one or more bytes of UTF-8 and maps them to a single character (and vice versa). This allows it to support all 100,000 characters in Unicode. The following code shows how to start the encoding adapter.

`"Start on a different port from your standard (WAKom) port"`WAKomEncoded startOn: 8081

WAKom. Now what `WAKom` does, is a one to one mapping from bytes to characters. This works fine if and only if your character set has 255 or less entries and your encoding maps one to one. Examples for such combination are ASCII and ISO-8859-1 (latin-1).

If you run a non-encoded web server adapter like `WAKom`, Seaside will give you strings in the encoding of the web page (!) and expect from you strings in the encoding of the web page.

Example. If you have the character `ä` in a UTF-8 encoded page and you run an encoding server adapter like `WAKomEncoded` this character is represented by the Squeak string:

```
String with: (Character value: 16rE4)
```

However if you run an adapter like `WAKom`, the same character `ä` is represented by the string:

```
String with: (Character value: 16rC3) with: (Character value: 16rA4)
```

Yes, that is a string with two Characters! How can this be? Because `ä` (the Unicode character U+00E4) is encoded in UTF-8 with the two byte sequence `0xC3 0xA4` and WAKom does not interpret that, it just serves the two bytes. **Use UTF-8.** Try to use UTF-8 for your external encodings because it supports Unicode. So you can have access to the largest character set. Then use `WAKomEncoded`; this way your internal string will be encoded on WideString. `WAKomEncoded` will do the conversion of the response/answer between WideString and UTF-8. To see if your encoding works, go to http://localhost:8080/tests/alltests and then to the “Encoding” test (select `WAEncodingTest`). There’s a link there to a page with a lot of foreign characters, pick the most foreign text you can find and paste it into the upper input field, submit the field and repeat it for the lower field.

Telling the browser the encoding. So now that you decided which encoding to use and that Seaside will send pages to the browser in that encoding, you will have to tell the browser which encoding you decided to use. Seaside does this automatically for you. Override `charSet` in your session class (the default is `'utf-8'` in Pharo). In Seaside 3.0 this is a configuration setting in the application.

The charset will make sure that the generated html specifies the encodings as shown below.

```
Content-Type:text/html;charset=utf-8

<meta content="text/html;charset=utf-8"
http-equiv="Content-Type"/>
```

Now you should understand a little more about character encodings and how Seaside deals with them. Pay attention that the contents of uploaded files are not encoded even if you use WAKomEncoded. In addition you have to be aware that you may have other parts of your application that will have to deal with such issues: LDAP, Database, Host OS, etc.

CHAPTER 16

Managing Sessions

When a user interacts with a Seaside application for the first time, a new *session* object is automatically instantiated. This instance lasts as long as the user interacts with the application. Eventually, after the user has not interacted with the session for a while, it will time-out – we say that the session *expires*. The session is internally used by Seaside to remember page-views and action callbacks. Most of the time developers don't need to worry about sessions.

In some cases the session can be a good place to keep information that should be available globally. The session is typically used to keep information about the current user or open database connections. For simple applications, you might consider keeping that information within your components. However, if big parts of your code need access to such objects it might be easier to use a custom session class instead.

Having your own session class can be also useful when you need to clean-up external resources upon session expiry, or when you need extra behavior that is performed for every request.

In this chapter you will learn how to access the current session, debug a session, define your own session to implement a simple login, recover from session expiration, and how to define bookmarkable urls.

16.1 Accessing the Current Session

From within your components the current session is always available by sending `self session`. This can happen during the rendering phase or while processing the callbacks: you get the same object in either case. To demonstrate a way to access the current session, quickly add the following code to a rendering method in your application:

```
html anchor
    callback: [ self show: (WAInspector current on: self session) ];
    with: 'Inspect Session'
```

This displays a link that opens a Seaside inspector on the session. Click the link and explore the contents of the active session. To get an inspector within your image you can use the code `self session inspect`. In both cases you should be able to navigate through the object.

In rare cases it might be necessary to access the current session from outside your component tree. Think twice before doing that though: it is considered to be extremely bad coding style to

depend on the session from outside your component tree. Anyway, in some cases it might come in handy. In such a case, you can use the following expressions:

```
WARequestContext value session
```

But again you should avoid accessing the session from outside of the component tree.

Accessing the Session from the Debugger

In older versions of Seaside, session objects could not be inspected from the debugger as normal objects. If you tried to evaluate `self session` the debugger would answer `nil` instead of the expected session object. This is because sessions are only accessible from within your web application process, and the Smalltalk debugger lives somewhere else. In Seaside 3.0 this problem is fixed on most platforms.

If this doesn't work for you, then you need to use a little workaround to access the session from within the debugger. Put the following expression into your code to open an inspector from within the web application and halt the application by opening a debugger:

```
self session inspect; halt
```

16.2 Customizing the Session for Login

We will now implement an extremely simple login facility to show how to use a custom session. We will enhance the `miniInn` application we developed in Chapter and add a login facility.

When a user interacts with a Seaside application for the first time, an instance of the application's session class is created. The class `WASession` is the default session class, but this can be changed for each application, allowing you to store key information on this class. Different parts of the system will then be able to take advantage of the information to offer different services to the user.

We will define our own session class and use it to store user login information. We will add login functionality to our existing component. The login functionality could also be supported by using a task and/or a specific login component. The principle is the same: you use the session to store some data that is accessible from everywhere within the current session.

In our application we want to store whether the user is logged in. Therefore we create a subclass called `InnSession` of the class `WASession` and we will associate such a new session class to our hotel application. We add the instance variable `user` to the session to hold the identity of the user who is currently logged in.

```
WASession subclass: #InnSession
   instanceVariableNames: 'user'
   classVariableNames: ''
   package: 'SeasideBook'
```

We define some utility methods to query the user login information.

```
InnSession >> login: aString
    user := aString
```

```
InnSession >> logout
    user := nil
```

```
InnSession >> isLoggedIn
    ^ user isNil not
```

```
InnSession >> user
    ^ user
```

Now you need to associate the session we just created with your existing application; you can either use the configuration panel or register the new application setup programmatically.

Configuration Panel. To access the configuration panel of your application go to http://localhost:8080/config/. In the list select your application (probably called 'miniinn') and click on its associated *configure* link. You should get to the configuration panel which lists several things such as: the library your application uses (see **??**); and its general configuration such as its root component (see **??**).

Click on the drop-down list by *Session Class* – if there is only text here, press the *override* link first . Among the choices you should find the class `InnSession`. Select it and you should get the result shown in 16-1. Now *Save* your changes.

Figure 16-1 The session of miniInn is now InnSession.

Configuring the application programmatically. To change the associated session of an application, we can set the preference `#sessionClass` using the message `WASession>>preferencesAt:put:`. We can do that by redefining the *class* `initialize` method of the application as follows. Since this method is invoked automatically only when the application is loaded, make sure that you evaluate it manually after changing it.

```
MiniInn class >> initialize
    | application |
    application := WAAdmin register: self asApplicationAt: 'miniInn'.
    application preferenceAt: #sessionClass put: InnSession
```

To access the current session use the message `WAComponent>>session`. We define the methods `login` and `logout` in our component.

```
MiniInn >> login
    self session login: (self request: 'Enter your name:')
```

```
MiniInn >> logout
    self session logout
```

Then we define the method `renderLogin:` which, depending on the session state, offers the possibility to either login or logout.

```
MiniInn >> renderLogin: html
    self session isLoggedIn
        ifTrue: [
            html text: 'Logged in as: ' , self session user , ' '.
            html anchor
```

```
                callback: [ self logout ];
                with: 'Logout' ]
        ifFalse: [
            html anchor
                callback: [ self login ];
                with: 'Login']
```

We define a dummy method `renderSpecialPrice:` to demonstrate behavior only available for users that are logged in.

```
MiniInn >> renderSpecialPrice: html
    html text: 'Dear ' , self session user, ', you can benefit from
    our special prices!'
```

Then we redefine the method `renderContentOn:` to present the new functionality.

```
MiniInn>>renderContentOn: html
    self renderLogin: html.
    html heading: 'Starting date'.
    html render: calendar1.
    startDate isNil
        ifFalse: [ html text: 'Selected start: ' , startDate
    asString ].
    html heading: 'Ending date'.
    html render: calendar2.
    (startDate isNil not and: [ endDate isNil not ]) ifTrue: [
         html text: (endDate - startDate) days asString ,
             ' days from ', startDate asString, ' to ', endDate
    asString, ' ' ].
    self session isLoggedIn "<-- Added"
        ifTrue: [ self renderSpecialPrice: html ]
```

Figures 16-2, 16-3 and 16-4 illustrate the behavior we just implemented. The user may log in using the top level link. Once logged in, extra information is available to the user.

16.3 LifeCycle of a Session

It is important to understand the lifecycle of a session to know which hooks to customize.16-5 depicts the lifetime of a session:

1. When the user accesses a Seaside application for the first time a new session instance is created and the root component is instantiated. Seaside sends the message `WAComponent>>initialRequest:` to the active component tree, just before triggering the rendering of the components. Specializing the method `initialRequest:` enables developers to inspect the head fields of the first request to an application, and to parse and restore state if necessary.
2. All subsequent requests are processed the same way. First, Seaside gives the components the ability to process the callbacks that have been defined during the last rendering pass. These callbacks are usually triggered by clicking a link or submitting a form. Then Seaside calls `WAComponent>>updateUrl:` of all visible components. This gives the developer the ability to modify the default URL automatically generated by Seaside. Then Seaside redirects the user to the new URL. This redirect is important, because it avoids processing the callbacks unnecessarily when the user hits the *Back* button. Finally Seaside renders the component tree.

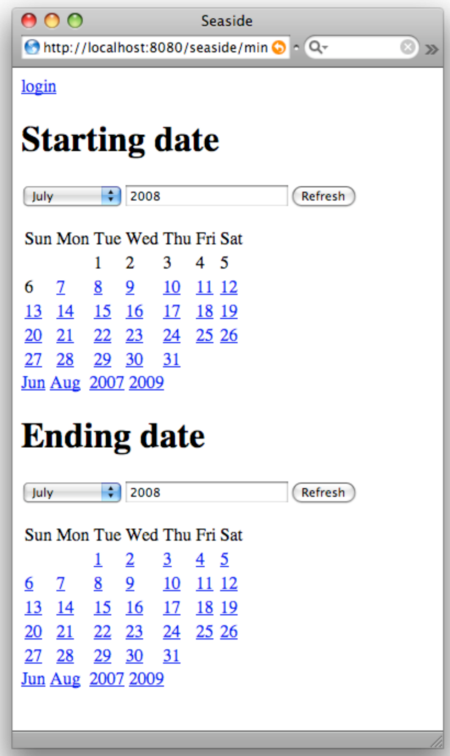


Figure 16-2 With Session.

3. If the session is not used for an extended period of time, Seaside automatically expires it and calls the method `WASession>>unregistered`. If the user bookmarked the application, or comes back to the expired session for another reason, a new session is spawned and the lifecycle of the session starts from the beginning.

16.4 Catching the Session Expiry Notification

Sessions last a certain period of time if there are no requests coming in, after which they expire. The default is 600 seconds or 10 minutes. You can change this value to any other number using the configuration interface, or programmatically using the following expression:

```
"Set the session timeout to 1200 seconds (20 minutes)"
anApplication cache expiryPolicy configuration
    at: #cacheTimeout put: 1200
```

Depending on the type of your application you might want to increase this number. In industrial settings 10 minutes (600 seconds) has shown to be quite practical: it is a good compromise

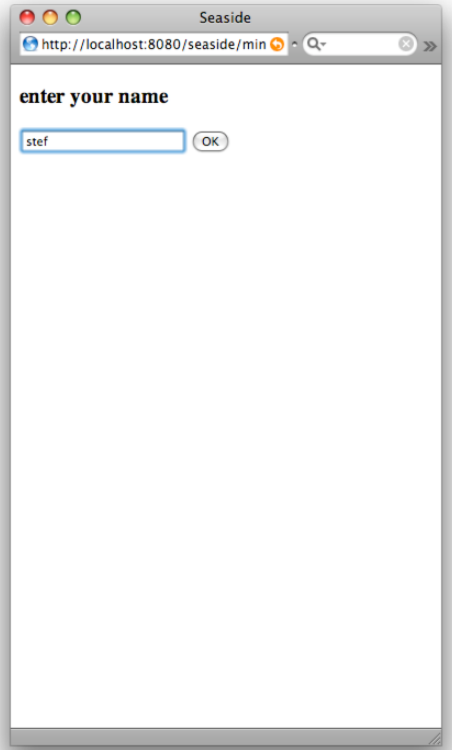


Figure 16-3 With Session: Enter your name.

between user convenience and memory usage.

When a session expires Seaside sends the message `WASession>>unregistered` to `WASession`. You can override this method to clean up your session, for example if you have open files or database connections. In our small example this is not really necessary, but to illustrate the functionally we will now logout the user automatically when the session expires:

```
InnSession >> unregistered
    super unregistered.
    user := nil
```

Note that at the time the message `unregistered` is sent, there is no way to inform the user in the web browser about the session expiry. The message `unregistered` is called asynchronously by the Seaside server thread and there is no open connection that you could use to send something to the client – in fact the user may have already closed the browser window. We will see in the next section how to recover if the user does try to return to the session.

Figure 16-4 With Session: Starting Date and Ending Date.

16.5 Manually Expiring Sessions

In some cases developers might want to expire a session manually. This is useful for example after a user has logged out, as it frees all the memory that was allocated during the session. More important it makes it impossible to use the *Back* button to get into the previously authenticated user-account and do something malicious.

A session can be marked for expiry by sending the message `WASession>>expire` to a `WASession`. Note that calling `expire` will not cause the session to disappear immediately, it is just marked as expired and not accessible from the web anymore. At a later point in time Seaside will call `unregistered` and the garbage collector eventually frees the occupied memory.

Let us apply it to our hotel application: we change our MiniInn application to automatically expire the session when the user logs out.

```
InnSession >> logout
    user := nil.
    self expire
```

Note that expiring a session without redirecting the user to a different location will automati-

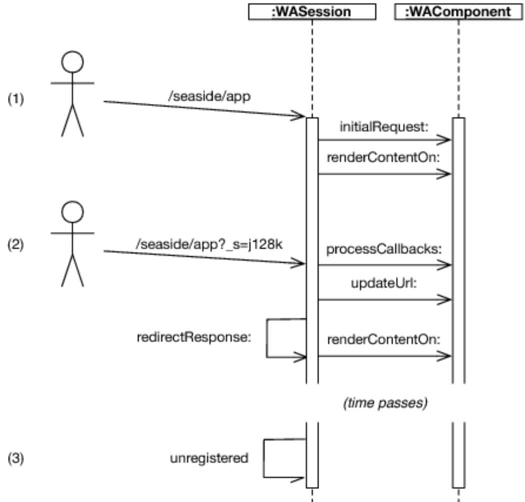


Figure 16-5 Life cycle of a session.

cally start a new session within the same application. Here we change that behavior to make it point to the Seaside web site as follows.

```
InnSession >> logout
    user := nil.
    self expire.
    self redirectTo: 'http://www.seaside.st'
```

If the user tries to get back to the application, he is automatically redirected to a new session.

16.6 Summary

Sessions are Seaside's central mechanism for remembering user specific interaction state. Sessions are identified using the _s parameter in the URL. As an application developer there is normally no need to access or change the session, because it is used internally by Seaside to manage the callbacks and to store the component tree. In certain cases it might be useful to change the behavior of the default implementation or to make information accessible from anywhere in the application.

Pay attention that if components depend on the presence of a specific session class, you introduce strong coupling between the component and the session. Such sessions act as global variables and should not be overused.

CHAPTER 17

A Really Simple Syndication

RSS is a special XML format used to publish frequently updated content, such as blog posts, news items or podcasts. Users don't need to check their favorite web site for updates. Rather, they can subscribe to a URL and be notified about changes automatically. This is can be done using a dedicated tool called *feed reader* or *aggregator*, but most web browsers integrate this capability as part of their core functionality.

The RSS XML format is very much like XHTML, but much simpler. As standardised in the RSS 2.0 Specificationhttp://cyber.law.harvard.edu/rss/rss.html, RSS essentially is composed of two parts, the *channel* and the *news item* specifications. While the channel describes some general properties of the news feed, the items contain the actual stories that change over time. Below we see an example of such a feed. We will see how the same feed is presented within a feed reader.

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
   <channel>
      <title>Seaside ToDo</title>
      <link>http://localhost:8080/todo</link>
      <description>There are always things left to do.</description>
      <item>
         <title>Smalltalk</title>
         <description>(done) 5 March 2008</description>
      </item>
      <item>
         <title>Seaside</title>
         <description>5 September 2008</description>
      </item>
      <item>
         <title>Scriptaculous</title>
         <description>7 September 2008</description>
      </item>
   </channel>
</rss>
```

17.1 Creating a News Feed

There is a Seaside package extension that helps us to build such feeds in a manner similar to what we used to build XHTML for component rendering.

Load the packages RSS-Core, RSS-Examples, and RSS-Tests-Core from the Seaside repository.

```
Metacello new
  githubUser: 'SeasideSt' project: 'Seaside' commitish: 'master'
    path: 'repository';
  baseline: 'Seaside3';
  loads: #('RSS' 'RSS Tests' 'RSS Examples')
```

Let's create a news feed for our todo items.

Define the Feed Component. The package defines a root class named RRComponent that allows you to describe both the news feed channel (title, description, language, date of publication) and also the news items. Therefore, the next step is to create a new subclass of RRComponent named ToDoRssFeed. This will be the entry point of our feed generator. In our example, we don't need extra instance variables.

```
RRComponent subclass: #ToDoRssFeed
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'ToDo-RSS'
```

Register the Component as Entry Point. Next we need to register the component at a fixed URL. The aggregator will use this URL to access the feed. We do this by adding a class side initialize method. Don't forget to evaluate the code.

```
ToDoRssFeed class >> initialize
    (WAAdmin register: RRRssHandler at: 'todo.rss')
        rootComponentClass: self
```

At this point we can begin to download our feed at http://localhost:8080/todo.rss, however it is mostly empty except for some standard markup as shown by the following RSS file. Your browser may be set up to handle RSS feeds automatically, so you may have difficulty in examining the raw source.

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
    <channel>
    </channel>
</rss>
```

17.2 Render the Channel Definition

Next we create the contents of the feed. To do so we need to access our model and pass the data to the RSS renderer. As a first step we render the required tags of the channel element.

```
ToDoRssFeed >> model
    ^ ToDoList default
```

```
ToDoRssFeed >> renderContentOn: rss
    self renderChannelOn: rss
```

```
ToDoRssFeed >> renderChannelOn: rss
    rss title: self model title.
    rss link: 'http://localhost:8080/todo'.
    rss description: 'There are always things left to do.'
```

A full list of all available tags is available in the following table.

RSS Tag	Selector	Description
title	title:	The name of the channel (required).
link	link:	The URL to website corresponding to the channel (required).
description	description:	Phrase or sentence describing the channel (required).
language	language:	The language the channel is written in.
copyright	copyright:	Copyright notice for content in the channel.
managingEditor	managingEditor:	Email address for person responsible for editorial content.
webMaster	webMaster:	Email address for person responsible for technical issues.
pubDate	pubDate:	The publication date for the content in the channel.
lastBuildDate	lastBuildDate:	The last time the content of the channel changed.
category	category:	Specify one or more categories that the channel belongs to.
generator	generator:	A string indicating the program used to generate the channel.

17.3 Rendering News Items

Finally, we want to render the todo items. Each news item is enclosed within a `item` tag. We will display the title and show the due date as part of the description. Also we prepend the string `(done)`, if the item has been completed.

```
ToDoRssFeed >> renderContentOn: rss
    self renderChannelOn: rss.
    self model items
        do: [ :each | self renderItem: each on: rss ]
```

```
ToDoRssFeed >> renderItem: aToDoItem on: rss
    rss item: [
        rss title: aToDoItem title.
        rss description: [
            aToDoItem done
                ifTrue: [ rss text: '(done) ' ].
            rss render: aToDoItem due ] ]
```

Doing so will generate the required XML structure for the item tag.

```
<item>
    <title>Smalltalk</title>
    <description>(done) 5 March 2008</description>
</item>
```

At the minimum, a title or a description must be present. All the other sub-elements are optional.

RSS Tag	Selector	Description
title	title	The title of the item.

link	link	The URL of the item.
description	description	Phrase or sentence describing the channel.
author	author	The item synopsis.
category	category	Includes the item in one or more categories.
comments	comments	URL of a page for comments relating to the item.
enclosure	enclosure	Describes a media object that is attached to the item.
guid	guid	A string that uniquely identifies the item.
pubDate	pubDate	Indicates when the item was published.
source	source	The RSS channel that the item came from.

17.4 Subscribe to the Feed

Now we have done all that is required to let users subscribe. Figure 17-1 shows how the feed is presented to the user in the feed reader when the URL was added manually.

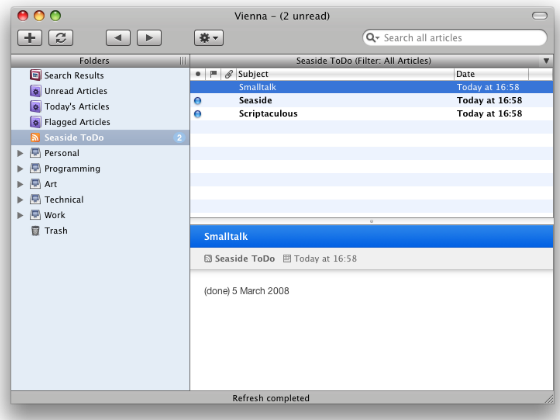


Figure 17-1 The ToDo Feed subscribed.

One remaining thing to do is to tell the users of our todo application where they can subscribe to the RSS feed. Of course we could simply put an anchor at the bottom our web application, however there is a more elegant solution. We override the method `WAComponent>>updateRoot:` in our Seaside component to add a link to our feed into the XHTML head. Most modern web browser will pick up this tag and show the RSS logo in the toolbar to allow people to register for the feed with one click.

```
ToDoListView >> updateRoot: aHtmlRoot
    super updateRoot: aHtmlRoot.
    aHtmlRoot link
        beRss;
        title: self model title;
        url: 'http://localhost:8080/todo.rss'
```

Note the use of the message `beRss` tells the web browser that the given link points to an RSS feed.

In Firefox you may have to add the relationship property to make the rss logo visible.

```
updateRoot: aHtmlRoot
    super updateRoot: aHtmlRoot.
    aHtmlRoot link
        beRss;
        relationship: 'alternate';
        title: self model title;
        url: 'http://localhost:8080/todo.rss'
```

17.5 Summary

In Seaside you don't manipulate tags directly. The elegant generation of RSS feeds nicely shows how the canvas can be extended to produce something other than XHTML. In particular, it is important to see that Seaside is not limited to serve XHTML but can be extended to serve SVG, WAP and RSS.

CHAPTER 18

REST

Seaside is not built around REST services by default, to increase programmer productivity and make to development much more fun. In some cases, it might be necessary to provide a REST API to increase the usability and interoperability of a web application though. Luckily Seaside provides a *Seaside REST* package to fill the gap and to allow one to mix both approaches.

In this chapter, we show how to integrate web applications with Seaside REST services. We start with a short presentation of REST. Then we define a simple REST service for the todo application we implemented in Chapter . We finish this chapter by inspecting how HTTP requests and responses work. We want to thank Olivier Auverlot for providing us with an initial draft of this chapter in French.

18.1 REST in a Nutshell

REST (Representational State Transfer) refers to an architectural model for the design of web services. It was defined by Roy Fielding in his dissertation on Architectural Styles and the Design of Network-based Software Architectureshttp://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. The REST architecture is based on the following simple ideas:

- REST uses URIs to refer to and to access resources.
- REST is built on top of the stateless HTTP 1.1 protocol.
- REST uses HTTP commands to define operations.

This last point is essential in REST architecture. HTTP commands have precise semantics:

- GET lists or retrieves a resource at a given URI.
- PUT replaces or updates a resource at a given URI.
- POST creates a resources at a given URI.
- DELETE removes the resources at a given URI.

Seaside takes a different approach by default: Seaside generates URIs automatically, Seaside keeps state on the server, and Seaside does not interact well with HTTP commands. While the approach of Seaside simplifies a lot of things in web development, sometimes it is necessary to

play with the rules. REST is used by a large number of web products and adhering to the REST standard might increase the usability of an application.

REST applications with Seaside can take two shapes: The first approach creates or extends the interoperability of an existing application by adding a REST API. Web browsers and other client applications can (programmatically) access the functionality and data of an application server, see 18-1.

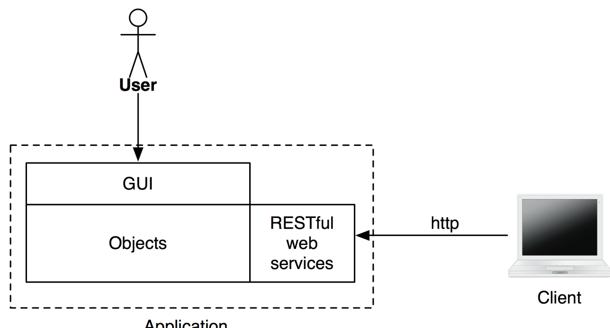


Figure 18-1 First architecture: adding REST to an existing application.

A second approach consists of using REST as the back-end of an application and make it a fundamental element of its architecture. All objects are exposed via REST services to potential clients as well as to the other parts of the application such as its Seaside user-interface, see 18-2.

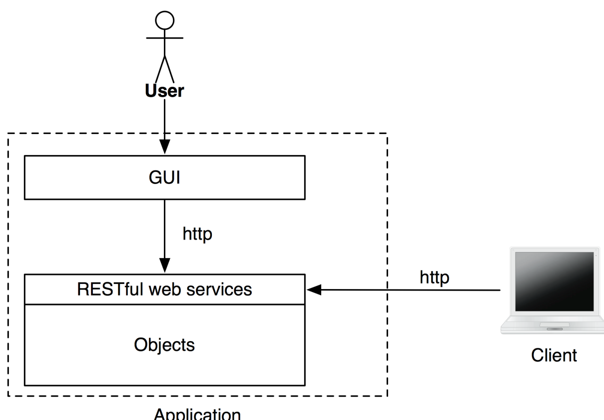


Figure 18-2 Second architecture: REST centric core.

This second approach offers a low coupling and eases deployment. Load-balacing and fail-over mechanisms can easily be put in place and the application can be distributed over multiple machines.

With Seaside and its Rest package you can implement both architectures. In this chapter we are going to look at the first example only, that is we will extend an existing application with a REST API.

18.2 Getting Started with REST

To get started load the package `Seaside-Rest-Core`, and if you are on Pharo `Seaside-Pharo-Rest-Core`. All packages are available from the `Seaside30Addons` repository and you can load then easily with the following Gofer script: check that and update to github and

```
Gofer new
  squeaksource: 'Seaside30Addons';
  package: 'Seaside-REST-Core';
  package: 'Seaside-Pharo-REST-Core';
  package: 'Seaside-Tests-REST-Core';
  load.
```

Recent Seaside images already contain the REST packages preloaded.

Defining a Handler

We are going to extend the todo application from Chapter 13 with a REST API. We will first build a service that returns a textual list of todo items.

Our REST handler, named `ToDoHandler`, should be declared by defining a Seaside class which inherits from `WARestfulHandler`. This way we indicate to Seaside that `ToDoHandler` is a REST handler. The todo items will be accessed through the same model as the existing todo application: `ToDoList default`. This means we do not need to specify additional state in our handler class.

```
WARestfulHandler subclass: #ToDoHandler
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'ToDo-REST'
```

With Seaside-REST, we do not subclass from `WAComponent` that is reserved to the generation of stateful graphical components, but you should subclass from WARestfulHandler. Last we need to initialize our hander by defining a class-side initialization method. We register the handler at the entry point `todo-api` so that it is reachable at http://localhost:8080/todo-api. Don't forget to call the method to make sure the handler is properly registered.

```
ToDoHandler class >> initialize
   WAAdmin register: self at: 'todo-api'
```

Defining a System

The idea behind Seaside-REST is that each HTTP request triggers a method of the appropriate service implementation. All service methods are annotated with specific method annotations or pragmas.

It is possible to define a method that should be executed when the handler receives a GET request by adding the annotation `<get>` to the method. As we will see in , a wide range of other annotations are supported to match other request types, content types, and the elements of the path and query arguments.

To implement our todo service, we merely need to add the following method to `ToDoHandler` that returns the current todo items as a string:

```
ToDoHandler >> list
   <get>

   ^ String streamContents: [ :stream |
      ToDoList default items do: [ :each |
         stream nextPutAll: each title; crlf ] ]
```

The important thing here is the method annotation `<get>`, the name of the method itself does not matter. The annotation declares that the method is associated with any GET request the service receives. Later on we will see how to define handlers for other types of requests.

In a web browser enter the URL http://localhost:8080/todo-api. You should get a file containing the list of existing todo items of your applications. If the file is empty verify that you have some todos on your application by trying it at http://localhost:8080/todo. In case of problems, verify that the server is working using the Seaside Control Panel. If everything works well you should obtain a page with the list of todo items. To verify that our service works as expected we can also use *cURL* or any other HTTP client to inspect the response:

```
$ curl -i curl -i http://localhost:8080/todo-api
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 71
Date: Sun, 20 Nov 2011 17:04:52 GMT
Server: Zinc HTTP Components 1.0

Finish todo app chapter
Annotate first chapter
Discuss cover design
```

By default Seaside tries to convert whatever the method returns into a response. In our initial example this was enough, but in many cases we want more control over how the response is built. We gain full control by asking the *request context* to respond with a custom response. The following re-implementation of the `list` method has the same behavior as the previous one, but creates the response manually.

```
ToDoHandler >> list
   <get>

   self requestContext respond: [ :response |
      ToDoList default items do: [ :each |
         response contentType: 'text/plain'.
         response
            nextPutAll: each title;
            nextPutAll: String crlf ] ]
```

18.3 Matching Requests to Responses

In the initial example we have seen how to define a service that catches all GET requests to the handler. In the following sections we will look at defining more complicated services using more elaborate patterns. In we are going to look at matching other request types, such as POST and PUT. In we are going to see how to serve different content types depending on the requested data. In we will see how to match path elements and in how to extract query parameters.

HTTP Method

Every service method must have a pragma that indicates the HTTP method on which it should be invoked.

If we would like to add a service to create a todo item with a POST request, we could add the following method:

```
ToDoHandler >> create
   <post>

   ToDoList default items
      add: (ToDoItem new
         title: self requestContext request rawBody;
         yourself).
   ^ 'OK'
```

We use the message `rawBody` to access the body of the request. The code creates a new todo item and sets its title. It then replies with a simple `OK` message.

To give our new service a try we could use cURL. With the `-d` option we define the data to be posted to the service:

```
$ curl -d "Give REST a try" http://localhost:8080/todo-api
OK
```

If we list the todo items as implemented in the previous section we should see the newly created entry:

```
$ curl http://localhost:8080/todo-api
Finish todo app chapter
Annotate first chapter
Discuss cover design
Give REST a try
```

Similarly Seaside supports the following request methods:

Request Method	Method Annotation	Description
GET	`<get>`	lists or retrieves a resource
PUT	`<put>`	replaces or updates a resource
POST	`<post>`	creates a resource
DELETE	`<delete>`	removes a resource
MOVE	`<move>`	moves a resource
COPY	`<copy>`	copies a resource

Content Type

Using HTTP and Seaside-REST, we can also specify the format of the data that is requested or sent. To do that we use the `Accept` header of the HTTP request. Depending on it, the REST web service will adapt itself and provide the corresponding data type.

We will take the previous example and we will modify it so that it serves the list of todo items not only as text, but also as JSON or XML. To do so define two new methods named `listJson` and `listXml`. Both methods will be a GET request, but additionally we annotate them with the mime type they produce using `<produces: 'mime-type'>`. This annotation specifies the type of the data returned by the method. A structured format like XML and JSON is friendly to other applications that would like to read the output.

```
ToDoHandler >> listJson
   <get>
   <produces: 'text/json'>

   ^ (Array streamContents: [ :stream |
      ToDoList default items do: [ :each |
         stream nextPut: (Dictionary new
            at: 'title' put: each title;
            at: 'done' put: each done;
            yourself) ] ])
      asJavascript
```

```
ToDoHandler >> listXml
   <get>
   <produces: 'text/xml'>

   ^ WAXmlCanvas builder
      documentClass: WAXmlDocument;
      render: [ :xml |
         xml tag: 'items' with: [
            ToDoList default items do: [ :each |
               xml tag: 'item' with: [
                  xml tag: 'title' with: each title.
                  xml tag: 'due' with: each due ] ] ] ]
```

While in the examples above we (mis)use the JSON and XML builders that come with our Seaside image. You might want to use any other framework or technique to build your output strings.

By specifying the accept-header we can verify that our implementation serves the expected implementations:

```
[{"title": "Finish todo app chapter", "done": false},
 {"title": "Annotate first chapter", "done": true},
 {"title": "Discuss cover design", "done": false},
 {"title": "Give REST a try", "done": true}]
```

```
$ curl -H "Accept: text/xml" http://localhost:8080/todo-api
<items>
  <item>
    <title>Finish todo app chapter</title>
    <done>false</done>
  </item>
  <item>
    ...
```

If the accept-header is missing or unknown, our old textual implementation is called. This illustrates that several methods can get a get annotation and that one is selected and executed depending on the information available in the request. We explain this point later.

Similarly the client can specify the MIME type of data passed to the server using the content-type header. Such behavior only makes sense with PUT and POST requests and is specified using the `<consumes: 'mime-type'>` annotation. The following example states that the data posted to the server is encoded as JSON.

```
ToDoHandler >> createJson
   <post>
   <consumes: '*/json'>

   | json |
   json := JSJsonParser parse: self requestContext request rawBody.
   ToDoList default items
      add: (ToDoItem new
         title: (json at: 'title');
         done: (json at: 'done' ifAbsent: [ false ]);
         yourself).
   ^ 'OK'
```

We can test the implementation with the following cURL query:

```
$ curl -H "Content-Type: text/json" \
   -d '{"title": "Check out latest Seaside"}' \
   http://localhost:8080/todo-api
OK
```

Request Path

URIs are a powerful mechanism to specify hierarchical information. They allow one to specify and access to specific resources. Seaside-Rest offers a number of methods to support the manipulation of URIs. Some predefined methods are invoked by Seaside when you define them in your service.

The method `list` we implemented in is executed when the URI does not contain any access path beside the one of the application.

```
ToDoHandler >> list
   <get>

   ^ String streamContents: [ :stream |
      ToDoList default items do: [ :each |
         stream nextPutAll: each title; crlf ] ]
```

If we define services with methods that expect multiple arguments, the arguments get mapped to the unconsumed path elements. In the example below we use the first path element to identify a todo item by title, and then perform an action on it using the second path element:

```
ToDoHandler >> command: aTitleString action: anActionString
   <get>

   | item |
   item := ToDoList default items
      detect: [ :each | each title = aTitleString ]
      ifNone: [ ^ 'unknown todo item' ].
   anActionString = 'isDone' ifTrue: [
      ^ item done
         ifTrue: [ 'done' ]
         ifFalse: [ 'todo' ] ].
   ...
```

```
    ^ 'invalid command'
```

Now we can query the model like in the following examples:

```
$ curl http://localhost:8080/todo-api/Invalid/isDone
unknown todo item
$ curl http://localhost:8080/todo-api/Discuss+cover+design/isDone
done
$ curl http://localhost:8080/todo-api/Annotate+first+chapter/isDone
todo
```

Query Parameters

So far we used the request type (), the content type () and the request path () to dispatch requests to methods. The last method which is also the most powerful one, is to dispatch on specific path elements and query parameters.

Using the annotation `<path:>` we can define flexible masks to extract elements of an URI. The method containing method is triggered when the path matches verbatim. Variable parts in the path definition are enclosed in curly braces `{aString}` and will be assigned to method arguments. Variable repeated parts in the path definition are enclosed in stars `*anArray*` and will be assigned as an array to method arguments.

The following example implements a search listing for our todo application. Note that the code is almost exactly the same as the one we had in our initial example, except that it filters for the query string:

```
ToDoHandler >> searchFor: aString
   <get>
   <path: '/search?query={aString}'>

   ^ String streamContents: [ :stream |
      ToDoList default items do: [ :each |
         (each title includesSubString: aString)
            ifTrue: [ stream nextPutAll: each title; crlf ] ] ]
```

The method is executed when the client sends a GET request which starts with the path `/search` and contains the query parameter `query`. The expression `{aString}` makes sure that the method argument `aString` is bound to that request argument.

Give it a try on the console. With the right query string only the todo items with the respective substring are printed:

```
$ curl http://localhost:8080/todo-api/search?query=REST
Give REST a try
```

Conflict Resolution

Sometimes there are several methods which Seaside-REST could choose for a request, here's how it finds the "best" one:

1. Exact path matches like `/index.html` take precedence over partial `/index.{var}` or `{var}.html` or wildcard ones `{var}`.
2. Partial path matches like `/index.{var}` or `{var}.html` take precedence over wildcard ones `{var}`.

3. Partial single element matches {var} take precedence over multi element matches *var*.
4. Exact mime type matches like text/xml take precedence over partial */xml or xml/*, wildcard */* and missing ones.
5. Partial mime type matches like */xml or xml/* take precedence over wildcard ones */* or missing ones.
6. If the user agent supplies quality values for the Accept header, then that is taken into account as well.

18.4 Handler and Filter

So far, our REST service did not interact much with the existing Seaside todo application (other than through the shared model). Often it is however desired to have both – the application and the REST services – served from the same URL.

To achieve this we have to subclass WARestfulFilter instead of WARestfulHandler. The WARestfulFilter simply wraps a Seaside application. That is, it handles REST requests exactly as the WARestfulHandler, but it can also delegate to the wrapped Seaside application.

To update our existing service we rename ToDoHandler to ToDoFilter and change its superclass to WARestfulFilter. Now the class definition should look like:

```
WARestfulFilter subclass: #ToDoFilter
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'ToDo-REST'
```

A filter cannot be registered as a an independent entry point anymore, thus we should remove it from the dispatcher to avoid errors:

```
WAAdmin unregister: 'todo-api'
```

Instead we attach the filter the todo application itself. On the class-side of ToDoListView we adapt the initialize method to:

```
ToDoListView class >> initialize
   (WAAdmin register: self asApplicationAt: 'todo')
      addFilter: ToDoFilter new
```

After evaluating the initialization code, the ToDoFilter is now executed whenever somebody accesses our application. The process is visualized in 18-3. Whenever a request hits the filter (1), it processes the annotations (2). Eventually, if none of the annotated methods matched, it delegates to the wrapped application by invoking the method noRouteFound: (3).

Unfortunately – if you followed the creation of the REST API in the previous sections – our #list service hides the application by consuming all requests to http://localhost:8080/todo. We have two possibilities to fix the problem:

1. We remove the method #list so that ToDoFilter automatically calls noRouteFound: that eventually calls the application.
2. We add a new service that captures requests directed at our web application and explicitly dispatches them to the Seaside application. For example, the following code triggers the wrapped application whenever HTML is requested:

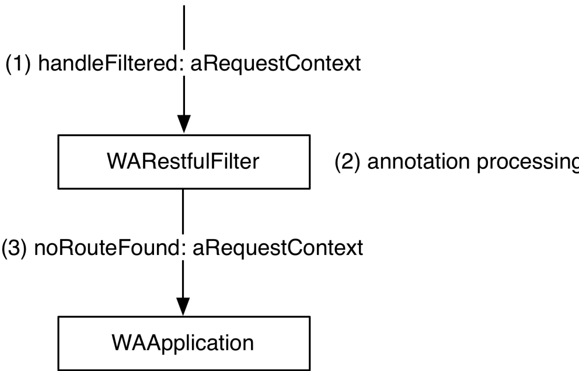


Figure 18-3 Request handling of `WARestfulFilter` and `WAApplication`.

```
ToDoFilter >> app
    <get>
    <produces: 'text/html'>

    ^ self noRouteFound: self requestContext
```

This change leaves the existing API intact and lets users access our web application with their favorite web browser. This works, because browser request documents with the mime-type `text/html` by default. Of course, we can combine this technique with any other of the matching techniques we discussed in the previous chapters.

18.5 Request and Response

Accessing and exploring HTTP requests emitted by a client is an important task during the development of a REST web service. The request gives access to information about the client (IP address, HTTP agent, ...).

To access the request we can add the expression `self requestContext request inspect` anywhere into Seaside code. This works inside a `WAComponent` as well as inside a `WARestfulHandler`.

When the method is executed, you get an inspector on the current request as shown in 18-4.

*linux.png width=70&label=fig:headerhttp*linux.png)

The following example uses the expression `headers at: 'content-type'` that returns the `Content-Type` present in the inspected HTTP client request.

```
ToDoHandler >> list
    <get>

    self requestContext request inspect.
    ^ String streamContents: [ :stream |
        ...
```

In the case of the transmission of a form (corresponding to the application/x-www-form-urlencoded MIME type), we can access the submitted fields using the message `postFields`.

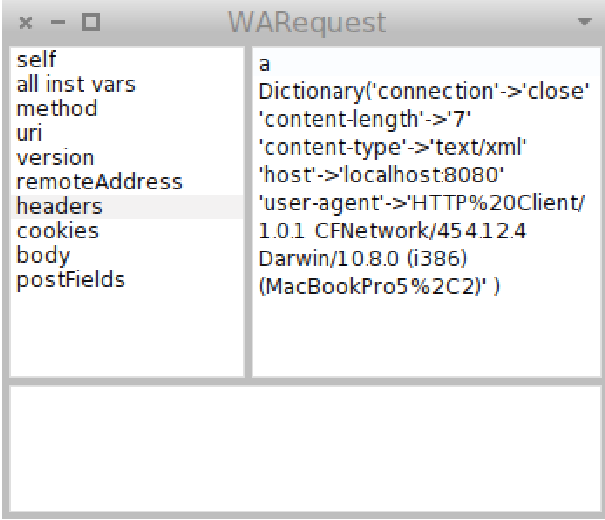


Figure 18-4 Inspecting a request.

It is also possible to customize the HTTP response. A common task is to set the HTTP response code to indicate a result to the client.

For example we could implement a delete operation for our todo items as follows:

```
ToDoHandler >> delete: aString
    <delete>

    | item |
    item := ToDoList default items
       detect: [ :each | each title = aString ]
       ifNone: [ nil ].
    self requestContext respond: [ :response |
       item isNil
         ifTrue: [ response status: WARequest statusNotFound ]
         ifFalse: [
            ToDoList default remove: item.
            response status: WARequest statusOk ] ]
```

The different status codes are implemented on the class side of WARequest. They are grouped in five main families with numbers between 100 and 500. The most common one is status code `WARequest statusOk` (200) and `WARequest statusNotFound` (404).

18.6 Advices and Conclusion

This chapter shows that while Seaside provides a powerful way to build dynamic application using a stateful approach, it can also seamlessly integrate with existing stateless protocols. This chapter illustrated that an object-oriented model of an application in combination with Seaside

is very powerful: You can develop flexible web interfaces as composable Seaside components, and you can easily enrich them with an API for interoperability with REST clients. Seaside provides you with the best of all worlds: the power of object-design, the flexibility and elegance of Seaside components, and the integration of traditional HTTP architectures.

A piece of advice:

- Do not use cookies with a REST service. Such service should respect the stateless philosophie of HTTP. Each request should be independent of others.
- During the development, organize your tagged methods following the HTTP commands: (GET, POST, PUT, DELETE, HEAD). You can use protocols to access them faster.
- A good service web should be able to produce different types of contents depending on the capabilities of the clients. Been able to produce different formats such as plain text (text/plain), XML (text/xml), or JSON (text/json) increases the interoperability of your web services.

You should now have a better understanding of the possibilities offered by Seaside-REST and be ready to produce nice web services.

CHAPTER 19

Deployment

At some point you certainly want to go public with your web application. This means you need to find a server that is publicly reachable and that can host your Seaside application. If your application is successful, you might need to scale it to handle thousands of concurrent users. All this requires some technical knowledge.

In 19.1 we are going to have a look at some best practices before deploying an application. Next, in 19.2 we present how to setup your own server using Apache. Last but not least, in 19.6, we demonstrate ways to maintain a deployed image.

19.1 Preparing for Deployment

Because Pharo offers you an image-based development environment, deploying your application can be as simple as copying your development image to your server of choice. However, this approach has a number of drawbacks. We will review a number of these drawbacks and how to overcome them.

Stripping down your image.

The image you have been working in may have accumulated lots of code and tools that aren't needed for your final application; removing these will give you a smaller, cleaner image. How much you remove will depend on how many support tools you wish to include in your deployed image.

@todo: we should mention the mini image

Alternatively, you may find it easier to copy your application code into a pre-prepared, 'stripped-down' image. For Pharo we have had good experiences using the Pharo Corehttp://www.pharo-project.org/ or Pharo minimal images.

Preparing Seaside.

The first task in preparing Seaside for a server image is to remove all unused applications. To do this go to the *configuration* application at http://localhost:8080/config and click on *remove* for

all the entry points you don't need to be deployed. Especially make sure that you remove (or password protect) the *configuration* application and the *code browser* (at http://localhost:8080/tools/classbrowser), as these tools allow other people to access and potentially execute arbitrary code on your server.

Disable Development Tools.

If you still want the development tools loaded, then the best way is to remove `WADevelopmentConfiguration` from the shared configuration called "Application Defaults". You can do this by evaluating the code:

```
WAAdmin applicationDefaults
    removeParent: WADevelopmentConfiguration instance
```

You can always add it back by evaluating:

```
WAAdmin applicationDefaults
    addParent: WADevelopmentConfiguration instance
```

Alternatively you can use the configuration interface: In the configuration of any application select *Application Defaults* from the list of the *Assigned parents* in the *Inherited Configuration* section and click on *Configure*. This opens an editor on the settings that are common to all registered applications. Remove `WAToolDecoration` from the list of *Root Decoration Classes*.

Password Protection.

If you want to limit access to deployed applications make sure that you password protect them. To password protect an application do the following:

1. Click on *Configure* of the particular entry point.
2. In the section *Inherited Configuration* click select `WAAuthConfiguration` from the drop down box and click on *Add*. This will will add the authentication settings below.
3. Set *login* and *password* in the *Configuration* section below.
4. Click on *Save*.

If you want to programmatically change the password of the Seaside configure application, adapt and execute the following code:

```
| application |
application := WADispatcher default handlerAt: 'config'.
application configuration
    addParent: WAAuthConfiguration instance.
application
    preferenceAt: #login put: 'admin';
    preferenceAt: #passwordHash put: (GRPlatform current
     secureHashFor: 'seaside').
application
    addFilter: WAAuthenticationFilter new.
```

Alternatively you can use the method `WAAdmin>>register:asApplicationAt:user:password:` to do all that for you when registering the application:

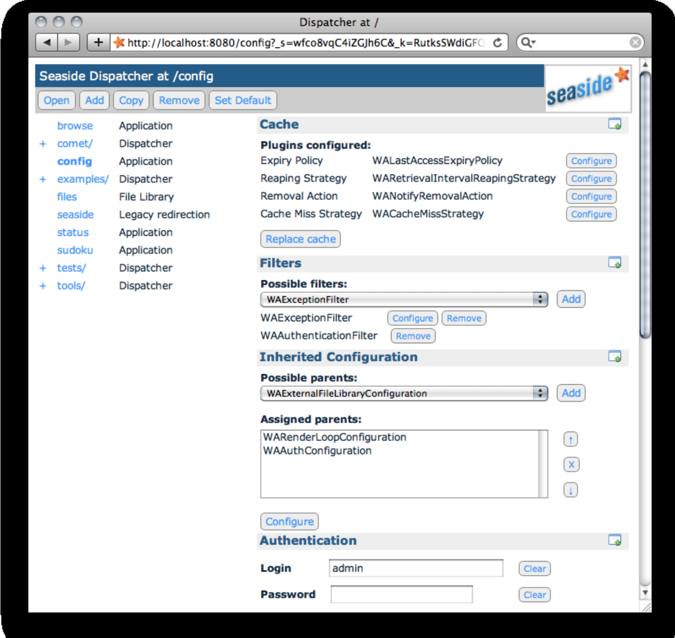


Figure 19-1 Configure an application for deployment.

```
WAConfigurationTool class >> initialize
    WAAdmin register: self asApplicationAt: 'config' user: 'admin'
    password: 'seaside'
```

Next we have a look at the configuration settings relevant for deployment. Click on *Configure* of the application you are going to deploy. If you don't understand all the settings described here, don't worry, everything will become clearer in the course of the following sections.

Resource Base URL.

This defines the URL prefix for URLs created with `WAAnchorTag>>resourceUrl:`. This setting avoids you having to duplicate the base-path for URLs to resource files all over your application. You will find this setting useful if you host your static files on a different machine than the application itself or if you want to quickly change the resources depending on your deployment scenario.

As an example, let's have a look at the following rendering code: `html image resourceUrl: 'logo-plain.png'`. If the resource base URL setting is set to http://www.seaside.st/styles/, this will point to the image at http://www.seaside.st/styles/logo-plain.png. Note that this setting only affects URLs created with `WAImageTag>>resourceUrl:`, it does not affect the generated pages and URLs otherwise.

Set it programmatically with:

```
application
    preferenceAt: #resourceBaseUrl
    put: 'http://www.seaside.st/resources/'
```

Server Protocol, Hostname, Port and Base Path.

Seaside creates absolute URLs by default. This is necessary to properly implement HTTP redirects. To be able to know what absolute path to generate, Seaside needs some additional information and this is what these settings are about. These settings will be useful if you are deploying Seaside behind an external front-end web-server like Apache.

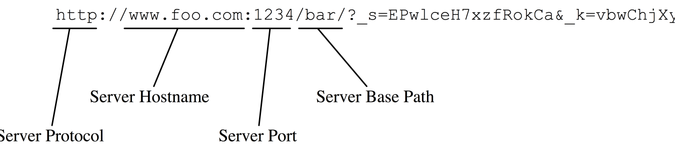


Figure 19-2 Configuration options for absolute URLs.

Have a look at 19-2 to see visually how these settings affect the URL. *Server Protocol* lets you change between `http` and `https` (Secure HTTP). Note that this changes only the way the URL is generated, it does not implement HTTPS – if you want secure HTTP you need to pass the requests through a HTTPS proxy. *Server Hostname* and *Server Port* define the hostname and port respectively. In most setups, you can leave these settings undefined, as Seaside is able to figure out the correct preferences itself. If you are using an older web server such as Apache 1 you have to give the appropriate values here. *Server Path* defines the URL prefix that is used in the URLs. Again, this only affects how the URL is generated, it does not change the lookup of the application in Seaside. This setting is useful if you want to closely integrate your application into an existing web site or if you want to get rid or change the prefix `/appname` of your applications.

Again, if you want to script the deployment, adapt and execute the following code:

```
application
    preferenceAt: #serverProtocol put: 'http';
    preferenceAt: #serverHostname put: 'localhost';
    preferenceAt: #serverPort put: 8080;
    preferenceAt: #serverPath put: '/'
```

19.2 Deployment with Apache

In this section we discuss a typical server setup for Seaside using Debian Linux as operating system. Even if you are not on a Unix system, you might want to continue reading, as the basic principles are the same everywhere. Due to the deviation of different Linux and Apache distributions, the instructions given here cannot replace the documentation for your particular target system.

Preparing the server

Before getting started you need a server machine. This is a computer with a static IP address that is always connected to the Internet. It is not required that you have physical access to your machine. You might well decide to host your application on a virtual private server (VPS). It is

important to note that you require superuser-level access to be able to run Smalltalk images. The ability to execute PHP scripts is not enough.

We assume that your server already has a working Linux installation. In the following sections we use Debian Linux 5.0 (Lenny), however with minor adjustments you will also be able to deploy applications on other distributions.

Before starting with the setup of your application, it is important to make sure that your server software is up-to-date. It is crucial that you always keep your server up to the latest version to prevent malicious attacks and well-known bugs in the software.

To update your server execute the following commands from a terminal. Most commands we use in this chapter require administrative privileges, therefore we prepend them with sudo (super user do):

```
$ sudo apt-get update
$ sudo apt-get upgrade
Reading package lists... Done
Building dependency tree... Done
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

Installing Apache

Next we install Apache 2.2, an industry leading open-source web server. Depending on your requirements you might decide to install a different web server. Lighttpd, for example, might be better suited in a high performance environment.

Some people prefer to use one of the web servers written in Smalltalk. This is a good choice during development and prototyping, as such a server is easy to setup and maintain. We strongly discourage to use such a setup for production applications due to the following reasons:

- The web server is something accessible from outside the world and therefore exposed to malicious attacks. Apache is a proven industry standard used by more than 50% (depending on the survey) of all of today's web sites.
- To listen on port 80, the standard port used by the HTTP protocol, the web server needs to run as root. Running a public service as root is a huge security issue. Dedicated web servers such as Apache drop their root privileges after startup. This allows them to listen to port 80 while not being root. Unfortunately this is not something that can be easily done from within the Smalltalk VM.
- Smalltalk is relatively slow when reading files and processing large amounts of data (the fact that everything is an object is rather a disadvantage in this case). A web server running natively on the host platform is always faster by an order of magnitude. A standalone web server can take advantages of the underlying operating system and advise it to directly stream data from the file-system to the socket as efficiently as possible. Furthermore web servers usually provide highly efficient caching strategies.
- Most of today's Smalltalk systems (with the exception of GemStone) are single threaded. This means that when your image is serving files, Seaside is blocked and cannot produce dynamic content at the same time. On most of today's multi-core systems you get much better performance when serving static files through Apache running in parallel to your Seaside application server.
- External web servers integrate well with the rest of the world. Your web application might need to integrate into an existing site. Often a web site consists of static as well as dynamic content provided by different technologies. The seamingless integration of all these technologies is simple with Apache.

Let's go and install Apache then. If you are running an older version you might want to consider upgrading, as it makes the integration with Seaside considerably simpler, although it is not strictly necessary.

```
$ sudo apt-get install apache2
```

Ensure the server is running and make it come up automatically when the machine boots:

```
$ sudo apache2 -k restart
$ sudo update-rc.d apache2 defaults
```

Installing the VM

update Depending on the Smalltalk dialect you are using, the installation of the VM is different. Installing Pharo on a Debian system is simple. Install Pharo by entering the following command on the terminal:

```
$ sudo apt-get install squeak-vm
```

Note that installing and running Squeak does not require you to have the *X Window System* installed. Just tell the installer not to pull these dependencies in, when you are asked for it. Squeak remains runnable headless without a user-interface, this is what you want to do on most servers anyway. Up-to-date information on the status of the Squeak VM you find at http://www.squeakvm.org/unix/.

Now you should be able to start the VM. Typing the `squeak` command executes a helper script that allows one to install new images and sources in the current directory, and run the VM. The VM itself can be started using the `squeakvm` command.

```
$ squeakvm -help
$ squeakvm -vm-display-null imagename.image
```

You can find additional help on starting the VM and the possible command line parameters in the man pages:

```
$ man squeak
```

In the next section we are going to look at how we can run the VM as a daemon.

Running the VM

Before we hook up the Pharo side with the web server, we need a reliable way to start and keep the Smalltalk images running as daemon (background process). We have had positive experience using the *daemontools*, a free collection of tools for managing UNIX service written by Daniel J. Bernstein. Contrary to other tools like `inittab`, `ttys`, `init.d`, or `rc.local`, the `daemontools` are reliable and easy to use. Adding a new service means linking a directory with a script that runs your VM into a centralized place. Removing the service means removing the linked directory.

Type the following command to install daemontools. Please refer to official website (http://cr.yp.to/daemontools.html) for additional information.

```
$ apt-get install daemontools-run
```

On the server create a new folder to carry all the files of your Seaside application. We usually put this folder into a subdirectory of `/srv` and name it according to our application `/srv/appname`, but this is up to you. Copy the deployment image you prepared in into that directory. Next we create a `run` script in the same directory:

```
#!/bin/bash

# settings
USER="www-data"
VM="/usr/bin/squeakvm"
VM_PARAMS="-mmap 256m -vm-sound-null -vm-display-null"
IMAGE="seaside.image"

# start the vm
exec \
    setuidgid "$USER" \
    "$VM" $VM_PARAMS "$IMAGE"
```

On lines 3 to 7 we define some generic settings. `$USER` is the user of the system that should run your Smalltalk image. If you don't set a user here, the web service will run as root, something you must avoid. Make sure you have the user specified here on your system. `www-data` is the default user for web services on Debian systems. Make sure that this is not a user with root privileges. `$VM` defines the full path to the Squeak VM. If you have a different installation or Smalltalk dialect, you again need to adapt this setting. `$VM_PARAMS` defines the parameters passed to the Squeak VM. If you use a different environment, you need to consult the documentation and adapt these parameters accordingly. The first parameter `-mmap 256m` limits the dynamic heap size of the Squeak VM to 256 MB and makes Squeak VMs run more stably. `-vm-sound-null` disables the sound plugin, something we certainly don't need on the server. `-vm-display-null` makes the VM run headless. This is crucial on our server, as we presumably don't have any windowing server installed.

The last four lines of the script actually start the VM with the given parameters. Line 11 changes the user id of the Squeak VM, and line 12 actually runs the Squeak VM.

To test the script mark it as executable and run it from the command line. You need to do this as superuser, otherwise you will get an error when trying to change the user id of the VM.

```
$ chmod +x ./run
$ sudo ./run
```

The VM should be running now. You can verify that by using on of the UNIX console tools. In the following example we assume that the image has a web server installed and is listening on port 8080:

```
$ curl http://localhost:8080
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"><html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head><title>Dispatcher at /</title>
...
```

You might want to change the URL to point to your application. As long as you don't get an error message like `curl: (7) couldn't connect to host` everything is fine. You can install curl using

```
apt-get install curl
```

Troubleshooting the VM.

You may encounter some common problems at this point. One of these problems is that the VM is unable to find or read the image, change or source files. Make sure that all these files are in

the same directory and that their permissions are correctly set so that the user `www-data` can actually read them. Also ensure that the image has been saved with the web server running on the correct port. Pharo displays an error message when the `.changes` or `.sources` file cannot be found or accessed. Unfortunately this message is not visible from the console, but only pops up virtually in the headless window of the VM. The modal dialog prevents the VM from starting up the server and thus the image remains unreachable. To solve the problem make sure that `.changes` and `.sources` files are in the same directory as the image-file and that all files can be read by the user `www-data`. Another possibility (if you really do not want to distribute the files) is to disable the missing files warning using:

```
Preferences disable: #warnIfNoSourcesFile
```

A valuable tool to further troubleshoot a running but otherwise not responding VM is `lsof`, an utility that lists opened files and sockets by process. You can install it using `apt-get install lsof`. Use a different terminal to type the following commands:

```
$ ps -A | grep squeakvm
  22315 ?        00:17:49 squeakvm
$ lsof -p 22315 | grep LISTEN
squeakvm 22315 www-data  7u  IPv4 411140468  TCP *:webcache (LISTEN)
squeakvm 22315 www-data  8u  IPv4 409571873  TCP *:5900 (LISTEN)
```

With the first line, we find out the process id of the running Squeak VM. `lsof -p 22315` lists all the open files and sockets of process `22315`. Since we are only interested in the sockets Squeak is listening on we grep for the string `LISTEN`. In this case we see that a web server is correctly listening on port 8080 (webcache) and that another service is listening on port 5900. In fact the latter is the RFB server, which we will discuss in .

In the original terminal, press Ctrl+C to stop the VM.

Starting the service.

Go to `/service` and link the directory with your run-script in there, to let *daemontools* automatically start your image.

Go to `/etc/service` (on other distributions this might be `/service`) and link the directory with your run-script in there, to let *daemontools* automatically start your image.

```
$ cd /service
$ ln -s /srv/appname .
```

You can do an `svstat` to see if the new service is running correctly:

```
$ svstat *
appname: up (pid 4165) 8 seconds
```

The output of the command tells you that the service `appname` is up and running for 8 seconds with process id 4165. From now on *daemontools* makes sure that the image is running all the time. The image gets started automatically when the machine boots and – should it crash – it is immediately restarted.

Stopping the service.

To stop the image unlink the directory from `/service` and terminate the service.

[[[$ rm /etc/service/appname $ cd /srv/appname $ svc -t .]]]

Note that only unlinking the service doesn't stop it from running, it is just taking it away from the pool of services. Also note that terminating the service without first unlinking it from the service directory will cause it to restart immediately. *daemontools* is very strict on that and will try to keep all services running all the time.

As you have seen starting and stopping an image with *daemontools* is simple and can be easily scripted with a few shell scripts.

Configuring Apache

Now we have all the pieces to run our application. The last remaining thing to do is to get Apache configured correctly. In most UNIX distributions this is done by modifying or adding configuration files to `/etc/apache2`. On older systems the configuration might also be in a directory named `/etc/httpd`.

The main configuration file is situated in `apache2.conf`, or `httpd.conf` on older systems. Before we change the configuration of the server and add the Seaside web application, have a look at this file. It is usually instructive to see how the default configuration looks like and there is plenty of documentation in the configuration file itself. On most systems this main configuration file includes other configuration files that specify what modules (plug-ins) are loaded and what sites are served through the web server.

Loading Modules.

On Debian the directory `/etc/apache2/mods-available` contains all the available modules that could be loaded. To make them actually available from your configuration you have to link (`ln`) the `.load` files to `/etc/apache2/mods-enabled`. We need to do that for the proxy and rewrite modules:

```
$ a2enmod proxy
$ a2enmod proxy_http
$ a2enmod rewrite
```

If you are running an older version you might need to uncomment some lines in the main configuration file to add these modules.

Adding a new site.

Next we add a new site. The procedure here is very similar to the one of the modules, except that we have to write the configuration file ourselves. `/etc/apache2/sites-available/` contains configuration directives files of different virtual hosts that might be used with Apache 2. `/etc/apache2/sites-enabled/` contains links to the sites in `sites-enabled/` that the administrator wishes to enable.

Step 1. In `/etc/apache2/sites-available/` create a new configuration file called `appname.conf` as follow.

```
<VirtualHost *>
    # set server name
    ProxyPreserveHost On
    ServerName www.appname.com

    # rewrite incoming requests
    RewriteEngine On
    RewriteRule ^/(.*)$ http://localhost:8080/appname/$1 [proxy,last]
```

```
</VirtualHost>
```

The file defines a default virtual host. If you want to have different virtual hosts (or domain names) to be served from the same computer replace the `*` in the first line with your domain name. The domain name that should be used to generate absolute URLs is specified with `ServerName`. The setting `ProxyPreserveHost` enables Seaside to figure out the server name automatically, but this setting is not available for versions prior to Apache 2. In this case you have to change the Seaside preference 'hostname' for all your applications manually, see

`RewriteEngine` enables the rewrite engine that is used in the line below. The last line actually does all the magic and passes on the request to Seaside. A rewrite rule always consists of 3 parts. The first part matches the URL, in this case all URLs are matched. The second part defines how the URL is transformed. In this case this is the URL that you would use locally when accessing the application. And finally, the last part in square brackets defines the actions to be taken with the transformed URL. In this case we want to proxy the request, this means it should be passed to Squeak using the new URL. We also tell Apache that this is the last rule to be used and no further processing should be done.

Step 2. Now link the file from `/etc/apache2/sites-enabled/` and restart Apache:

```
$ cd /etc/apache2/sites-enabled
$ ln -s /etc/apache2/sites-available/appname.conf .
$ sudo apache2ctl restart
```

If the domain name `appname.com` is correctly setup and pointing to your machine, everything should be up and running now. Make sure that you set the 'server base path' in the application configuration to `/`, so that Seaside creates correct URLs.

Troubleshooting the Proxy. On some systems the above configuration may not suffice to get Seaside working. Check the error_log and if you get an error messages in the Apache log saying `client denied by server configuration` just remove the file `/etc/mods-enabled/proxy.conf` from the configuration.

19.3 Serving File with Apache

Most web applications consist of serving static files at one point or the other. These are all files that don't change over time, as opposed to the XHTML of the web applications. Common static files are style sheets, Javascript code, images, videos, sound or simply document files. As we have seen in Chapter such files can be easily served through the image, however the same drawbacks apply here as those listed in .

The simplest way to use an external file server is to overlay a directory tree on the hard disk of the web server over the Seaside application. For example, when someone requests the file http://www.appname.com/seaside.png the server serves the file `/srv/appname/web/seaside.png`. With Apache this can be done with a few additional statements in the configuration file:

```
<VirtualHost *>

    # set server name
    ProxyPreserveHost On
    ServerName www.appname.com

    # configure static file serving
    DocumentRoot /srv/appname/web
    <Directory /srv/appname/web>
        Order deny,allow
        Allow from all
```

```
    </Directory>

    # rewrite incoming requests
    RewriteEngine On
    RewriteCond /srv/appname/web%{REQUEST_FILENAME} !-f
    RewriteRule ^/(.*)$ http://localhost:8080/appname/$1 [proxy,last]

</VirtualHost>
```

The added part starts line 9 with the comment `#configure static file serving`. Line 9 and following mark a location on the local harddisc to be used as the source of files. So when someone requests the file http://www.appname.com/seaside.png Apache will try to serve the file found at `/srv/appname/web/seaside.png`. For security reasons, the default Apache setup forbids serving any files from the local hard disk, even if the filesystem permissions allow the process to access these files. With the lines between `Directory` and `Directory` we specify that Apache can serve all files within `/srv/appname/web`. There are many more configuration options available there, so check out the Apache documentation.

The next thing we have to do is add a condition in front of our rewrite rule. As you certainly remember, this rewrite rule passes (proxies) all incoming requests to Seaside. Now, we would only like to do this if the requested file does not exist on the file-system. To take the previous example again, if somebody requests http://www.appname.com/seaside.png Apache should check if a file named `/srv/appname/web/seaside.png` exists. This is the meaning of the line 16 where `%{REQUEST_FILENAME}` is a variable representing the file looked up, here the variable `REQUEST_FILENAME` is bound to `seaside.png`. Furthermore, the cryptic expression `!-f` means that the following rewrite rule should conditionally be executed *if the file specified does not exist*. In our case, assuming the file `/srv/appname/web/seaside.png` exists, this means that the rewrite rule is skipped and Apache does the default request handling. Which is to serve the static files as specified with the `DocumentRoot` directive. Most other requests, assuming that there are only a few files in `/srv/appname/web`, are passed on to Seaside.

A typical layout of the directory `/srv/appname/web` might look like this:

`favicon.ico`	A shortcut icon, which most graphical web browsers automatically make use of. The icon
`robots.txt`	A robots exclusion standard, which most search engines request to get information on wl
`resources/`	A subdirectory of resources used by the graphical designer of your application.
`resources/css/`	All the CSS resources of your application.
`resources/script/`	All the external Javascript files of your application.

19.4 Load Balancing Multiple Images

There are several ways to load balance Seaside images, one common way is to use mod_proxy_balancerhttp://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html*proxy*balancer.html) that comes with Apache. Compared to other solutions it is relatively easy to set up, does not modify the response and thus has no performance impact. It requires to only load one small additional package to load into Seaside. Additionally `mod_proxy_balancer` provides some advanced features like a manager application, pluggable scheduler algorithms and configurable load factors.

When load balancing multiple Seaside images care must be taken that all requests that require a particular session are processed by the same image, because unless GemStone is used session do not travel between images. This has to work with and without session cookies. This is referred to as sticky sessions because a session sticks to its unique image. `mod_proxy_balancer` does this by associating each image with an route name. Seaside has to append this route to the session id. `mod_proxy_balancer` reads the route from the request and proxies to the appropriate image. This has the advantage that `mod_proxy_balancer` does not have to keep track of all the sessions and does not have to modify the response. Additionally the mapping is defined statically in the the Apache configuration and is not affected by server restarts.

First we need to define our cluster of images. In this example we use two images, the first one on port 8881 with the route name "first" the second on port 8882 with route name "second". We can of course choose other ports and route names as long as they are unique:

```
<Proxy balancer://mycluster>
   BalancerMember http://127.0.0.1:8881 route=first
   BalancerMember http://127.0.0.1:8882 route=second
</Proxy>
```

Next we need to define the actual proxy configuration, which is similar to what we do with a single Seaside image behind an Apache:

```
ProxyPass / balancer://mycluster/ stickysession=_s|_s nofailover=On
=ProxyPassReverse / http://127.0.0.1:8881/
=ProxyPassReverse / http://127.0.0.1:8882/
```

Note that we configure _s to be the session id for the URL and the cookie.

Finally, we can optionally add the balancer manager application:

```
ProxyPass /balancer-manager !
<Location /balancer-manager>
   SetHandler balancer-manager
</Location>
```

As well as an optional application displaying the server status:

```
ProxyPass /server-status !
<Location /server-status>
   SetHandler server-status
</Location>
```

Putting all the parts together this gives an apache configuration like the following:

```
<VirtualHost *>

   ProxyRequests Off
   ProxyStatus On
   ProxyPreserveHost On
   ProxyPass /balancer-manager !
   ProxyPass /server-status !
   ProxyPass / balancer://mycluster/ STICKYSESSION=_s|_s
   ProxyPassReverse / http://127.0.0.1:8881/
   ProxyPassReverse / http://127.0.0.1:8882/

   <Proxy balancer://mycluster>
      BalancerMember http://127.0.0.1:8881 route=first
      BalancerMember http://127.0.0.1:8882 route=second
   </Proxy>

   <Location /balancer-manager>
      SetHandler balancer-manager
   </Location>

   <Location /server-status>
```

```
    SetHandler server-status
  </Location>

</VirtualHost>
```

The rest can be configured from within Seaside. First we need to load the package `Seaside-Cluster` from http://www.squeaksource.com/ajp/. Then we need to configure each image individually with the correct route name. It is important that this matches the Apache configuration from above. The easiest way to do this is evaluate the expression:

```
WAAdmin makeAllClusteredWith: 'first'
```

in the image on port 8881 and

```
WAAdmin makeAllClusteredWith: 'second'
```

in the image on port 8882.

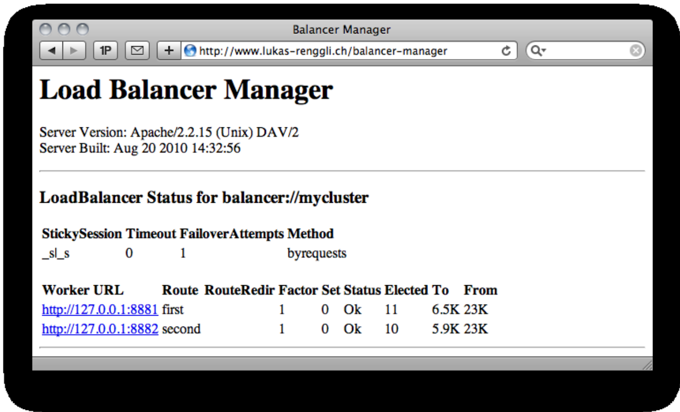


Figure 19-3 Apache Load Balancer Manager.

This is it, the cluster is ready to go. If enabled the server manager can be found at http://localhost/balancer-manager, see Figure 19-3, and the server status can be queried on http://localhost/server-status. Note that before going to production these two admin applications need to be properly protected from unauthorized access.

19.5 Using AJP

AJPv13 is a binary protocol between Apache and Seaside. It was originally developed for Java Tomcat but there is nothing Java specific about it. Compared to conventional HTTP it has less overhead and better support for SSL.

Starting with version Apache 2.2 the required module mod_proxy_ajphttp://httpd.apache.org/docs/2.2/mod/mod_proxy_ajp.html*proxy*ajp.html) is included with the default setup making it much simpler to use. The configuration looks almost the same as the one we saw in Section . The only difference is that you need to replace `proxy_http` with `proxy_ajp`, and that the protocol in the URL of the rewrite rule is `ajp` instead of `http`.

The adapted configuration looks like this:

```
<VirtualHost *>

    # set server name
    ProxyPreserveHost On
    ServerName www.appname.com

    # rewrite incoming requests
    RewriteEngine On
    RewriteRule ^/(.*)$ ajp://localhost:8003/appname/$1 [proxy,last]

</VirtualHost>
```

On the Smalltalk side you need to load the packages `AJP-Core` and `AJP-Pharo-Core` directly with Monticello from http://www.squeaksource.com/ajp. More conveniently you can also use the following Metacello script:

```
Gofer new
    squeaksource: 'MetacelloRepository';
    package: 'ConfigurationOfAjp';
    load.
(Smalltalk globals at: #ConfigurationOfAjp)
    project latestVersion load: 'AJP-Core'
```

At that point you need to add and start the `AJPPharoAdaptor` on the correct port from within your image (8003 in this example) and your web application is up and running with AJP.

19.6 Maintaining Deployed Images

If you followed all the instructions up to now, you should have a working Seaside server based on Seaside, Apache and some other tools. Apache is handling the file requests and passing on other requests to your Seaside application server. This setup is straightforward and enough for smaller productive applications.

As your web application becomes widely used, you want to regularly provide fixes and new features to your users. Also you might want to investigate and debug the deployed server. To do that, you need a way to get our hands on the running VM. There are several possibilities to do that, we are going to look at those in this section.

Headful Pharo

Instead of running the VM headless as we did previously, it is also possible to run it headful as you do during development. This is common practice on Windows servers, but it is rarely done on Unix. Normally servers doesn't come with a windowing system for performance and security reasons. Managing a headful image is straightforward, so we will not be discussing this case further.

VNC

A common technique is to run a VNC server within your deployed image. VNC (Virtual Network Computing) is a graphical desktop sharing system, which allows one to visually control another computer. The server constantly sends graphical screen updates through the network using a

remote frame buffer (RFB) protocol, and the client sends back keyboard and mouse events. VNC is platform independent and there are several open-source server and client implementations available.

Pharo comes with a VNC client and server implementation, which can optionally be loaded. It is called *Remote Frame Buffer (RFB)*. Unfortunately the project is not officially maintained anymore and the latest code is broken in Pharo, however you can get a working version from http://source.lukas-renggli.ch/unsorted/. Update Install the RFB package, define a password and start the server. Now you are able to connect to the Pharo screen using any VNC client. Either using the built-in client from a different Pharo image, or more likely using any other native client. Now you are able to connect to the server image from anywhere in the world, and this even works if the image is started headless. This is very useful to be able to directly interact with server images, for example to update code or investigate and fix a problem in the running image.

19.7 Deployment tools

Seaside comes with several tools included that help you with the management of deployed applications. The tools included with the Pharo distribution of Seaside includes:

Configuration	http://localhost:8080/config
System Status	http://localhost:8080/status
Class Browser	http://localhost:8080/tools/classbrowser
Screenshot	http://localhost:8080/tools/screenshot
Version Uploader	http://localhost:8080/tools/versionuploader

Configuration. The Seaside configuration interface is described throughout this book, especially in the previous sections so we are not going to discuss this further.

System Status. The System status is a tool that provides useful information on the system. It includes information on the image (see 19-4), the virtual machine, Seaside, the garbage collector and the running processes.

Figure 19-4 System Status Tool.

Class Browser. The class browser provides access the source code of the system and allows you to edit any method or class while your application is running.

Screenshot. The screenshot application provides a view into your image, even if it runs headless. Clicking on the screenshot even allows you to open menus and to interact with the tools within your deployed image.

Version Uploader. The version uploader is a simple interface to Monticello. It allows you to check what code is loaded into the image and gives you the possibility to update the code on the fly. If you plan to use any of these tools in a deployed image make sure that they are properly secured from unauthorized access. You don't want that any of your users accidentally stumble upon one of them and you don't want to give hackers the possibility to compromise your system.

19.8 Request Handler

Another possibility to manage your headless images from the outside is to add a request handler that allows an administrator to access and manipulate the deployed application. The technique described here is not limited to administering images, but can also be used for public services and data access using a RESTful API. Many web applications today provide such a functionality to interact with other web and desktop application. To get started we subclass `WARequestHandler` and register it as a new entry point:

```
WARequestHandler subclass: #ManagementHandler
     instanceVariableNames: ''
     classVariableNames: ''
     package: 'ManagementHandler'
```

```
ManagementHandler class >> initialize
     WAAdmin register: self at: 'manager'
```

The key method to override is `#handleFiltered:`. If the URL `manager` is accessed, then the registered handler receives `aRequestContext` passed into this method and has the possibility to produce a response and pass it back to the web server.

The most generic way of handling this is to provide a piece of code that can be called to evaluate Pharo code within the image:

```
ManagementHandler >> handleFiltered: aRequestContext
    | source result |
    source := aRequestContext request
        at: 'code'
        ifAbsent: [ self error: 'Missing code' ].
    result := Compiler evaluate: source.
    aRequestContext respond: [ :response |
        response
            contentType: WAMimeType textPlain;
            nextPutAll: result asString ]
```

The first few lines of the code fetch the request parameter with the name code and store it into the temp `source`. Then we call the compiler to evaluate the code and store it into `result`. The last few lines generate a textual response and send it back to the web server.

Now you can go to a web server and send commands to your image by navigating to an URL like: http://localhost:8080/manager?code=SystemVersioncurrent. This will send the Smalltalk code `SystemVersion current` to the image, evaluate it and send you back the result.

Alternatively you might want to write some scripts that allow you to directly contact one or more images from the command line:

```
$ curl 'http://localhost:8080/manager?code=SystemVersion current'
Pharo1.0rc1 of 19 October 2009 update 10492
```

If you install a request handler like the one presented here in your application make sure to properly protect it from unauthorized access, see Section .

19.9 Summary

We show how you can deploy your Seaside applications as well as maintained them over time.

CHAPTER 20

JQuery

jQueryhttp://jquery.com is one of the most popular open-source JavaScript frameworks today. jQuery was created by John Resig and focuses on simplifying HTML document traversing, event handling, animating, and AJAX interactions for rapid web development.

There is a huge collection of pluginshttp://plugins.jquery.com/ available that extend the base framework with new functionality. One of the most popular of these plugins is jQuery UIhttp://jqueryui.com/. It provides additional abstractions over low-level interaction and animation, advanced effects and high-level themeable widgets for building highly interactive web applications.

jQuery and jQuery UI are both well integrated into Seaside 3.0. This allows you to access all aspects of the library from Smalltalk by writing Smalltalk code only. The Pharo side of the integration is automatically built from the excellent jQuery documentation - http://docs.jquery.comhttp://docs.jquery.com, so you can be sure that the integration is up-to-date and feature-complete.

20.1 Getting Ready

Make sure to have the packages Javascript-Core, JQuery-Core and JQuery-UI-Core loaded. For examples and functional tests also load the test packages Javascript-Tests-Core, JQuery-Tests-Core and JQuery-Tests-UI.

To use the libraries in your applications, you will need to load them in the Seaside web configuration application. You will notice that the core JQuery and JQueryUI libraries come in three forms which may be installed interchangeably. The `Development` versions have the full human-readable Javascript, and so are ideal for inspection and debugging during development; the `Deployment` versions are minified and gzipped to about 1/10th of the size of the development libraries, and so are much faster-loading for end users; and the `Google` versions link to copies of the libraries hosted by Google – as many sites reference these versions, your users may already have them cached, and so these can be the fastest loading versions.

`JQDevelopmentLibrary`	JQuery	Full
`JQDeploymentLibrary`	JQuery	Compressed

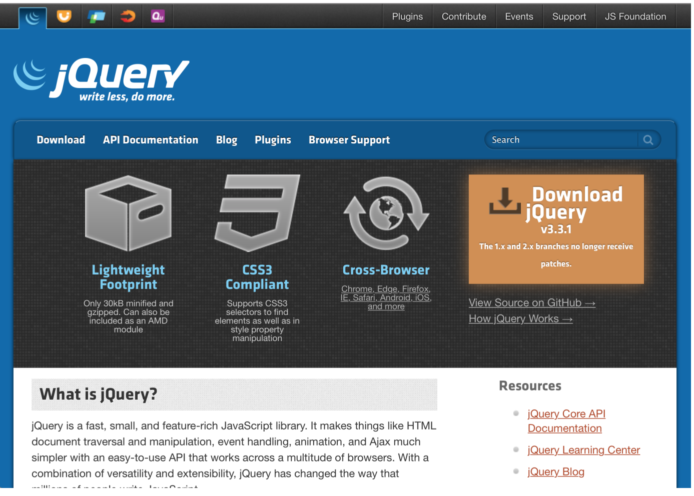


Figure 20-1 jQuery Demo and Functional Test Suite.

`JQGoogleLibrary`	JQuery	Google
`JQUiDevelopmentLibrary`	JQuery UI	Full
`JQUiDeploymentLibrary`	JQuery UI	Compressed
`JQUiGoogleLibrary`	JQuery UI	Google

For many of the most popular jQuery plugins there are ready-made Smalltalk wrappers in the Project JQueryWidgetBoxhttp://www.squeaksource.com/JQueryWidgetBox on SqueakSource available.

20.2 JQuery Basics

jQuery has a simple but powerful model for its interactions. It always follows the same pattern depicted in Figure 20-2. You basically get aJQuery and configure it, navigate its elements and activate it.

To instantiate a `JQueryClass`, you ask a factory object for a new instance by sending the message `jQuery`. In most cases the factory object is your `WAHtmlCanvas`, but it can also be a `JSScript`.

```
html jQuery
```

While the `JQueryClass` is conceptually a Javascript class, it is implemented as a Pharo instance. `html jQuery` returns an instance of `JQueryClass`.

Creating Queries

To create a `JQueryInstance`, we specify a CSS selector that queries for certain DOM elements on the your web-page. For example, to select all HTML div tags with the CSS class `special` one would write:

```
html jQuery expression: 'div.special'
```

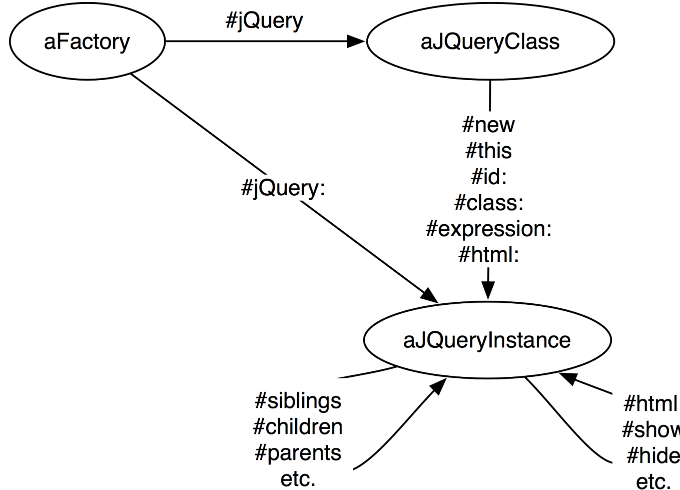


Figure 20-2 jQuery Lifecycle.

This expression returns a `JQueryInstance` object that represents all HTML tags matching the given CSS query `div.special`. There is also a slightly shorter form that does exactly the same:

```
html jQuery: 'div.special'
```

You find more details on creating queries in Section 20.3.

Refining Queries

If you browse the class `JQueryInstance`, you will see that you can add more elements or filter out elements before applying the jQuery action. For example, to select the siblings of the currently selected elements you would write:

```
(html jQuery: 'div.special') siblings
```

You find more details on refining queries in Section 20.4.

Performing Actions

Once you have identified the elements, you can specify the actions you wish to perform. These actions can delete, move, transform, animate or change the contents of the element. For example, to remove the elements we selected earlier we write:

```
(html jQuery: 'div.special') siblings; remove
```

There are over 180 actions provided by jQuery; these can be investigated by browsing the `JQueryInstance` Pharo class, and by visiting the jQuery documentation at http://api.jquery.com/.

You find more details on performing actions in Section 20.5.

20.3 Creating Queries

If you've already used jQuery (or followed the link to the documentation), you will already be familiar with the `$()` syntax for specifying CSS queries to select DOM elements. `JQueryClass>>expression:` exposes this same interface, but there are also a number of shortcut forms available to you. All the constructor methods return an instance of `JQueryInstance`.

$("div.hint")

Normally a jQuery instance is setup with a CSS selector. You can either use the long form (1) or take the shortcut (2). Of course, both forms are absolutely equivalent, in practice you will mostly encounter the shorter second form:

```
html jQuery expression: 'div.hint'.      "(1)"
html jQuery: 'div.hint'.                 "(2)"
```

$("#foo")

Often you want to create a query with an element ID. Again we have different possibilities to instantiate that query. (1) and (3) use a normal CSS selector for element IDs. (2) uses the `id:` selector, and (4) uses a shortcut using a symbol. Note that the forth form only works for symbols, if you pass a string it will be interpreted as a CSS selector.

```
html jQuery expression: '#foo'.          "(1)"
html jQuery id: 'foo'.                   "(2)"
html jQuery: '#foo'.                     "(3)"
html jQuery: #foo.                       "(4)"
```

$("*")

The CSS selector to match all elements in the page is `*`. Again you have several equivalent possibilities to achieve the same in jQuery. The first two use a CSS selector, while the last one uses a convenience method:

```
html jQuery expression: '*'.
html jQuery: '*'.
html jQuery all.
```

$(this)

If you want to refer to the currently active DOM element from an event handler you can use `new` or `this`.

```
html jQuery this.
html jQuery new.
```

Note that the `new` you call here is not the one implemented in the Smalltalk class `Behavior`, but a custom one implemented on the instance side of `JQueryClass`. Similar to all other constructor methods it returns an instance of `JQueryInstance`.

$("<div></div>")

Furthermore, jQuery provides the possibility to create new HTML code on the fly, that inserted into an existing element. Again we have different equivalent possibilities to do this. The first

one uses a raw HTML string, with Seaside we want to avoid this in most cases. The second and third variation uses a block with a new renderer that we can use with the normal Seaside rendering API.

```
html jQuery expression: '<div></div>'.
html jQuery html: [ :r | r div ].
html jQuery: [ :r | r div ].
```

$(function() { alert('Hello'); })

Last but not least there is the case of the `$()` syntax allows you to specify some action that should happen once the page is ready. This is done by attaching

```
html jQuery ready: (html javascript alert: 'Hello').
html jQuery: (html javascript alert: 'Hello').
```

20.4 Refining Queries

After you made an initial query you can refine the result with additional operations. All existing operations are described in this section:

Siblings

Get a set of elements containing all of the unique siblings of each of the matched set of elements.

```
aQuery siblings.
aQuery siblings: 'div'.
```

Next Siblings

Get a set of elements containing the unique next siblings of each of the given set of elements.

```
aQuery next.
aQuery next: 'div'.
```

Or, find all sibling elements after the current element.

```
aQuery nextAll.
aQuery nextAll: 'div'.
```

Or, find all following siblings of each element up to but not including the element matched by the selector.

```
aQuery nextUntil: 'div'.
```

Previous Siblings

Get a set of elements containing the unique previous siblings of each of the matched set of elements.

```
aQuery previous.
aQuery previous: 'div'.
```

Or, find all sibling elements in front of the current element.

```
aQuery previousAll.
aQuery previousAll: 'div'.
```

Or, find all previous siblings of each element up to but not including the element matched by the selector.

```
aQuery previousUntil: 'div'.
```

Children

Get a set of elements containing all of the unique immediate children of each of the matched set of elements.

```
aQuery children.
aQuery children: 'div'.
```

Find all the child nodes inside the matched elements (including text nodes), or the content document, if the element is an iframe.

```
aQuery contents.
```

Searches for all elements that match the specified expression.

```
aQuery find: 'div'.
```

Parents

Get a set of elements containing the unique parents of the matched set of elements.

```
aQuery parent.
aQuery parent: 'div'.
```

Or, find all following siblings of each element up to but not including the element matched by the selector.

```
aQuery parents.
aQuery parents: 'div'.
```

Or, find all the ancestors of each element in the current set of matched elements, up to but not including the element matched by the selector.

```
aQuery parentsUntil: 'div'.
```

Get a set of elements containing the closest parent element that matches the specified selector, the starting element included.

```
aQuery closest.
aQuery closest: 'div'.
```

20.5 Performing Actions

There is a wide variety of actions that come supported with jQuery. jQuery UI and thousands of other plugins add even more. In this section we present some of the most common actions provided by the core framework.

Classes

The following examples add, remove or toggle the CSS class `important` given as the first argument. These methods are commonly used to change the appearance of one or more HTML elements for example to visualize a state change in the application.

```
aQuery addClass: 'important'.
aQuery removeClass: 'important'.
aQuery toggleClass: 'important'.
```

Also you can query if a particular class is set:

```
aQuery hasClass: 'important'.
```

Styles

Similarly you can change the style of one or more HTML elements. By providing a dictionary you can change multiple CSS styles at once:

```
aQuery css: aDictionary.
```

Alternatively you can use a dictionary-like protocol to read and write specific style properties:

```
aQuery cssAt: 'color'.
aQuery cssAt: 'color' put: '#ff0'.
```

Note that in most cases it is preferred to use CSS classes instead of hardcoding your style settings into the application code.

Attributes

While the above methods change the `class` and `style` attribute of one or more DOM elements, there are also accessor methods to change arbitrary HTML attributes. By providing a dictionary of key-value pairs you can change multiple attributes at once:

```
aQuery attributes: aDictionary.
```

Alternatively you can use a dictionary-like protocol to read and write attributes:

```
aQuery attributeAt: 'href'.
aQuery attributeAt: 'href' put: 'http://www.seaside.st/'.
```

Replace Contents

A common operation on DOM elements is to change their contents, for example to update a view or to display additional information. To set the HTML contents of matched elements you can use the following construct that will replace the contents with `<div></div>`:

```
aQuery html: [ :r | r div ].
```

Alternatively you can set the text contents of each element in the set of matched elements:

```
aQuery text: 'some text'.
```

Last but not least you can set the value. This is especially useful for form fields, that require different ways to set the current contents (input fields require you to change the attribute value,

text areas require you to change the contents). The following code takes care of the details automatically:

```
aQuery value: 'some value'.
```

Insert Contents

Alternatively to replacing the contents you can append new contents. `before:` inserts content before each element in the set of matched elements; `prepend:` inserts content to the beginning of each element in the set of matched elements; `append:` inserts content to the end of each element in the set of matched elements; and `after:` inserts content after each element in the set of matched elements.

```
aQuery before: [ :r | r div ].
aQuery prepend: [ :r | r div ].
aQuery append: [ :r | r div ].
aQuery after: [ :r | r div ].
```

Note that, as with `html:`, the argument can be any renderable object: a string, a Seaside component, or a render block as in the given examples.

Animations

Showing or hiding DOM elements is one of the most common operations. While this is typically done by adding or removing a CSS class, jQuery provides a simpler way. The action `show` makes sure that the matching DOM elements are visible. If a duration is given as a first parameter, the elements are faded-in:

```
aQuery show.
aQuery show: 1 second.
```

The same functionality is available to hide one or more DOM elements with `hide`:

```
aQuery hide.
aQuery hide: 1 second.
```

20.6 Adding JQuery

After creating a jQuery object on the Pharo side, it is time to investigate on how to add them to the Seaside application.

The standard way of doing so in jQuery is to keep all the Javascript functionality *unobtrusive* in a separate Javascript file. This is possible with Seaside, but not the suggested way. In Seaside we try to encapsulate views and view-related functionality in components. Furthermore we keep components independent of each other and reusable in different contexts, what does not work well with sharing unobtrusive Javascript code. Additionally, the unobtrusiveness comes into the way when we want to define AJAX interactions.

Attaching to Element

The following scripts show how a query can be attached to an anchor and activated.

```
html anchor
    onClick: (html jQuery: 'div') remove;
    with: 'Remove DIVs'
```

```
html anchor
    onClick: (html jQuery this) remove;
    with: 'Remove Myself'
```

Execute at Load-Time

Javascript proposes a way to execute a query at load-time using `$(document).ready(...)`. With Seaside, you can forget about `$(document).ready(...)`, since Seaside has its own mechanism there.

```
html document addLoadScript: (html jQuery: 'div') remove
```

20.7 AJAX

AJAX is an acronym for *Asynchronous JavaScript and XML*. The fact that it is asynchronous means that additional data is passed to, or requested from the web server in the background, without the user waiting for it to arrive. JavaScript obviously names the programming language that is used to trigger the request. Fortunately the data being transmitted by an AJAX request doesn't have to be in XML. It can be anything that can be sent through the HTTP protocol. The reason for the "XML" in the name is that in most web browsers the internal implementation of this functionality can be found in an object called `XMLHttpRequest`. JQuery also supports the AJAX.

Loading

```
aQuery load html: [ :r | r div: Time now ].
```

No Query

Once loaded we can get an AJAX object using `ajax` query.

```
html jQuery ajax
```

Generators

An AJAX object can be configure to emit either html or script using the corresponding messages:

```
anAjax html: [ :r | r div ].
anAjax script: [ :s | s alert: 'Hello' ].
```

Triggering Callbacks

Finally a query can be stored and triggered later.

```
anAjax serialize: aQuery.
anAjax trigger: [ :p | ... ] passengers: aQuery.
anAjax callback: [ :v | ... ] value: anObject.
```

20.8 How tos

The following shows a list of possible actions

Click and Show

The following shows how to toggle a given div (here help) on click.

```
html anchor
     onClick: (html jQuery: 'div.help') toggle;
     with: 'About jQuery'.

html div
    class: 'help';
    style: 'display: none';
    with: 'jQuery is a fast and ...'
```

Open Light box

In the following we show how we can open a modal dialog box.

```
| id |
html div
    id: (id := html nextId);
    script: (html jQuery new dialog
         title: 'Lightbox Dialog';
         modal: true);
     with: [ self renderDialogOn: html ]
html anchor
     onClick: (html jQuery id: id) dialog open;
     with: 'Open Lightbox'
```

Replace a Component

In the following we show how to replace a component by a new one.

```
html div
     id: (id := html nextId);
     with: child.

html anchor
     onClick: ((html jQuery id: id) load
         html: [ :r |
             child := OtherComponent new;
             r render: child ]);
     with: 'Change Component'
```

Updating multiple components

The following snippet shows how we can update multiple element of the DOM. Here we have two components date and time. In the script we will render each component on click.

```
html div id: #date.
...
html div id: #time.

html anchor
    onClick: (html jQuery ajax script: [ :s |
        s << (s jQuery: #date)
             html: [ :r | r render: Date today ].
        s << (s jQuery: #time)
             html: [ :r | r render: Time now ] ]);
     with: 'Update'
```

20.9 Enhanced To Do Application

jQuery is an increasingly popular Javascript library. Let's port the the ToDo application to use jQuery for the Javascript functionality.

First, we'll implement the heading highlight effect with jQuery UI. Then we'll move on to implementing a couple of interesting effects and eye-candy possible with jQuery. Drag and drop is easy to implement, but we'll need to do something special to get the "in place" editing to work in jQuery.

If you have already worked through enhancing the ToDo application with Prototype and Scriptaculous, then the jQuery version will seem very familiar - we are still working with JavaScript underneath the covers after all.

Adding an Effect

We'll go ahead and factor the `renderContentOn:` method to add a method to handle rendering the heading and just make modifications to the new method.

```
ToDoListView >> renderContentOn: html
    self renderHeadingOn: html.    "<-- added."
    html form: [
        html unorderedList
            id: 'items';
            with: [ self renderItemsOn: html ].
        html submitButton
            text: 'Save'.
        html submitButton
            callback: [ self add ];
            text: 'Add' ].
    html render: editor
```

The `renderHeadingOn:` method leverages the jQuery UI library to add the highlight effect to the header of our component.

```
ToDoListView >> renderHeadingOn: html
    html heading
        onClick: html jQuery this effect highlight;
        with: self model title.
```

We create a query using `html jQuery this` that selects the heading DOM element. Next we send effect to get a `JQEffect` instance. Then finally we send `JQEffect>>highlight` which highligths the background color.

Altering the highlight color is left as an exercise for the reader.

Now for something a little more fun - let's add some help test that appears when you click on the heading; and it won't just "appear", it will slide open at a rate that we determine.

We do this by rendering a new `<div>` element that contains the help text, and changing the onClick of the header to apply our new cool effect to the new element. We also need some new CSS to help us out with this.

```
ToDoListView >> renderHeadingOn: html
    | helpId |
    helpId := html nextId.
    (html heading)
        class: 'helplink';
        onClick: ((html jQuery id: helpId)
            slideToggle: 1 seconds);
        with: self model title.
    (html div)
        id: helpId;
        class: 'help';
        style: 'display: none';
        with: 'The ToDo app enhanced with jQuery.'
```

```
ToDoListView >> style
  ^ '
  .help {
      padding: 1em;
      margin-bottom: 1em;
      border: 1px solid #008aff;
      background-color: #e6f4ff;
  }
  .helplink {
      cursor: help;
  }

  body {
      color: #222;
      font-size: 75%;
      font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  }
  h1 {
      color: #111;
      font-size: 2em;
      font-weight: normal;
      margin-bottom: 0.5em;
```

```
}
ul {
    list-style: none;
    padding-left: 0;
    margin-bottom: 1em;
}
li.overdue {
    color: #8a1f11;
}
li.done {
    color: #264409;
}'
```

20.10 Summary

This chapter showed how Seaside can interact with Javascript. It opens a large spectrum of possibilities.

CHAPTER 21

Wish list of new chapter

21.1 Willow and more

21.2 Bootstrap

21.3 MDL

- https://github.com/DuneSt/MaterialDesignLite
- https://mdl.ferlicot.fr

21.4 WebSockets