

Dynamic Web Development

with

seaside *



Stéphane Ducasse, Lukas Renggli,
David C. Shaffer, Rick Zaccone
with Michael Davies

Dynamic Web Development with Seaside

Stephane Ducasse

Lukas Renggli

C. David Shaffer

Rick Zaccone

with Michael Davies

16 July 2014

This book is available as a download from book.seaside.st.

Copyright © July 16, 2014 Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone.
This book is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0
license.

Published by Square Bracket Associates, Switzerland.

squarebracketassociates.org

ISBN 978-3-9523341-1-9

First Edition, August, 2010.

Cover art by Samuel Morello.

Contents

1	Introduction	3
1.1	What is Seaside?	3
1.2	Seaside Applications	5
1.3	What is Smalltalk?	6
1.3.1	One-Click Image	6
1.4	Structure of the Book	7
1.5	Formatting Conventions	8
1.6	About the Online Book	9
1.7	Acknowledgments	10
I	Getting Started	13
2	Pharo Smalltalk	17
2.1	Using the One Click Image	17
2.1.1	Of Mice and Menus	18
2.2	What is a Smalltalk Image?	19
2.3	The Comanche Server	20
2.4	A First Seaside Component	21
2.4.1	Defining a Category	21
2.4.2	Defining a Component	21
2.4.3	Defining the Code	23
2.4.4	Rendering a Counter	27
2.4.5	Registering as a Seaside Application	28
2.4.6	Automatically Registering a Component	29
2.4.7	Adding Behavior	31
2.4.8	Adding a Class Comment	33
2.5	Saving your Package to Monticello	33
2.6	Summary	35
3	Cincom Smalltalk	37
3.1	Loading Seaside into VisualWorks	37
3.2	Seaside Operations Menu	38

3.3	Seaside Settings	39
3.4	Persistence	40
3.5	Developing in VisualWorks	40
3.5.1	Basic Tools	40
3.5.2	Packages and Categories	41
3.5.3	Name Spaces	41
3.5.4	Additional Components	41
3.6	Developing a First Component	42
3.6.1	Create a Package	42
3.6.2	Create a Name Space	42
3.6.3	Define a Component	43
3.6.4	Editing Generated Methods	43
3.6.5	Rendering the Counter	44
3.6.6	Registering the Application	44
3.6.7	Adding Behavior	46
3.6.8	Rendering the Behavior	46
4	GemStone/S	49
4.1	Using the GLASS Virtual Appliance	49
4.2	A First Seaside Component	52
4.2.1	Defining a Component	52
4.2.2	Defining Some Methods	54
4.2.3	Rendering a counter	55
4.2.4	Registering the Application	55
4.2.5	Adding Behavior	55
4.3	Keeping Up With the Latest Features	56
5	GNU Smalltalk	59
5.1	Creating a GNU Smalltalk image with Seaside loaded	59
5.2	Operating the GNU Smalltalk virtual machine remotely	60
5.3	Developing in GNU Smalltalk	61
5.4	Developing your first component	62
6	VA Smalltalk	65
6.1	Loading Seaside into VA Smalltalk	65
6.2	Starting VA Smalltalk Seaside	66
6.2.1	Seaside Server Control Panel Menu Options	67
6.2.2	Adding a Server Adaptor	68
6.2.3	Starting a Server Adaptor	68
6.2.4	A Simple Seaside Example	69
6.3	Developing Your First Seaside Component	70
6.3.1	Defining a Component	70
6.3.2	Adding Some Methods	71
6.3.3	Rendering a Counter	71
6.3.4	Registering the Counter Component	72

6.3.5 Adding Behavior to the Counter	73
II Fundamentals	75
7 Rendering Components	79
7.1 Rendering Hello World	80
7.2 Fun with Seaside XHTML Canvas	82
7.3 More Fun with the Seaside Canvas	85
7.4 Rendering Objects	86
7.5 Brush Structure	90
7.6 Learning Canvas and Brush APIs	92
7.7 Rendering Lists and Tables	96
7.8 Style Sheets	99
7.9 Summary	101
8 CSS in a Nutshell	103
8.1 CSS Principles	103
8.2 CSS Selectors	105
8.2.1 Tag Selector	105
8.2.2 Class Selector	105
8.2.3 Pseudo Class Selector	106
8.2.4 Reference or ID Selector	107
8.3 Composed Selectors	108
8.4 Summary	108
9 Anchors and Callbacks	111
9.1 From Anchors to Callbacks	112
9.2 Callbacks	113
9.3 About Callbacks	114
9.4 Contact Information Model	115
9.5 Listing the Contacts	116
9.6 Adding a Contact	117
9.7 Removing a Contact	118
9.8 Creating a mailto: Anchor	120
9.9 Summary	121
10 Forms	123
10.1 Text Input Fields and Buttons	123
10.2 Convenience Methods	126
10.3 Drop-Down Menus and List Boxes	128
10.4 Radio Buttons	131
10.5 Check Boxes	133
10.6 Date Inputs	135
10.7 File Uploads	137

10.8 Summary	139
III Using Components	141
11 Calling Components	145
11.1 Displaying a Component Modally	145
11.2 Example of call/answer	146
11.3 Call/Answer Explained	147
11.4 Component Sequencing	148
11.5 Answer to the Caller	149
11.6 Don't call while rendering	151
11.7 A Look at Built-In Dialogs	151
11.8 Handling The Back Button	152
11.9 Show/Answer Explained	153
11.9.1 Transforming a Call to a Show	154
11.10 Summary	155
12 Embedding Components	157
12.1 Principle: Component Children	157
12.2 Example: Embedding an Editor	158
12.3 Components All The Way Down	162
12.4 Intercepting a Subcomponent's Answer	166
12.5 A Word about Reuse	166
12.6 Decorations	167
12.6.1 Visual Decorations	169
12.6.2 Behavioral Decorations	171
12.7 Component Coupling	172
12.8 Summary	175
13 Tasks	177
13.1 Sequencing Components	177
13.2 Hotel Reservation: Task vs. Component	179
13.3 Mini Inn: Embedding Components	180
13.4 Summary	182
14 Writing good Seaside Code	185
14.1 A Seaside Program Checker	185
14.2 Slime at Work	187
14.3 Summary	188
IV Seaside In Action	189
15 A Simple ToDo Application	193
15.1 Defining A Model	193

15.2 Defining the View	196
15.3 Rendering and Brushes	198
15.4 Adding Callbacks	200
15.5 Adding a Form	202
15.6 Calling Other Components	203
15.7 Answer	205
15.8 Embedding Child Components	206
15.9 Summary	209
16 A Web Sudoku Player	211
16.1 Sudoku Solver	212
16.2 Sudoku Component	212
16.3 Rendering the Sudoku Grid	214
16.4 Adding Input	218
16.5 Back Button	220
16.6 Summary	222
17 Serving Files	223
17.1 Images	223
17.2 Including CSS and Javascript	225
17.3 Working With File Libraries	227
17.3.1 Creating a File Library	228
17.3.2 Referencing FileLibrary files by URL	230
17.4 Example of FileLibrary in use	231
17.5 Which method should I use?	233
17.6 A Word about Character Encodings	234
17.6.1 Character sets	234
17.6.2 Encodings	236
17.6.3 In Seaside and Pharo	237
18 Managing Sessions	241
18.1 Accessing the Current Session	241
18.2 Accessing the Session from the Debugger	242
18.3 Customizing the Session for Login	243
18.4 Lifecycle of a Session	245
18.5 Catching the Session Expiry Notification	247
18.6 Recovering from Expired Sessions	248
18.7 Manually Expiring Sessions	251
18.8 Summary	252
V Web 2.0	253
19 Really Simple Syndication	257
19.1 Creating a News Feed	258

19.2 Render the Channel Definition	259
19.3 Rendering News Items	260
19.4 Subscribe to the Feed	261
19.5 Summary	262
20 Dynamic Content with Scriptaculous	263
20.1 Prototype and script.aculo.us	264
20.1.1 Installation	264
20.1.2 Adding the Library	265
20.2 Snippets and Brushes	266
20.2.1 Instantiate a Brush	266
20.2.2 Using a Brush	268
20.2.3 Configure a Brush	270
20.3 Adding an Effect	270
20.4 AJAX: Talking back to the Server	274
20.4.1 Defining a Callback	274
20.4.2 Serializing a Form	275
20.4.3 Updating XHTML	277
20.4.4 Behind the curtains	279
20.4.5 Wrap Up	281
20.5 Drag and Drop	282
20.6 JavaScript Controls	284
20.7 Debugging AJAX	287
20.8 Summary	291
21 jQuery	293
21.1 Getting Ready	293
21.2 jQuery Basics	294
21.2.1 Creating Queries	296
21.2.2 Refining Queries	297
21.2.3 Performing Actions	299
21.3 Adding jQuery	301
21.4 Ajax	302
21.5 How To	302
21.5.1 Click and Show	302
21.5.2 Replace a Component	303
21.5.3 Update Multiple Elements	303
21.5.4 Open a Lightbox	303
21.6 Enhanced ToDo Application	304
21.6.1 Adding an Effect	304
21.6.2 Callbacks Redux	306
21.6.3 Drag and Drop	307
21.6.4 Summary	307
22 Comet	309

22.1 Inside Comet	309
22.2 Getting Started	310
22.3 The Counter Explained	312
22.4 Summary	314
VI Advanced Topics	315
23 Deployment	319
23.1 Preparing for Deployment	319
23.2 Seaside-Hosting	323
23.3 Deployment with Apache	325
23.3.1 Preparing the Server	325
23.3.2 Installing Apache	326
23.3.3 Installing the Squeak VM	327
23.3.4 Running the VMs	328
23.3.5 Configuring Apache	331
23.3.6 Serving files with Apache	333
23.3.7 Load Balancing Multiple Images	335
23.3.8 Using AJP	337
23.4 Maintaining Deployed Images	338
23.4.1 Headful System	338
23.4.2 Virtual Network Computing	338
23.4.3 Deployment Tools	339
23.4.4 Request Handler	341
24 REST Services	343
24.1 REST in a Nutshell	343
24.2 Getting Started with REST	345
24.2.1 Defining a Handler	345
24.2.2 Defining a Service	346
24.3 Matching Requests to Responses	347
24.3.1 HTTP Method	348
24.3.2 Content Type	349
24.3.3 Request Path	350
24.3.4 Query Parameters	351
24.3.5 Conflict Resolution	352
24.4 Handler and Filter	353
24.5 Request and Response	354
24.6 Advices and Conclusion	356
25 Some Persistency Approaches	357
25.1 Image-Based Persistence	358
25.2 Object Serialization	364
25.3 Sandstone: an Active-Record Image-based Approach	366

25.3.1	The SandstoneDB API	367
25.3.2	About Concurrency	369
25.4	Magma: an Object-Oriented Database	371
25.4.1	How it works	372
25.4.2	Getting Started	373
25.4.3	Running Remotely	375
25.5	GLORP: an Object-Relational Mapper	376
26	Magritte: Meta-data at Work	377
26.1	Basic Principles	377
26.2	First Example	380
26.3	Descriptions	383
26.4	Exceptions	385
26.5	Adding Validation	385
26.6	Accessors and Mementos	386
26.7	Custom Views	388
26.8	Custom Descriptions	388
26.9	Summary	389

Foreword

If you are writing your first web application, you may look at Seaside and think, “What’s the fuss? This looks a lot like writing any other application.” But those of you who bear the scars of hours spent extracting POST variables, worrying about entity encoding, and passing data from page to page will understand the significance of this carefully crafted illusion.

Avi Bryant and I created Seaside because we wanted to spend our time writing great applications instead of worrying about what to name our form fields. If you actually like spending your time thinking about form field names – not to mention headers, cookies, URLs, redirects, session keys, and so on – then you should probably stop reading now. If, like we did, you believe there must be a better way, then keep reading. You can still deal with most of those details manually, of course; you just won’t need to.

A handful of experienced core developers and early adopters have collaborated on this book to show you how to develop for the web and let Seaside handle the distractions. And once you’ve finished your first Seaside application, I’m willing to bet it won’t be your last. Some people find the “Seaside way” of doing things a little foreign at first; others feel immediately at home. But, once they’ve mastered it, most developers just can’t stand the idea of going back.

Paul Graham observes in his essay “Beating the Averages” that web development is unique in giving you complete freedom to choose any development tools you want. Your users don’t care (or even know) that you wrote your application with Seaside. But you should (and so should your competitors) because it could be your competitive advantage. In an industry where late, over-budget, and under-performing projects are the unfortunate norm, delivering on your promises can set you apart. Using Seaside, you can deliver more in less time and when the requirements change you can adapt more quickly.

One of my first Seaside projects, with a small team at a university, was an application to manage Curriculum Vitae. A big part of the project was working out electronic data exchange with funding agencies and, since they

were big and slow, we tried to maintain their momentum by adjusting our system to accommodate them. Also, our users had suffered through at least two previous attempts to force similar (failed) initiatives on them so we wanted to build support by engaging them fully in the development process. All this meant that our requirements were continuously evolving.

From the outset we arranged regular meetings with users and partners. We demonstrated the system, collected feedback, even prototyped ideas on the spot. For more complex requests, we took notes and promised to get back to them next week. After these meetings, I think most people were certain they would never hear from us again but we took pride in proving them wrong. Seaside allowed us to quickly develop the most promising ideas and get back to them – often within 24 hours – with something to look at. That builds real support.

Seaside began as a simple development tool but has grown into a mature framework, now the most productive web development environment I know. It has helped a humbling and fast-growing community of talented users and developers rediscover the joy of programming for the web. I hope that in the pages of this book you too will discover some of that joy. Welcome to the seaside.

Julian Fitzell

Seaside co-creator

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 Smalltalk – 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 (collection of Smalltalk classes). 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,

- native (Smalltalk) debugging interface, and
- support for using AJAX and other Web 2.0 technologies.

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 Smalltalk code.

Seaside was originally created by Avi Bryant and Julian Fitzell. It is supported by an active community and enhanced by several Seaside experts. Currently, Julian Fitzell, Philippe Marshall, and Lukas Renggli (one of the authors of this book), are leading its development.

Seaside is often described as a *heretic web framework* because by design it goes 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.

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 Smalltalk, 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 Smalltalk.

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

- Session state is maintained on the server.
- XHTML 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 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 at seaside.st/about/users. 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 (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 (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 (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 (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 Smalltalk?

In his book “Beyond Java”, Bruce Tate asks whether Seaside can really be innovative if it was developed using Smalltalk, a language that emerged in the late 80s. It’s a relevant question, and there is an answer; there are several good reasons why it is so innovative. First, the design of Smalltalk 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 were all invented in Smalltalk. Third, and most important, in Smalltalk 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 Smalltalk 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 Smalltalk because of Seaside. To help you get up to speed, we suggest you read *Pharo by Example*. It is a free book available at www.pharobyexample.org. Chapters 3, 4 and 5 contain a minimal description of Smalltalk, 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.

1.3.1 One-Click Image

There are several implementations of Smalltalk. Some are commercial, such as Cincom Smalltalk, GemStone Smalltalk, VA Smalltalk, and Dolphin Smalltalk. Others are open source, such as Pharo, Squeak and GNU Smalltalk. Seaside is developed in Pharo, then ported to the other Smalltalks. The first chapter provides an equivalent of a “Getting Started” chapter to all major Smalltalk implementations.

In this book, we use the Seaside 3.0 “One Click Image” which you can find on the Seaside website at www.seaside.st. The “One Click Image” is a bundle of everything you need to run Seaside once you unzip it. This book is based on Pharo Smalltalk, a fork of Squeak that is used to build the One Click Image. 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.

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

1.4 Structure of the Book

Part I: Getting Started

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

Part II: 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.

Part III: Using 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.

Part IV: 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.

Part V: Web 2.0

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.

Part VI: 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 in Squeak. 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 Smalltalk, 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:

Note

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 About the Online Book

A free online version of this book is available at book.seaside.st. The online version is always up-to-date and permits readers to add notes at the bottom of every page. This immediately notifies other readers of problems and helps us to quickly resolve remaining issues. We will regularly go through the notes and address the issues raised in the main text.

The complete book is written using the Pier content management system that itself is written using Seaside. The PDF version of the book is automatically rebuilt every night from the contents of the website.

The online version of the book can be navigated using the following keyboard shortcuts. This allows you to quickly navigate the contents of the book.

Keys		Action
k	<i>left-arrow</i>	Previous Page
j	<i>right-arrow</i>	Next Page
p		Parent Page
i		Table of Contents

1.7 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: 0zkrpm, 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, benedict101, benoit.astruc, bgridley, bilesja, bjorn.eiderback, blank, bonzini, bouraqadi, brauer, briannolan45, bromagosa, bruefkasten, bschwab, bugmenot, cacciaretti, carlg, carlos.crosetti, cdrick65, cems, cesar.smx, chaetal, chicoary, chip, chris.pollard, chrismeyer206, christophe.rettien, chunsj, citizen428, cjbachinger, colson, craig, crystal.dry.eyes, cuye, 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, ismail-shuaibu, 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, klbogotz, 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.agr, nielvv, nikita.pristupchik, niko.saint, niko.schwarz, nizar.jouini, nrf, nwmuellen, offray, pat.maddox, paulpham, pdebruic, peterg, petton.nicolas, phil, philhunt, 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, she-shadri.mantha, simon, simon.denier, smalltalk, smalltalktelevision, snoob-abk, 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, zhangx-inchun2008. 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.

Part I

Getting Started

This part will show you how to get a simple Seaside application up and running in your favourite Smalltalk dialect in less than 15 minutes. There is no point in reading all of the following chapters, simply pick the one describing your favourite platform and skip the others. Seaside itself is the same everywhere, only the Smalltalk interface and development tools differ slightly from dialect to dialect.

If you're new to Smalltalk, the instructions for Pharo/Squeak will introduce you to the Seaside One Click Image, which is designed to get you up and running as quickly as possible.

Chapter 2

Pharo Smalltalk

In this chapter we will show you how to get started with Seaside using the Seaside “One Click Image”. Within 15 minutes, you will install and launch Seaside, interact with a Smalltalk IDE and write a simple Seaside counter application. You will follow the entire procedure of creating a Seaside application. This process will highlight some of the features of Seaside. If you are new to Smalltalk, we suggest you to read chapters 3, 4 and 5 of *Pharo by Example* which is a free and online book available from www.pharobyexample.org. This will highlight some of the features of Smalltalk.

As we mentioned previously, there are several implementations of Smalltalk. Some are commercial, such as VisualWorks, VA Smalltalk, Gemstone, and Dolphin. Others are open source, such as Pharo, Squeak and GNU Smalltalk; and finally, some are free but not open source, such as Smalltalk/X. Seaside is developed in Pharo, then ported to the other Smalltalks.

2.1 Using the One Click Image

In this book, we use Seaside 3.0.4, included in the “One Click Image” which you can find on the Seaside website at www.seaside.st/download. The “One Click Image” is a bundle of everything you need to run Seaside, including the Pharo implementation of Smalltalk. We suggest that you use this bundle to start, even if you prefer a different Smalltalk implementation. While we expect that all of our example code will run in all of the Seaside ports, we have only tested our code thoroughly in the Smalltalk that is included with this bundle.

Begin by downloading the “One Click Image” from the site. Unzip the file and you should be able to launch the Seaside Integrated Development Envi-

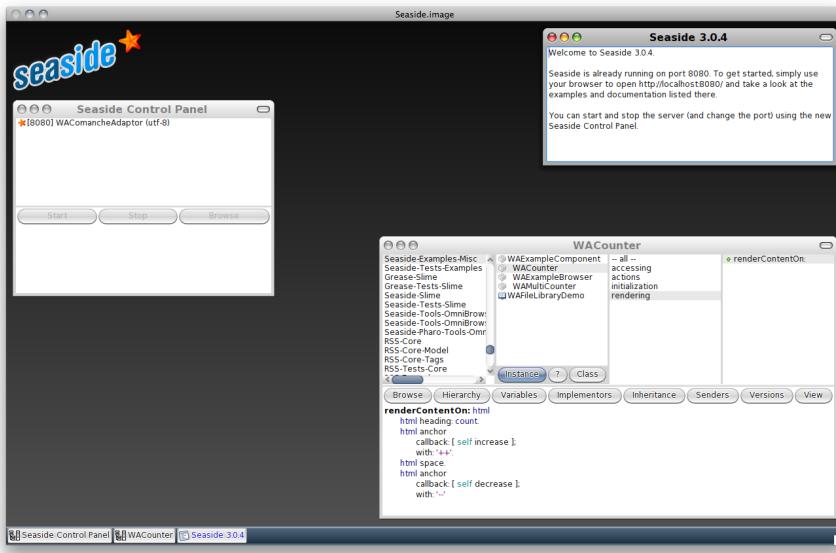


Figure 2.1: The Seaside development environment.

ronment (IDE) by double clicking on the icon appropriate for your platform. Windows users should double click `Seaside.lnk`, Linux users `Seaside.sh` and MacOS/OS-X users should simply double click on the application icon created when they unzipped the file. After this you should see the Seaside development environment open in a single window on your desktop similar to the one presented in Figure 2.1.

2.1.1 Of Mice and Menus

Because Smalltalk images are intended to work identically on many different operating systems, you may find some of the user interface may be slightly different from what you're used to. In order to help you understand the differences, we will outline the common stumbling points here.

Click. This is a standard mouse click, and is used to move focus to an item, to select an item in a list, and to select sections of text.

Right Click. We will use right-click to describe the action that will bring up the "context menu" on an item: this menu holds a list of actions relevant to the selected item. Mac users who are using a single button mouse will generally find that pressing the Control key while clicking the mouse button will have the same effect.

The Third Button. Smalltalk was first used with three-button mice, and some advanced features of Pharo may require you to emulate a three-button mouse. The ‘third-button’ may be bound to another button on your mouse, or the mouse scroll-wheel. Alternatively it may require you to press a key while clicking – the Alt key or the Command (Apple) keys on Macs are common choices. You shouldn’t worry about this until you need it, but it’s useful to know just in case you accidentally invoke one of these actions and wonder where it came from.

World Menu. To launch new applications and open new windows, you will use the World Menu. This can be opened by clicking anywhere on your Seaside desktop (i.e., left-clicking on the background area). We will use a shorthand to refer to this: *World | Workspace* means “click on the desktop to bring up the World Menu, then select the *Workspace* entry”.

Workspace. When you want to execute some code, you do so by opening a new workspace from the World Menu: use *World | Workspace*.

Try this new knowledge out now: Open a new workspace window. Type 1 + 1 into the window, and select it. Now right-click and select *Print it* from the context menu. You should see the answer 2.

2.2 What is a Smalltalk Image?

All Smalltalk objects live in something called an image. An image is a snapshot of memory containing all the objects at a given point in time. This means that your business objects, Seaside objects, all classes and their methods (since they are also objects), and development tools are all part of the image. The Seaside “One Click Image” includes a Smalltalk image with Seaside and a number of other tools pre-loaded to make it easier for you to get started using Seaside.

An image is loaded and executed by a virtual machine. When you ran Pharo in Section 2.1 you really were running the virtual machine on the pre-packaged “One Click Image” image. For the sake of brevity we call this “running the image.” The Smalltalk image includes active processes (Smalltalk processes are more akin to “threads” in other languages). When you load an image from a disk file you bring objects that were part of that image into RAM and resume execution of the active processes that were part of that image. If you were debugging when you saved the image, you’ll still be debugging when you load that image. We like to think of this as “picking up where you left off” (note that there are limits to this model: transient objects like a network connection that was active when the image was saved will not be available when the image is re-loaded). As an example, the Seaside “One Click Image” image was saved with the Comanche web server running so, when you load

this image that web server process will be running. We'll say more about that later.

Development in Pharo involves making incremental changes to the image (by creating classes, methods etc.). You will want to periodically save your Smalltalk image to disk so that your changes are saved. To save your image, select *World | Save* (i.e., raise the *world menu* by clicking somewhere in the background of the Pharo window, and click "Save" in that menu, as described previously). If you quit Pharo using *World | Quit*, you will also be prompted to save your image. You may resume your development efforts by running the image, as we did in Section 2.1.

2.3 The Comanche Server

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.

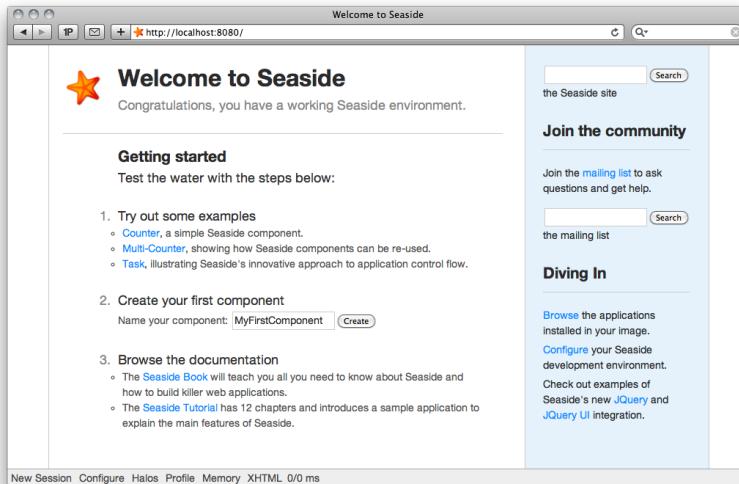


Figure 2.2: The Seaside server running.

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 following script asks the server to stop serving and start serving on port 8081:

```
WAKom stop.  
WAKom startOn: 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 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.

2.4 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. Finally we will see how we can save our work as a "Monticello" package.

2.4.1 Defining a Category

To start with, we define a new category that will contain the class that defines our component. If you don't have a class browser open yet, open one using *World | System Browser*. Raise the context menu for the category pane on the top left and select *Add category....* The menu can be found by right-clicking onto the list pane. You will get prompted to give a name as shown in Figure 2.3. We will use the name `WebCounter` for our category.

Figure 2.4 shows that the category has been created. Now we are ready to define a component.

In Pharo images you will often find the terms "Package" and "Category" used interchangeably. "Category" is a technical term based on the internal implementation, while "Package" better describes the intent of this pane. From now on, we will be using the term "Package".

2.4.2 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`

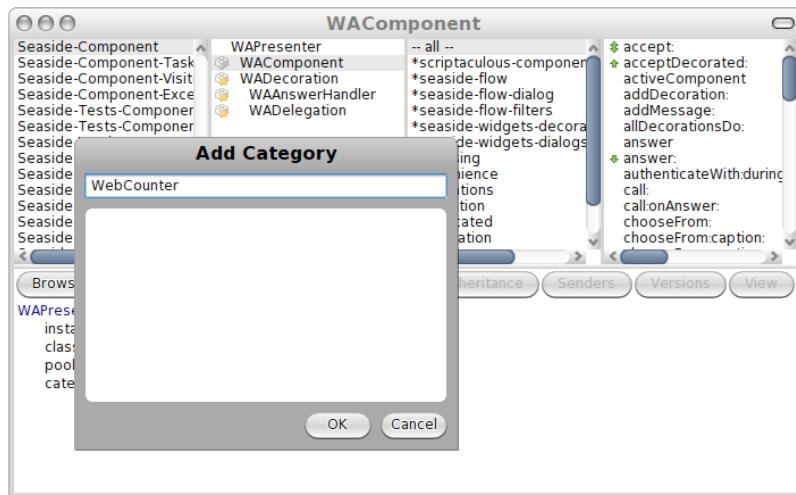


Figure 2.3: Create a category.

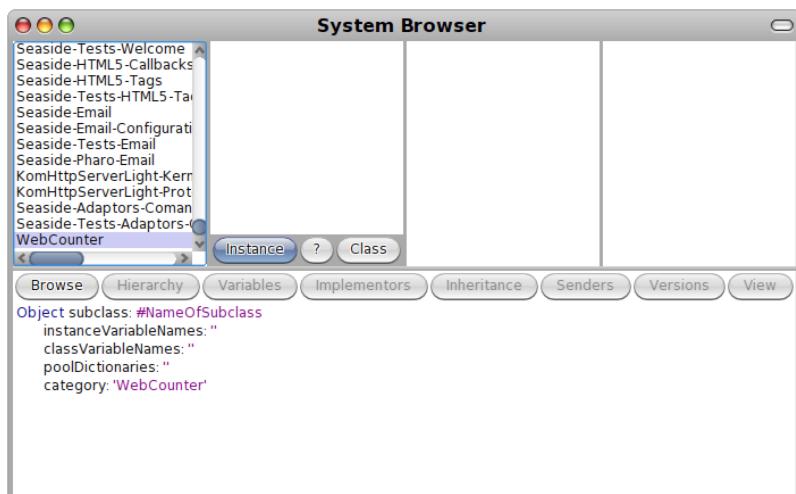


Figure 2.4: An empty category has been created.

(either directly or indirectly).

Note

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, to ensure that it is selected. The “class creation template” will appear in the source pane of the browser. Edit this template so that it looks as follows:

```
WAComponent subclass: #WebCounter
  instanceVariableNames: 'count'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'WebCounter'
```

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 “category” 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 2.5. Accept in Smalltalk jargon roughly means compile.

Once you have accepted, your browser should look similar to the one shown in Figure 2.6. 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.

2.4.3 Defining the Code

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.

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

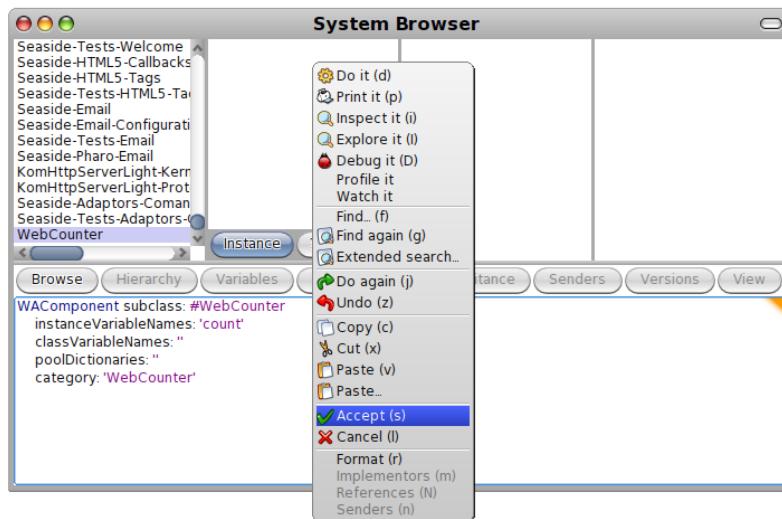


Figure 2.5: Creating the class WebCounter.

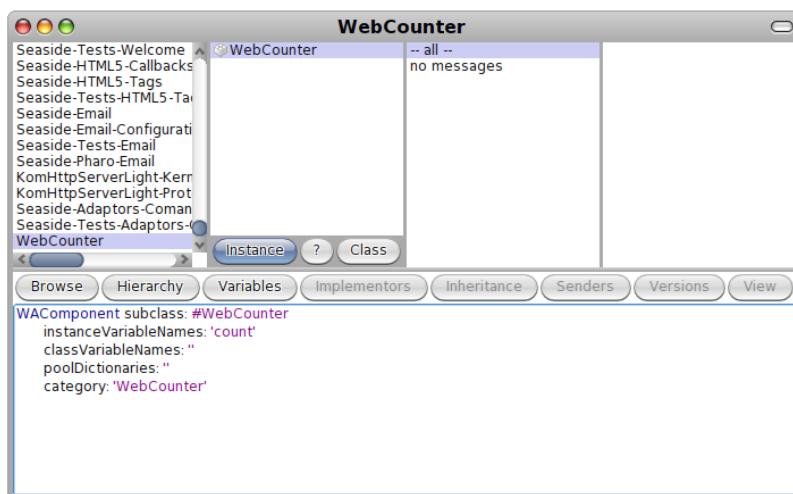


Figure 2.6: The class has been created.

follows normal Smalltalk 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.

First raise the context menu in the “method category” pane and select *Add category...* as shown in Figure 2.7. Select `initialization` from the resulting dialog, which will add `initialization` to the method category pane. Method categories have no effect on the functionality of your components; they are intended to help you organise your work.

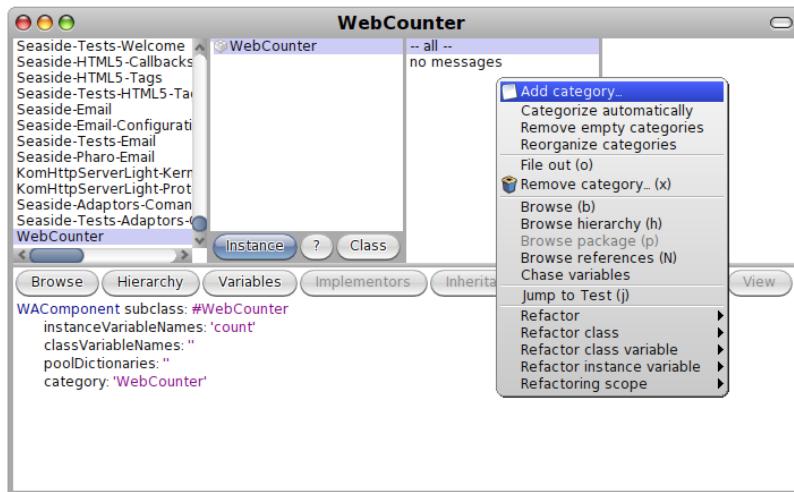


Figure 2.7: Adding a method category.

Now ensure that the `initialization` method category is selected, and then enter the following in the source pane – remember that you do not have to type `WebCounter>>` in the source code pane:

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

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 Figure 2.8.

Note

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.

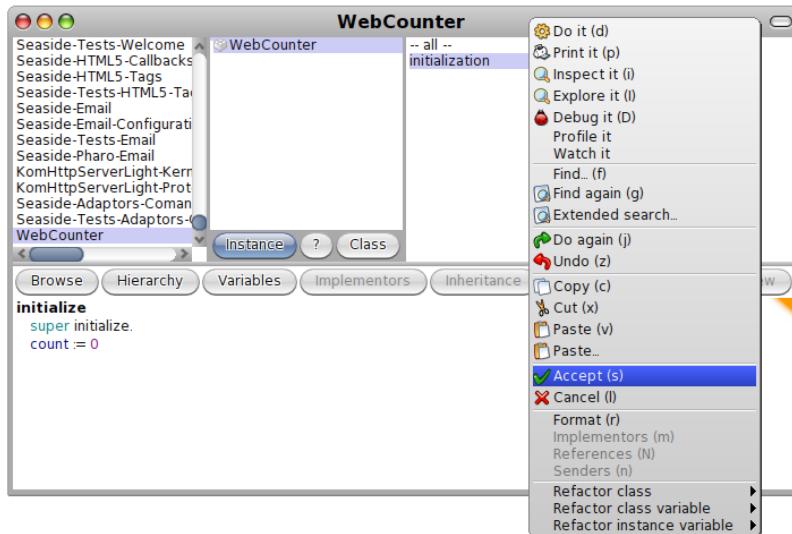


Figure 2.8: Compiling a method.

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 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, now create a new method category called `actions`. From the method category pane bring up the context menu and select *Add category...* and type the new category `actions`. In this new category define two more instance methods to change the counter state as follows.

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

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

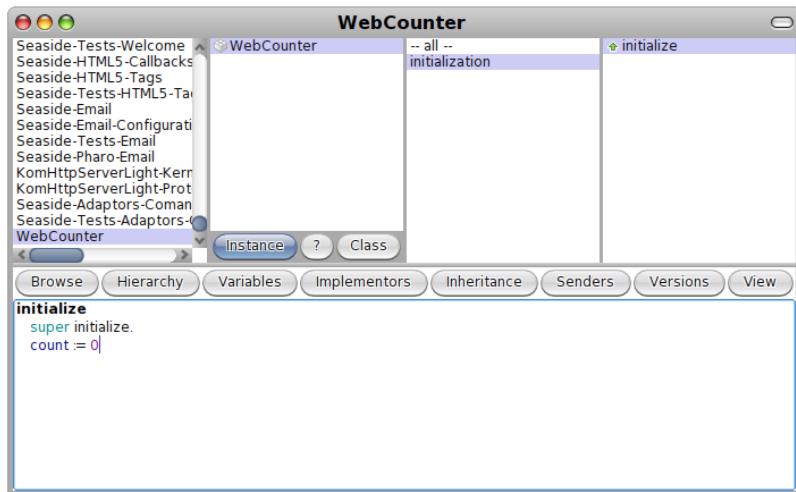


Figure 2.9: The method has been compiled.

Note

Many programmers like to keep their hands on the keyboard, avoiding the mouse whenever possible. Most of the actions we have described have keyboard shortcuts. Keyboard shortcuts for menu item actions are often indicated in the menu itself. For example the *Accept (s)* menu item can be activated by pressing the correct keyboard qualifier key together with the s-key. The keyboard qualifier key depends on what platform you're using, it may be *command*, *control* or *alt* depending on your platform.

2.4.4 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

```
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 2.10.

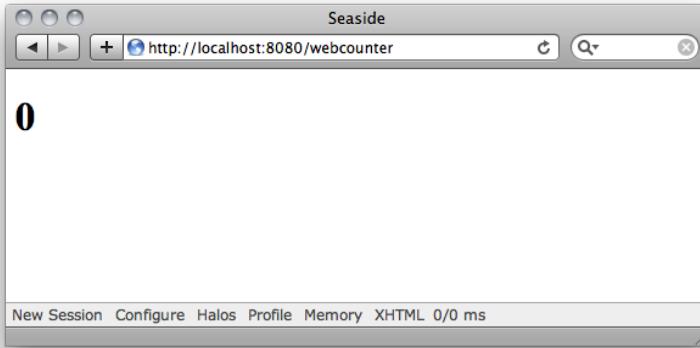


Figure 2.10: A simple counter.

2.4.5 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'` will register 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`.

Use *World | Workspace* to open a workspace, which is an area where you can run snippets of code. Type the text shown above, then select it with the mouse and bring up the context menu and select *Do it (d)*, alternatively use the keyboard shortcut.

Now you can launch the application in your web browser by going to `http://localhost:8080/webcounter/` and you will see your first Seaside component running.

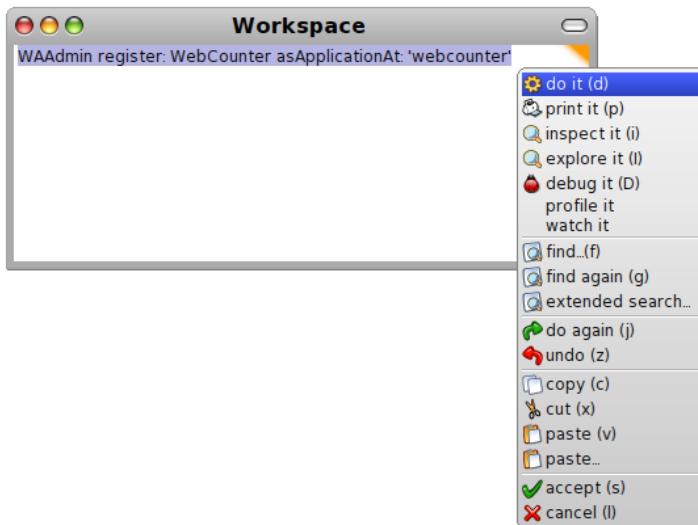


Figure 2.11: Register a component as an application from a workspace.

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.

2.4.6 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. Here is the `initialize` class method definition.

```
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.

Let's implement the method. Select the class `WebCounter`, click on the `Class` button under the class pane. You are now browsing the class methods of the class `WebCounter` (and there are none yet). Define a method category as

we did previously: click on the third pane and bring up the context menu. From this menu, select the menu item *add category*, and from the list select or type `class initialization`. Then in the bottom pane define and accept the method `initialize` as shown in Figure 2.12.

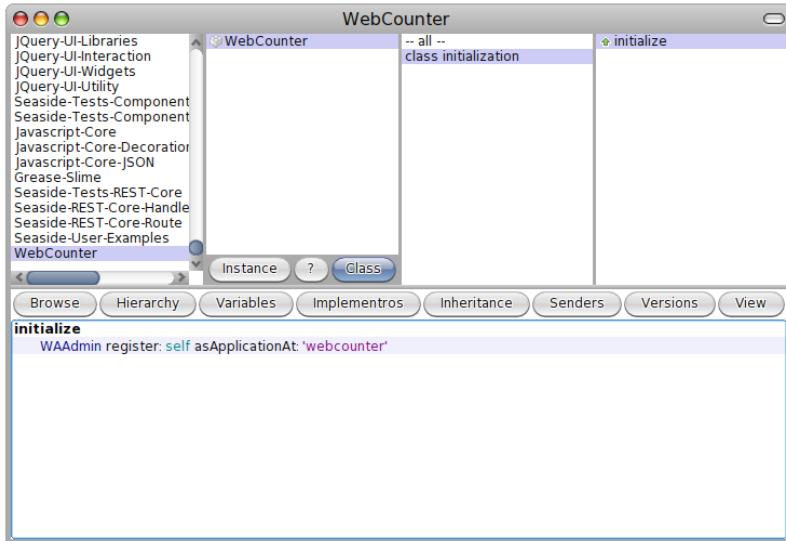


Figure 2.12: Compiling a class method.

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, type `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 Smalltalk 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 2.13 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.

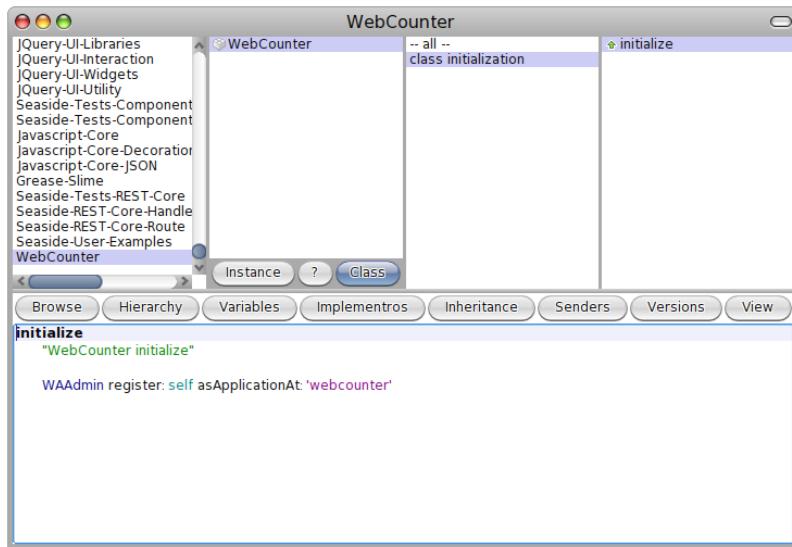


Figure 2.13: Adding the executable comment.

2.4.7 Adding Behavior

Now we can add some actions to our component. We will start with a very simple change; we will 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 2.14. Using *callbacks* allows us to define some code that will be executed when a link is clicked.

We modify the method `WAComponent>>renderContentOn:` as follows.

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

Note

Don't forget that `WAComponent>>renderContentOn:` is on the *instance* side.

Each callback is given a Smalltalk block: an anonymous method (strictly, a *lexical closure*) delimited by [and]. Here we send the message `callback:` (to

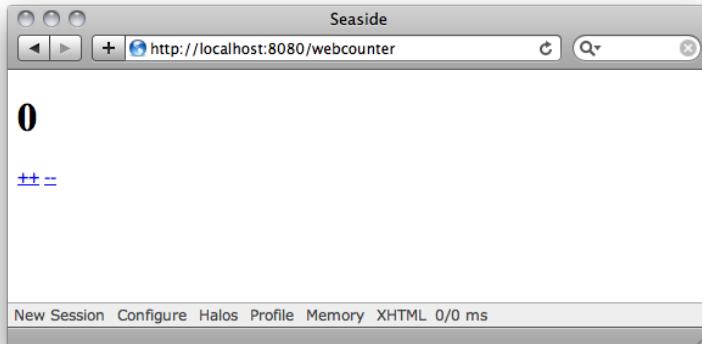


Figure 2.14: A simple counter with actions.

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 2.15.

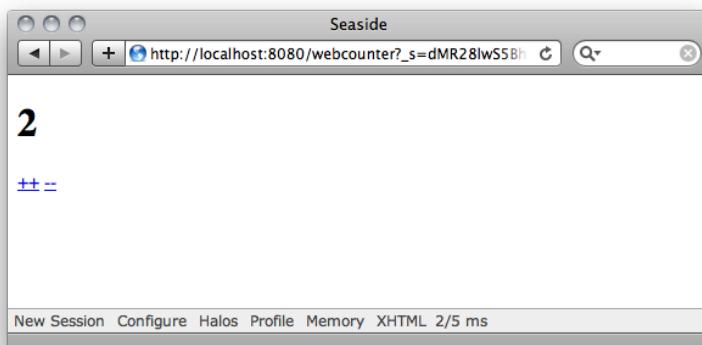


Figure 2.15: A simple counter with a different value.

2.4.8 Adding a Class Comment

A class comment (along with method comments) can help other developers understand a class. With your `WebCounter` class selected, press the class comment button ? in the class browser. The code pane in the browser will now show a class comment template. Delete this template and enter the comment shown in Figure 2.16. Use the context menu *Accept (s)* item to save your comment.

When you're studying a Smalltalk 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.

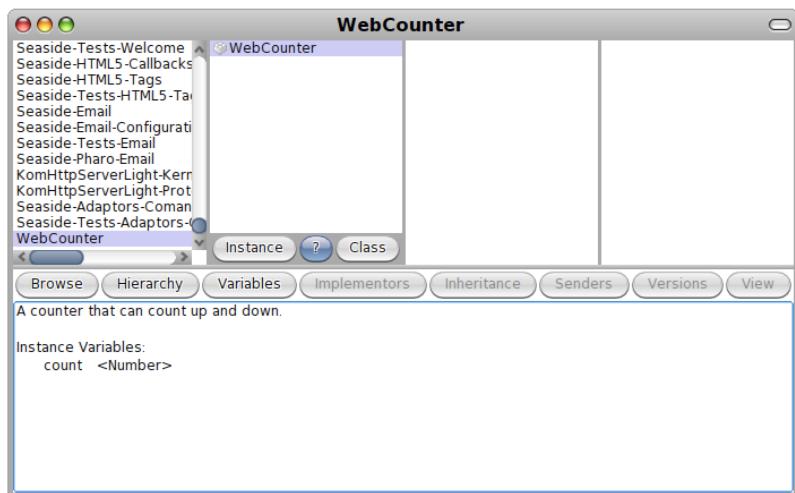


Figure 2.16: A class comment.

2.5 Saving your Package to Monticello

The Smalltalk image is a great place to work with live objects. It has a few drawbacks though, so it is useful to have some way to store your Smalltalk code in traditional files or on a server to share with others. Pharo uses the Monticello (<http://www.wiresong.ca/Monticello/>) source code control system for this purpose. Monticello stores code in *repositories*. These repositories can be network servers, databases, email, or just simple directories on a disk. We will create a directory repository so that Monticello can store your packages in files on your disk.

Note

The Pharo and Squeak communities use a free online repository called SqueakSource at <http://www.squeaksourc.com/> for sharing and collaborating on projects. Once you have registered as a member, you will be able to contribute to existing projects or start your own to save and share your code online. See the SqueakSource help pages for details.

Open Monticello. You can open Monticello by using *World | Monticello Browser*. You will see two main panes. The left hand pane shows packages installed in your image, and the right hand pane shows the repositories those packages came from.

Create your package. First create a package for your code, by pressing the *+Package* button. You should call your package *WebCounter* to ensure that it is automatically associated with the class category of the same name that you have already created.

Create a repository. Next, create a directory on your hard drive where you would like to store your Smalltalk source code. Now, in the Monticello browser in Pharo make sure no package is selected in the package pane. Click on the *+Repository* button and pick “Directory” from the resulting popup menu. You will be presented with a file/directory browser. Navigate to, and select, the directory you created. You should see this directory listed in the repository pane of the Monticello browser. Make sure this new repository is highlighted and select *add to package...* from the repository pane context menu. Select the *WebCounter* package from the resulting menu. You will need to navigate past a large number of package names to find the *WebCounter* package.

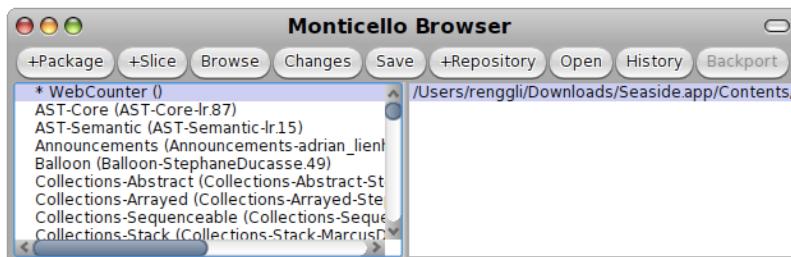


Figure 2.17: Monticello Browser.

Publishing your package. With all of this setup out of the way you can now save your package to the repository by selecting your package in the package pane, selecting your new repository in the repository pane and pressing *Save*. You will be asked to include a comment. Normally you indicate in a few words what you changed since the last time you saved the package. For now

we just enter “initial dump” and press *Accept*.

Normally you would publish a package any time you have made significant changes to it. That way if your image should become corrupted you can load the code from your last saved version.

Monticello can also be used to facilitate having multiple developers work simultaneously on a single package of code. For this you need a FTP or HTTP repository like SqueakSource. We will not discuss this advanced usage here, though.

Loading packages. Once your package has been published to a Monticello repository, it can be loaded into any Pharo image. To load your package into an image, first make sure that your repository is listed in this new image. If it isn’t, repeat the steps listed in *Create a repository* above. Now, in the Monticello browser, select your repository in the right pane and press *Open*. You will be presented with a repository browser such as the one shown in Figure 2.18. In the left pane, select the WebCounter package; in the right pane select the version; and then click *Load*.

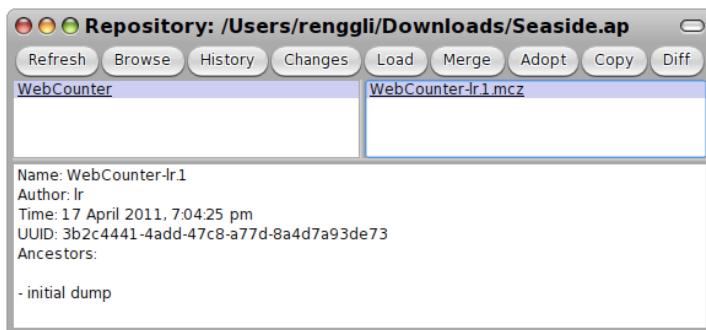


Figure 2.18: Monticello Repository Browser.

2.6 Summary

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

Chapter 3

Cincom Smalltalk

by Bruce Boyer, Cincom Systems, VisualWorks Development

In this section we describe how to get started developing a Seaside application in VisualWorks. We assume that you already have a 7.7 version of VisualWorks installed. If not, go to the Cincom Smalltalk download site <http://www.cincomsmalltalk.com/>. We'll be working with the noncommercial release, although the features that are preloaded into the noncommercial version aren't needed for working with Seaside.

As an alternative, Seaside and VisualWorks are tightly integrated in WebVelocity, which is also available on the Cincom Smalltalk download page. WebVelocity provides a browser-based development environment and detailed documentation that help guide you in developing Seaside applications.

3.1 Loading Seaside into VisualWorks

Seaside support is provided as a loadable parcel. There are actually several parcels, but most are prerequisites that are loaded automatically when you load the Seaside parcel.

Launch a VisualWorks image, such as `visualnc.im`. Then, open the Parcel Manager, *System | Parcel Manager* in the launcher. Select the Web Development category, then select the Seaside-All parcel, and click the load button, see Figure 3.1.

Once the parcel loads, a Seaside menu is added to the VisualWorks Launcher. Start a server and open a browser on the server using commands in this menu, see Figure 3.2.

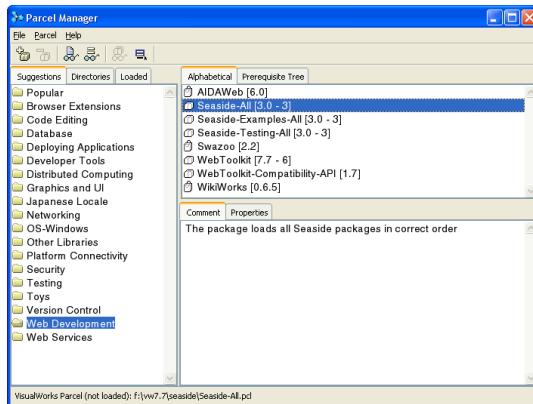


Figure 3.1: Loading Seaside.

By default, Seaside serves on `localhost:7777` and the entry address is `http://localhost:7777/seaside`. This is the default entry point, but can be changed in the Seaside Settings.

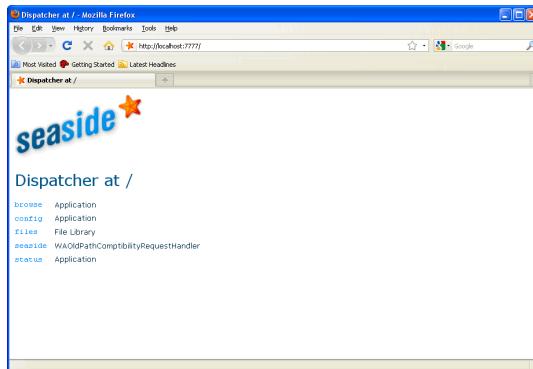


Figure 3.2: Seaside dispatcher.

3.2 Seaside Operations Menu

When the Seaside support parcel is loaded into VisualWorks, a Seaside menu is added to the VisualWorks Launcher. This provides menu access to several common control operations.

<i>Start Server</i>	Starts the Seaside server; requests can now be serviced. By default the server is running on port 7777.
<i>Stop Server</i>	Stops the Seaside server; no more transactions are accepted.
<i>Open Browser on Server</i>	Opens a web browser on the first page of the running Seaside server.
<i>Inspect Server</i>	Open a VisualWorks inspector on the server.
<i>Log to Transcript</i>	Log all server events (requests received, responses sent, etc.) into the Transcript.
<i>Debug Mode</i>	Server errors will open a debugger instead of being suppressed. This may prevent the server from being able to handle further requests, but allows you to investigate the errors on the server side.
<i>Settings</i>	Opens the Seaside Settings dialog.

3.3 Seaside Settings

Several properties of the Seaside environment can be set using the Settings Tool, accessible by picking *Seaside | Settings* in the Transcript. For each setting item there is online help provided. Click Help on the page with the setting item to view the description.

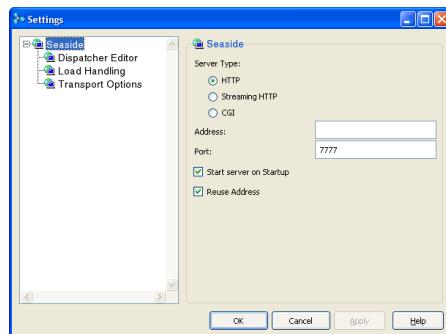


Figure 3.3: Seaside Settings.

For example, you can change the port the server is running on by entering the new number in the Port: field and clicking *Apply*. This stops the server and restarts it on the specified port.

3.4 Persistence

One of the strengths of VisualWorks as platform for Seaside is its strong support for relational databases commonly already installed in many environments. This provides an obvious persistence mechanism, when data being used by the Seaside application must be stored.

To simplify using these databases for persistence, Seaside on VisualWorks is integrated with GLORP, a framework for mappings between objects and the RDBMS data structures. GLORP is loaded when you load Seaside-All parcel. It's actually pre-loaded in the VisualWorks non-commercial image.

GLORP is a project of Camp Smalltalk. Information about the project, including current documentation, is available at <http://glorp.org>. Additional information about using GLORP in a Seaside/VisualWorks environment is available in the WebVelocity documentation, which is available by downloading that product from the Cincom Smalltalk Download site <http://www.cincomsmalltalk.com/>.

3.5 Developing in VisualWorks

Developers who are already familiar with VisualWorks can skip this section. For those who might be trying VisualWorks for the first time as a development environment for Seaside, a few brief comments about the development environment might be helpful. We'll make these comments in the context of developing the simple counter example.

3.5.1 Basic Tools

When a clean image is opened only two tools are open: the Launcher and a Workspace. The Launcher is the primary tool for opening additional tools, and also has a text area, called the transcript. The Workspace is a sandbox for testing code.

Most coding is done in the System Browser, which you open by selecting *Browse | System* in the Launcher. The System Browser provides access to all classes in the system, either by their containing package (or bundle or packages), or by class hierarchy. The package view is the more common working view.

Additional tools are opened as requested from the various menus in the Launcher and other tools. Online descriptions are available for most of these from the Help menu.

3.5.2 Packages and Categories

Smalltalk systems have traditionally provided class categories to help organize classes into related clusters. They had no semantic value, and were not represented by objects. In VisualWorks, categories have been supplanted by packages.

Unlike traditional categories, VisualWorks packages are real objects (instances, surprisingly enough, of the class `PackageModel`). They provide the organization feature of categories, but they are also the basic archival unit for the Store repository.

When developing application code, you should create your own new package to contain this work, rather than use an existing package. To create a new package, make sure no package is selected (Ctrl-click to deselect, or just select Local Image) in the System Browser, then pick *Package | New...* and enter a name in the dialog.

3.5.3 Name Spaces

Because of the strong potential for class and (global) variable name collisions in large VisualWorks applications, VisualWorks has implemented name spaces, a mechanism for restricting the referential scope of such names.

While advanced usage of name spaces can be quite involved and intricate, in practice can be quite simple, especially in the context of a Seaside application. Essential points are that:

- The top-level name space is named Smalltalk.
- All VisualWorks base Smalltalk classes and add-ons are defined in sub-namespaces of Smalltalk.
- Most Seaside classes are in the name space named Seaside.

Your application should, in general, be in a name space that you create for your own usage. However, for simplicity especially during early development, you can define your classes directly in the Smalltalk name space. The various examples included with Seaside are defined in their own name spaces.

3.5.4 Additional Components

A variety of Seaside expansions, enhancements, and examples are included with Seaside for VisualWorks, also provided as packages for easy loading. In

the parcel manager, browse the contents of the Seaside Web Development page. Each component has a comment describing its content.

3.6 Developing a First Component

To illustrate the above points, let's go through the initial steps of developing the Counter example in VisualWorks. We'll use a slightly different name so we won't conflict with the example already loaded with Seaside.

Open a System Browser from the Launcher. There is, of course, a tool button for this and for other tools as well.

3.6.1 Create a Package

First, create a package for your project. Make sure no package is selected in the top-left pane (ctrl-click on any currently selected package, or select *Local Image*), then pick *Package | New Package...*. As a name for our package enter `WebCounter`, and click OK. Leave the new package selected after it is created.

3.6.2 Create a Name Space

Next, we should create a name space for our work. Pick *Class | New | Namespace...* in the System Browser. In the dialog that opens, the package and Namespace fields are already correctly filled. Enter a name, which we will call `WebCounter` again. In the imports field, add `Seaside.*` so the field contents is: `private Smalltalk.* Seaside.*`



Figure 3.4: Creating a name space.

Then click OK. The `WebCounter` name space is created and selected in the System Browser, and the definition is shown as:

```
Smalltalk defineNameSpace: #WebCounter
    private: false
    imports: 'private Smalltalk.* Seaside.*'
    category: ''
```

3.6.3 Define a Component

A Seaside component is defined by a subclass of `WAComponent` class. Accordingly, to create a component, you create the corresponding class. To create the new class, with your package selected in the top-left pane of the System Browser, pick *Class | New Class....*. Again, the package and name space are correctly filled in already, as shown in Figure 3.5. For a class name, again enter `WebCounter`. Change the superclass to `WAComponent` in Seaside. You can search for it in several ways in the tool, and when selected it will show as `Seaside.WAComponent`. As is the case with the original `WACounter` example, we need the one instance variable `count`, so enter that.

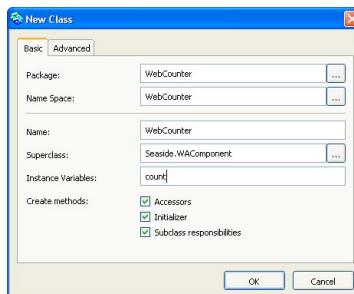


Figure 3.5: Create a class.

Leave the three checkboxes selected, and click *OK* to create the class. Because the check boxes were all checked, an initialize method for the class and two accessor methods for counter are also created. So, we have a start on the component.

3.6.4 Editing Generated Methods

Select the generated initialize method, and change it to:

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

To save the method, select *Edit | Accept* or press *Ctrl+S*.

3.6.5 Rendering the Counter

Each component is responsible for rendering itself in the web browser. This is done in Seaside by implementing a `WAComponent>>renderContentOn:` method. In this simple case, that means displaying the counter value.

Create a new protocol, called `rendering`, and add this method:

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

3.6.6 Registering the Application

To be able to access the application at a URL path, we must register it with Seaside.

First, we must declare that our application component can be a root component, or a starting point. To do this, click the Class tab in the System Browser, add a protocol called `testing`, and define this class method:

```
WebCounter class>>canBeRoot
    ^ true
```

Note that the method `canBeRoot` does not register the application (we will do that in a minute). It only declares that the component should appear in the list of registered applications.

To complete the registration, use the configuration tool in the web browser. If you have closed the browser, reopen it by selecting *Seaside | Open Browser* on Server in the VisualWorks launcher.

Click the config link in the browser (second item) to open the Configuration tool

Click the Add button to open the editor. Enter a name, `NewCounter`, for the handler, and select Application as the type, see Figure 3.6.

Click OK to accept this addition.

In the new page, scroll down to the General section. From the Root Class drop-down list, select our application, `WebCounter.WebCounter`, see Figure 3.7.

There are a lot of configuration options available on this page, but we don't need to bother with them for this example. Scroll down the page a little more and click Save.

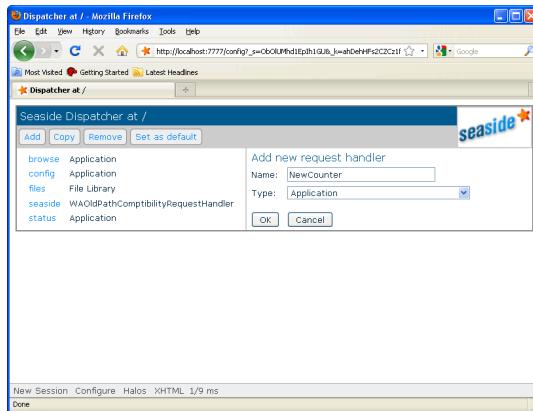


Figure 3.6: Add our application.

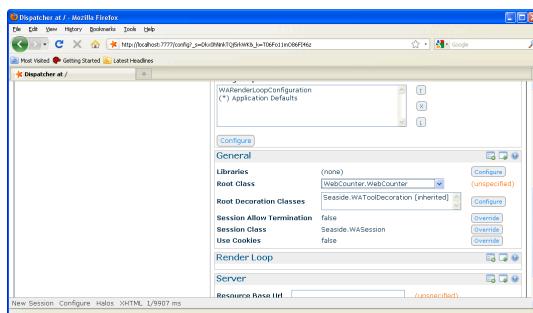


Figure 3.7: Configuring the application.

Our application is now listed as 'NewCounter', and will be listed on the opening web page. (Open a new browser on the server from the VisualWorks Launcher to verify that.)

This procedure is equivalent to what we do using the expression `WebCounter registerAsApplication: 'newcounter'` or redefining the class `initialize` as follows.

```
WebCounter class>>initialize
    self registerAsApplication: 'NewCounter'
```

To see that it works, go back to <http://localhost:7777> and click NewCounter to see the current state of our application.

Pretty boring at the moment.

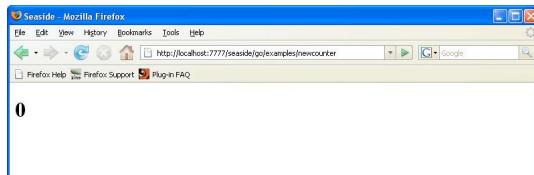


Figure 3.8: Registered Component.

3.6.7 Adding Behavior

We want to be able to increment and decrement the counter, which requires two methods.

In the System Browser, select *Protocol | New* to add a method category for methods to update the value of count. Call the category updating. Then add these two methods:

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

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

These use the `count:` accessor method that was generated when the class was first created.

3.6.8 Rendering the Behavior

Interactions between the browser and Seaside are done using callbacks in the Smalltalk code, and these are entered, directly or indirectly, in the `WAComponent>>renderContentOn:` method.

In the System Browser select the method and edit it to:

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

This adds two anchors with increment (++) and decrement (--) links that call back to our increase and decrease methods. Save the method. Then go back to the web browser and refresh the page to see the change.

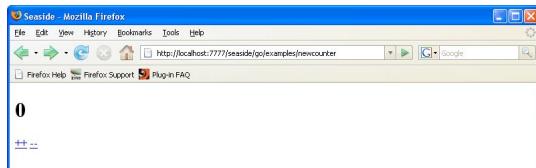


Figure 3.9: A counter with action.

Click the new links a few time to verify that it works.

That's it. You developed your first Seaside component in VisualWorks.

Chapter 4

GemStone/S

by Monty Williams, GemStone Systems

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 VVware 64-bit guest operating system.

4.1 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 4.1.



Figure 4.1: 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 Figure 4.2. If not, use the "GLASS Appliance" menu shown in Figure 4.4 to start, stop, or restart GLASS or its individual components.



Figure 4.2: GLASS Virtual Appliance status.

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 Figure 4.3. You can also view that web page from another computer on your network by using the "eth0:" IP address listed under "Network Information".



Figure 4.3: GLASS Virtual Appliance Seaside page.

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/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 4.4: GLASS Virtual Appliance menu.

4.2 A First Seaside Component

Let's build the counter application in GemStone/S. We'll be using a Squeak GUI as a development environment for GemStone/S. While this is quite similar to developing in Squeak, you will notice a few differences.

The most important difference developers need to be aware of is due to GemStone/S being a multi-user object database with ACID transaction properties. Each GemStone/S VM sees a consistent state of the database in isolation from intermediate changes underway in any other VM. Those changes cannot be seen until they are committed to the database. This is handled automatically in your Seaside application since the GemStone/S Seaside framework immediately commits any data sent from a web client to the GemStone/S application server. However, *when writing code, you must manually commit before that code can be used or even seen by other VM's*. Conversely, if you decide you've made a mistake, you can abort and your code changes will be erased.

Start by selecting "Run Squeak" in the "GLASS Appliance" menu in the toolbar. This will open a Squeak image containing a GemStone/S Login window (Figure 4.5). Click on the "Login" button, type your initials into the box that pops up, than click "accept". This will open a GemStone/S Transcript window (Figure 4.6). The text pane in the GemStone/S Transcript window is actually a workspace and not a transcript. This will be changed in the next release when this window is reimplemented in OmniBrowser.

4.2.1 Defining a Component

There are two ways to define the new `WebCounter` subclass of `WAComponent`.

Type the following class definition into the GemStone/S Transcript window or a GemStone workspace, then type CTL-d or use the "doit" menu item. Note



Figure 4.5: GemStone/S Login window.

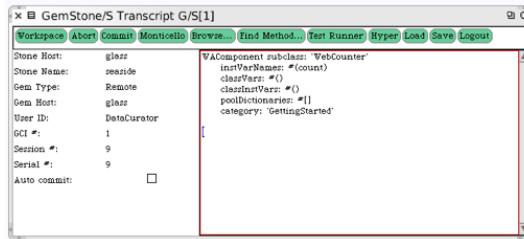


Figure 4.6: GemStone/S Transcript window.

that this GemStone/S class definition is slightly different than it would be in Squeak. The Squeak form works equally well, however you'll always see class definitions in the GemStone/S form when you browse the class.

```
WAComponent subclass: 'WebCounter'
instVarNames: #(count)
classVars: #()
classInstVars: #()
poolDictionaries: #[]
category: 'GettingStarted'
```

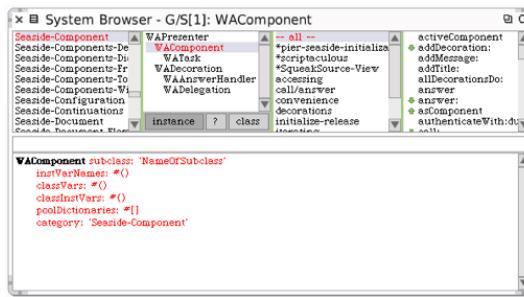


Figure 4.7: WACOMPONENT subclass template.

Alternatively, open a GemStone/S System Browser on the class `WAComponent` and fill in a "subclass template" as shown in Figure 4.7. To do this, click

the green "Browse" button in the GemStone/S Transcript window, and type WAComponent into the popup. Left click WAComponent and then select the "subclass template" menu item. After filling in the template with the class definition above, type CTL-s or select the "accept" menu item.

4.2.2 Defining Some Methods

Now we define some instance methods to initialize the value of the counter to zero , and to increment or decrement the value of the counter.

While it is not strictly necessary, it can be useful to define categories for methods you add. Middle button click on the all as shown in Figure 4.8, select "create category" and type in a category name. If you select a method category, new methods will appear in that category.

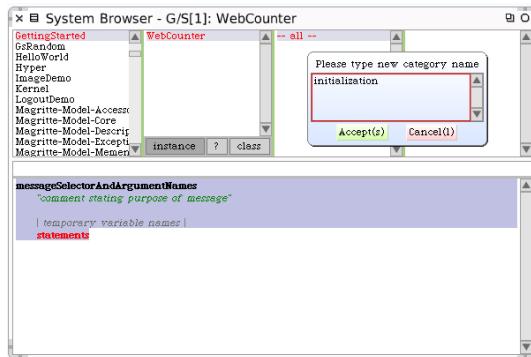


Figure 4.8: Create a category.

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

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

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

At this point it would be a good idea to commit your changes to the database. Click the commit button in the transcript window. You should always commit any changes to your code before logging out of GemStone, just as you would save a text document in an editor before logging off of your computer. If you want to throw away any code created or modified since your last commit, click the abort button instead.

4.2.3 Rendering a counter

In order to render the counter in a web browser, we must create a `WACOMPONENT>>renderContentOn:` method. The following method will display the value of the variable `count` as an HTML heading.

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

4.2.4 Registering the Application

Before you can access the counter application from your web browser, you must register it with Seaside. Type the following code into a Transcript or Workspace and then type CTRL-d or use the "doit" menu item. Afterwards, make sure to commit your code.

```
WebCounter registerAsApplication: 'WebCounter'.
```

Note that this expression can also be added in the class `initialize` method which is invoked at the time the class is loaded in memory as shown below.

```
WebCounter class>>initialize
    self registerAsApplication: 'WebCounter'
```

Now you can launch the application in your web browser by visiting `http://localhost/seaside/WebCounter`, see Figure 4.9.

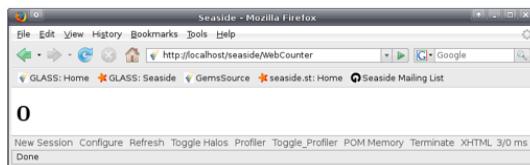


Figure 4.9: A simple counter.

4.2.5 Adding Behavior

Now we can add some actions by defining callbacks attached to anchors, see Figure 4.10. A callback is a piece of code that will be executed when a link is clicked. We will explain this in detail later in the book. Modify the method `WACOMPONENT>>renderContentOn:` as follows and commit your changes.

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

Now when you click on the links you will see the counter increment or decrement.



Figure 4.10: A simple counter with increment and decrement actions.

4.3 Keeping Up With the Latest Features

We regularly update the Seaside features of GemStone/S between major product releases. These are usually announced in Dale's blog at <http://gemstonesoup.wordpress.com/>. It's important to load Monticello packages in the sequence below to ensure the GLASS package in GemStone/S and the GemStone package in Squeak are in sync.

Step 1. Load the latest GLASS-dkh into GemStone from <http://seaside.gemstone.com/ss/GLASS>. This loads a large number of packages so it takes minutes to complete even on a fast connection. Be patient.

To load a Monticello package into GemStone/S, click the green "Monticello" button in the Transcript or Workspace window. Highlight the package name in the left pane (Figure 4.11), the URL starting with <http://seaside.gemstone.com/ss/> in the right pane, and then click the "open" button. This will open a Monticello repository browser. Highlight the package name in the left pane (Figure 4.12) and the package version in the right pane. Check the comments on the package for any precautions on compatibility before loading – you could need to use an earlier package.



Figure 4.11: GemStone Monticello browser.



Figure 4.12: GemStone Monticello repository.

After updating, it's a good idea to click the green "Test Runner" button in the Transcript window, and then click "Run Selected". If all the tests pass (Figure 4.14), commit your transaction. If some fail, it's probably simpler to abort rather than debug the problem.

Then log completely out of GemStone, or the next step will not succeed.

Step 2. Load the latest GemStone-dkh into Squeak from <http://seaside.gemstone.com/ss/GemStone>.

Then you can log back in to GemStone.

Loading a Monticello package into Squeak is similar (Figure 4.13), but you start by opening a Squeak Monticello browser rather than a GemStone/S one. It's important to remember Squeak and GemStone/S are two different systems. Even though they can run much of the same Smalltalk code, it's important to load your code into the one you intended.

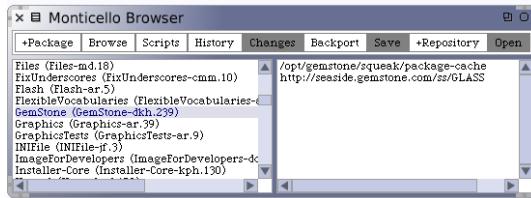


Figure 4.13: Squeak Monticello repository.

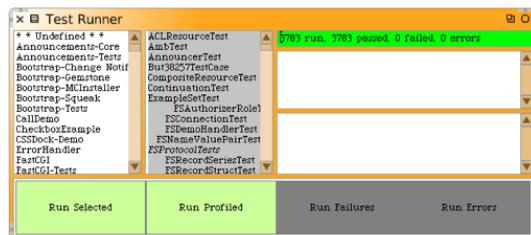


Figure 4.14: Making sure tests pass.

Chapter 5

GNU Smalltalk

by Paolo Bonzini, developer of GNU Smalltalk

In this section we describe how to get started developing a Seaside application in GNU Smalltalk. We assume that you already have the 3.0a version (or a later one) of GNU Smalltalk installed. This is the latest release at the time of this writing, and the first to include support for Seaside.

To download it, follow instructions at <http://smalltalk.gnu.org/download>.

5.1 Creating a GNU Smalltalk image with Seaside loaded

Seaside support is split into a number of separately loadable packages. Here are the ones that are available in GNU Smalltalk 3.0a:

- Seaside-Core
- Seaside-Adapters-Swazoo
- Seaside-Development
- Seaside-Examples

The Seaside package is a collective package that loads the first two.

First of all, you should create a new image with the package loaded. In GNU Smalltalk, the image acts as a kind of cache and preloading the package will speed up further operation.

In the remainder of this section, \$ is used as the prompt for things you type at the shell, and st> is used as the prompt for things you type for GNU Smalltalk.

```
$ gst
st> PackageLoader fileInPackage: 'Seaside'
st> PackageLoader fileInPackage: 'Seaside-Development'
st> PackageLoader fileInPackage: 'Seaside-Examples'
st> ObjectMemory snapshot: 'seaside.im'
```

5.2 Operating the GNU Smalltalk virtual machine remotely

At the end of the previous section you created a new image from the GNU Smalltalk read-eval-print loop. If you're familiar with other Smalltalk, it is a sort of console-based Transcript; if you're familiar with other scripting languages you will have already recognized it.

From now on, however, you will run Seaside applications within a remote-controlled instance of GNU Smalltalk, running in background as a daemon. The following three commands start the daemon, print the daemon's process id, and finally stop the daemon.

```
$ gst-remote -I seaside.im --daemon
$ gst-remote --pid
$ gst-remote --kill
```

The first command has the `-daemon` command-line option, and hence starts an instance of GNU Smalltalk that will run in the background and will be used to serve web pages. The other two don't have the command-line option, and all they do is interacting with the background instance of GNU Smalltalk. Note that you don't need to specify the image unless you are starting the background instance of GNU Smalltalk, because only the background virtual machine needs to have the Seaside packages loaded.

Now, let's start the daemon again and also start the web server:

```
$ gst-remote -I seaside.im --daemon
$ gst-remote --start=Seaside
```

Seaside is now serving on `http://localhost:8080/`; the entry address is by default `http://localhost:8080/seaside`. Try visiting the `http://localhost:8080/seaside/examples/counter` URL to make sure that the system works.

You can stop and restart the server without killing the daemon by running:

```
$ gst-remote --stop=Seaside
```

Be sure to restart serving web pages after running this command.

Another common operation is loading a file into the remote GNU Smalltalk instance. You do this using the following expression.

```
$ gst-remote --file FILENAME.st
```

You can also control an instance of GNU Smalltalk that's running in background using `gst-remote -eval`. After `-eval` you put a Smalltalk command that is executed within the server, for example:

```
$ gst-remote --eval '100 factorial printNl'
```

will compute $100!$ in the background image and print the result.

`gst-remote` supports running commands on a virtual machine running on a different machines, by specifying a hostname right after the command name itself. Note that, in this case, arguments to `-file` still refer to paths on the *local* machine.

5.3 Developing in GNU Smalltalk

Developers who are already familiar with other scripting languages, for example Ruby, will have few problems adapting to GNU Smalltalk.

A relatively important difference from other Smalltalk dialects is the availability of namespaces, a mechanism for restricting the referential scope of such names. While advanced usage of name spaces can be quite involved, in practice it is relatively simple and based on a few essential points.

- The top-level name space is named Smalltalk.
- All classes and add-ons are defined in sub-namespaces of Smalltalk.
- Most Seaside classes are in the namespace named Seaside.

Your application should, in general, be in a namespace that you create for your own usage. If you use the package system of GNU Smalltalk, switching to a separate namespace is actually done automatically while loading the package.

However, for simplicity especially during early development, you can define your classes directly in the Smalltalk name space.

5.4 Developing your first component

To illustrate the above points, we will go through the initial steps of developing the Counter example in GNU Smalltalk. We'll use a slightly different name so we won't conflict with the example already loaded with Seaside.

Place the following code in a file:

```
Seaside.WAComponent subclass: WebCounter [
    | count |
    WebCounter class >> canBeRoot [ ^true ]

    initialize [
        super initialize.
        count := 0.
    ]
    states [ ^{ self } ]
    renderContentOn: html [ html heading: count ]
]
```

```
WebCounter registerAsApplication: 'webcounter'
```

You can see that, apart from some syntactic sugar, the above is just Smalltalk as in any other dialect, except that method bodies are surrounded by square brackets. `count` declares an instance variable.

Here are the few concepts that the above basic component highlights:

- A Seaside component, is defined by a subclass of `WAComponent` class. Accordingly, to create a component, the above file creates the corresponding class.
- Each component is responsible for rendering itself in the web browser. This is done in Seaside by implementing a `renderContentOn:` method. In this simple case, that means displaying the counter value.
- Registering the application with Seaside makes its entry point is accessible at a URL path.

Registering an application is done in two steps. First, we must declare that our application component can be a root component, by defining a `WAComponent class>>canBeRoot` class method. Second, we must register the component as an application, which is done by the final line of the file. Code that is outside a class declaration corresponds to a *doit* (or, if you are not coming from Smalltalk, is just evaluated as the file is parsed).

After loading the code in a running server:

```
$ gst-remote --file Counter.st
```

the application will be visible at `http://localhost:8080/seaside/webcounter`. At the moment however it is pretty boring, so we improve the `WAComponent>>renderContentOn:` method like this:

```
renderContentOn: html [
    html heading: count.
    html anchor callback: [ count := count + 1 ]; with: '++'.
    html space.
    html anchor callback: [ count := count - 1 ]; with: '--'.
]
```

Reload the code in the server using the same `gst-remote` invocation as above. Go back to the web browser and refresh the page: there are now two anchors with increment (++) and decrement (–) links that call back to the actions.

Click the new links a few time to verify that it works. That's it. You developed your first Seaside component.

Chapter 6

VA Smalltalk

by John O'Keefe, Principal Software Architect, Instantiations Inc.

In this section we will show you how to get started developing Seaside applications with VA Smalltalk. First we will discuss how to obtain and install Seaside. Then we will show you how to develop your first application: a simple counter.

6.1 Loading Seaside into VA Smalltalk

Seaside was first delivered as part of the VA Smalltalk V8.0 release. You can download the latest VA Smalltalk release from the Instantiations download site at <http://www.instantiations.com/VAST/download/>. After installing VA Smalltalk, start the development image.

Seaside support is provided as a loadable feature in VA Smalltalk. From the Transcript, select *Tools | Load/Unload Features...* to open the Feature Loader, see Figure 6.1.

Scroll down to the *ST: Seaside Core* feature, select it, and use the *>>* button to mark it for loading. You can see other Seaside features in the list that you may also want to load. When you have selected all the features you want to load, and have moved them to the Loaded features list, select the *OK* button to load the features into your image.

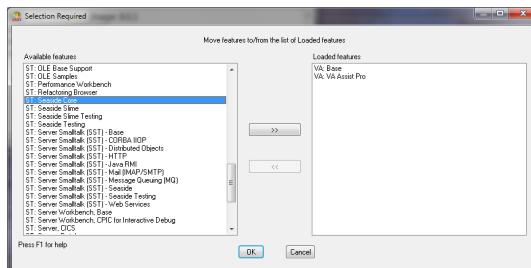


Figure 6.1: VA Smalltalk Feature Loader.

6.2 Starting VA Smalltalk Seaside

Once you have the VA Smalltalk image running and Seaside loaded, you can start a Seaside Server. From the Transcript, select *Tools | Open Seaside Control Panel* to open the Seaside Control Pane, see Figure 6.2.

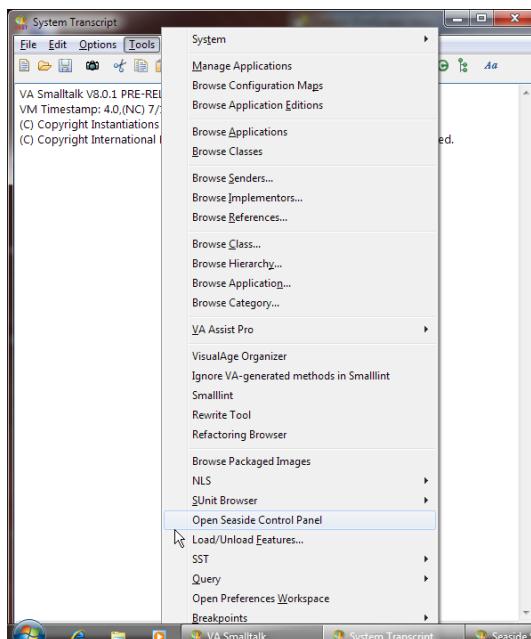


Figure 6.2: Open Seaside Control Panel.

6.2.1 Seaside Server Control Panel Menu Options

The Seaside Server Control Panel is the central control point for Seaside Servers. When the Seaside Server Control Panel is first opened it is empty, see Figure 6.3.

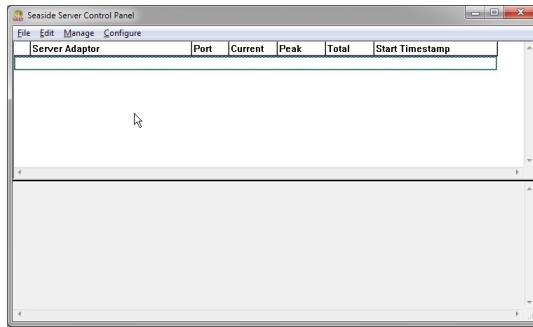


Figure 6.3: Initial Seaside Server Control Panel.

The menu on the control panel provides options to manage server adaptors and to configure the control panel. The options are:

Manage

<i>Add adaptor...</i>	Create a new server
<i>Start All</i>	Start all the servers that are not currently running
<i>Stop All</i>	Stop all the servers that are currently running
<i>Start</i>	Start the selected server(s)
<i>Stop</i>	Stop the selected server(s)
<i>Remove</i>	Remove the selected server(s) – they must be in the stopped state
<i>Inspect</i>	Open a VA Smalltalk inspector on the selected server
<i>Use new dispatcher</i>	
<i>Clear configuration caches</i>	
<i>Clear sessions</i>	

Configure

<i>Default port ...</i>	The default port used when adding a server adaptor
<i>Control Panel refresh interval ...</i>	The frequency of refreshing the control panel information. It is specified in seconds; a value of 0 turns off refresh.
<i>Reset to defaults</i>	Reset the default port to 8080 and the refresh interval to 5 seconds

6.2.2 Adding a Server Adaptor

Select *Manage | Add adaptor ...* to add an adaptor that will serve Seaside pages, see Figure 6.4. The added Server Adaptor will be in the stopped state.

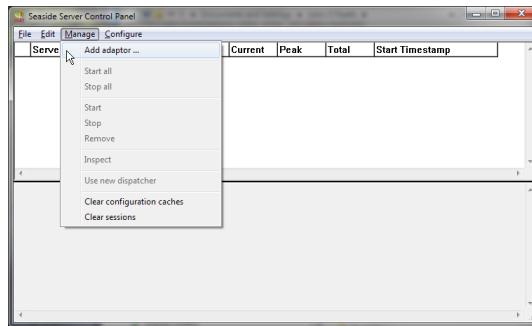


Figure 6.4: Seaside Server Control Panel Add Adaptor.

In the Add Server Adaptor dialog you can select the Server Adaptor class (currently only one class is available) and the port used to connect to the server, see Figure 6.5.

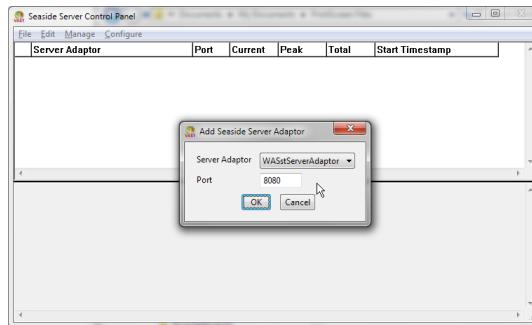


Figure 6.5: Add Seaside Server Adaptor Dialog.

You can add multiple server adaptors which can listen on different ports.

6.2.3 Starting a Server Adaptor

Once you have added a server adaptor to the Seaside Server Control Panel, you can start that server adaptor. Select the server you want to start; then select *Manage | Start* from the menu bar, see Figure 6.6. Alternatively you might want to use *Manage | Start All* to start all the server adaptors.

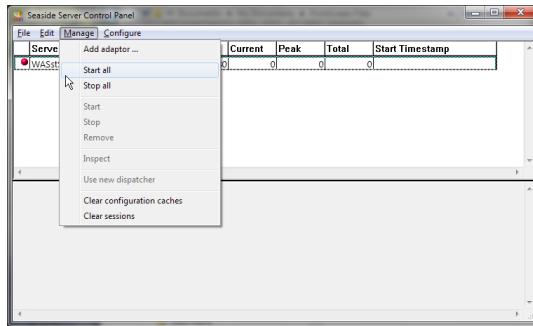


Figure 6.6: Start Server Adaptor.

At this point Seaside is running and ready to serve web pages, see Figure 6.7.

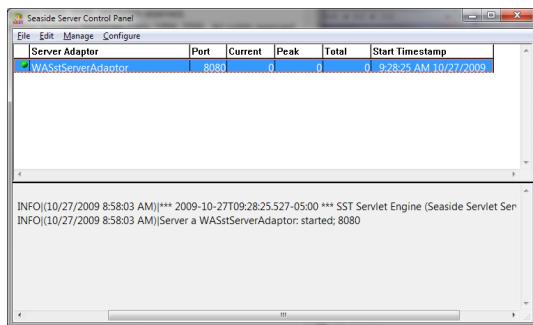


Figure 6.7: Started Server Adaptor.

6.2.4 A Simple Seaside Example

At this point you have a running Seaside Server. Now you can try a simple example shipped with Seaside. In a browser, enter the following URL: <http://localhost:8080/examples/counter> to see the pre-built counter application. You should see something like the page presented in Figure 6.8.

Selecting the ++ or – links will increment or decrement the counter. Later in this section you will build a Seaside application similar to this example.

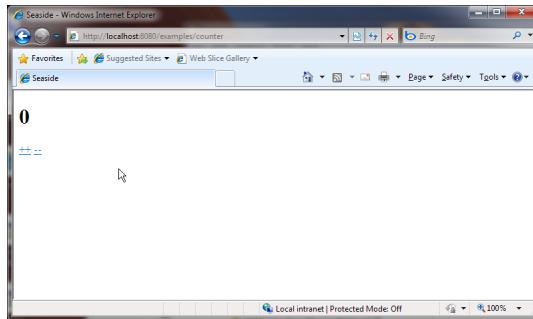


Figure 6.8: Sample Counter.

6.3 Developing Your First Seaside Component

Now you are ready to write your first Seaside component. You will code a simple counter in several steps: define an application and a class, define state, and define how the component is rendered. Finally you will declare it as a Seaside application.

6.3.1 Defining a Component

First we will define a new VA Smalltalk application to hold our simple counter. We create the SampleSeasideApp application using VA Smalltalk's Applications Manager. Select *Applications* | *Create* | *Application...* to specify the name of the new application, see Figure 6.9.

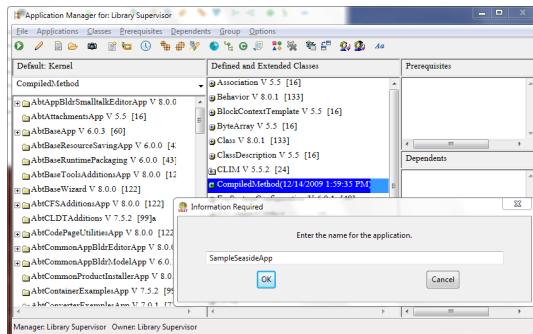


Figure 6.9: VASmalltalkApplicationManager.

You will also need to specify the prerequisite applications for your new

application. We are going to use the `WAAdmin`' tool to manage the application, so you should specify its application (`SeasideToolsCoreApp`) as the only direct prerequisite of your new application.

Now we define a new Seaside component named `WebCounter` by defining a subclass of `WACOMPONENT`. You can do this by selecting *Classes | Add | New Class....* Select `WACOMPONENT` as the superclass of your new class and select `OK`; enter the name of your new class (`WebCounter`) and select `OK`; finally, select `subclass` as the type of subclass you are creating and select `OK`. At this point you have created an application with 2 classes in it, `SampleWebApp` and `WebCounter`.

Now open a browser on the `WebCounter` class and add an instance variable that will contain the state of the counter. The class definition should look like this:

```
WACOMPONENT subclass: #WebCounter
  instanceVariableNames: 'count'
  classVariableNames: ''
  poolDictionaries: ''
```

6.3.2 Adding Some Methods

Now we define some instance methods to initialize and change the value of the counter.

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

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

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

6.3.3 Rendering a Counter

Now we define the instance method `renderContentOn:` to display the counter as a heading. Seaside will call such a method when it needs to display a component in the web browser. In the following method we just say that we want to display the value of the variable `count` using a heading HTML tag.

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

As you see, in Seaside you do not directly write HTML but rather you use a higher-level interface that models HTML. This helps you avoid making HTML mistakes. We will go into much more detail on this subject later in the book.

6.3.4 Registering the Counter Component

Now we should register the component as an application so that we can access it directly from the url path that will be associated with it. To register a component as an application, we ask the administration interface class `WAAdmin` to do the work for us:

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

will register the component `WebCounter` as the application named `webcounter`.

Note that this expression can also be added in the `WebCounter` class method `initialize` which is invoked when the class is loaded into memory:

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

Now you can launch the application in your web browser by going to `http://localhost:8080/webcounter`, see Figure 6.10.

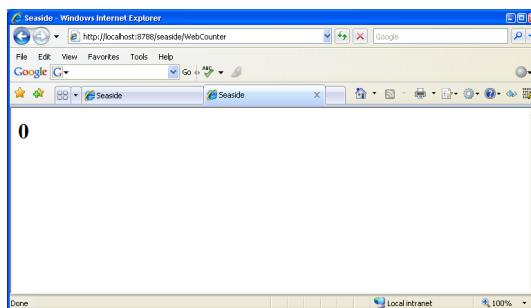


Figure 6.10: Simple-counter.

6.3.5 Adding Behavior to the Counter

Now we can add some actions by defining callbacks attached to anchors. We will explain actions in detail later in this book. A callback is a piece of code that will be executed when a link is clicked. For now this is just to give you a feel of Seaside programming. We update the `WebCounter>>renderContentOn:` method as follows.

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

After saving your changes, refresh the browser page and you will now see the increment and decrement links. Click on the links so see that the counter get incremented or decremented, see Figure 6.11.

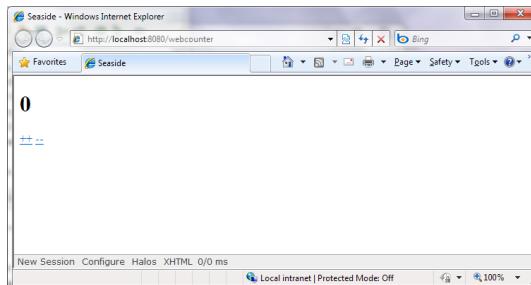


Figure 6.11: VAST sample counter with actions.

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 7

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.

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: `
`. Some people use `
` or `
</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 Smalltalk). 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 Smalltalk 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.

7.1 Rendering Hello World

Our first Seaside component will simply display *Hello world*. Begin by creating a category called 'SeasideBook-Hello' and then create the class `ScrapBook` as a

subclass of `WAComponent` as shown below.

```
WAComponent subclass: #ScrapBook
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: '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 Figure 7.1.

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.



Figure 7.1: Hello World in Seaside.

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.

7.2 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 7.2.

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

```
html image url: 'http://www.seaside.st/styles/logo-plain.png' ]
```

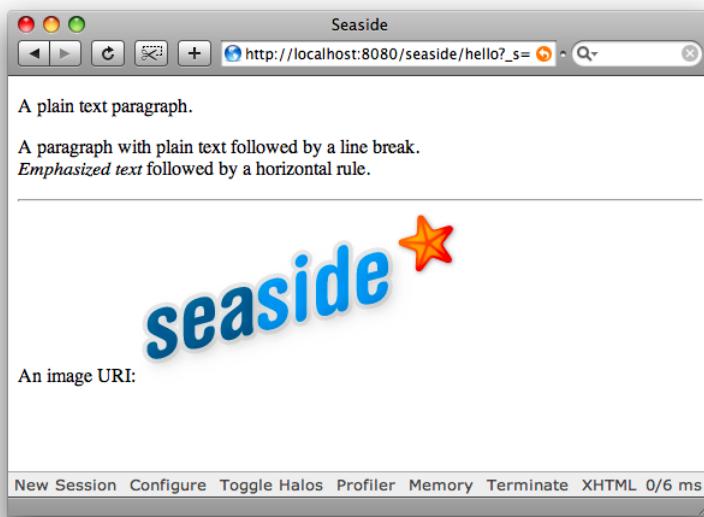


Figure 7.2: Some simple XHTML elements in Seaside.

You created two paragraphs, added some text, a break, a horizontal rule and an image. But notice that you did not edit any tags directly and you generated valid XHTML! In the following sections we will analyze what we did in detail but for now let's continue to explore what Seaside has generated for us.

Sometimes you would like to know exactly what XHTML elements Seaside is generating for you. Try to use your web browser to view the XHTML source for your `ScrapBook`. You'll find that it is not particularly readable since it is not formatted for human readers (no line-feeds, indentation, etc) and it contains much more than your single component XHTML. Not to worry, Seaside has a great tool called the *halos* that can be used to get to a display of nicely formatted XHTML source code of all the components displayed on a page and do much more as well. At the bottom of your web browser's window you should see a tool bar (see Figure 7.3). The tool bar contains tools that are available in development mode. For now you just have to know that such buttons let you interact with the tools. Note that depending on your Seaside version you may have different tools.

Halos let you interact directly with the components you are editing. Click the "Halos" link and notice that a border, or 'halo', appears around your

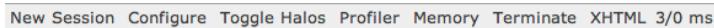


Figure 7.3: The Seaside tool bar.

component's visual representation. Figure 7.4 shows the component and its halo. Figure 7.5 shows the html generated for the component currently displayed. Even if your component contained links or actions, you can activate them even when browsing the generated XHTML.



Figure 7.4: A component decorated with halos.

Your component is now displayed in the web browser but it is decorated with a border. On the top you will now see a number of icons and links. For now let's focus on the links **Render**/**Source** on the right. The bold **Render** means that you are currently seeing your component as normally rendered in your web browser, as seen in Figure 7.4. Pressing **Source** will show you the XHTML generated by Seaside for the component. Notice that what you see is just the XHTML for the current component and not the complete application. Notice that
 and <hr/> are valid!

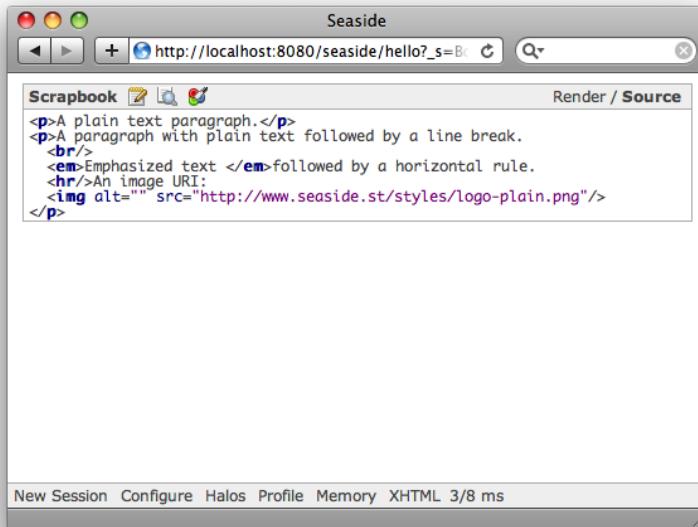


Figure 7.5: XHTML of a component generated by Seaside.

7.3 More Fun with the Seaside Canvas

Now let's have even more fun. Since Seaside uses plain Smalltalk rather than templates to build web pages, we can use Smalltalk to build the logic of our rendering method. We are only limited by our imagination and artistic taste. For example, suppose that we want to display 10 Seaside logos. We can simply use the `timesRepeat:` Smalltalk loop as shown in the next method. See the output in Figure 7.6.

```
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 ] ]
```

Since we're writing Smalltalk, changes are easy. In the next example, we added a horizontal rule inside the loop but noticed that it didn't look very nice (see Figure 7.7), so we moved it outside the loop (see Figure 7.8).

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

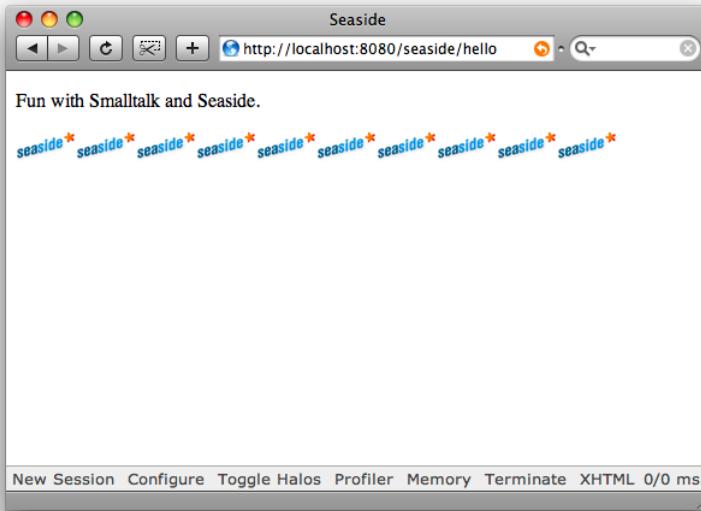


Figure 7.6: Using Smalltalk to write a loop in our rendering method.

```
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 ]
```

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

7.4 Rendering Objects

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

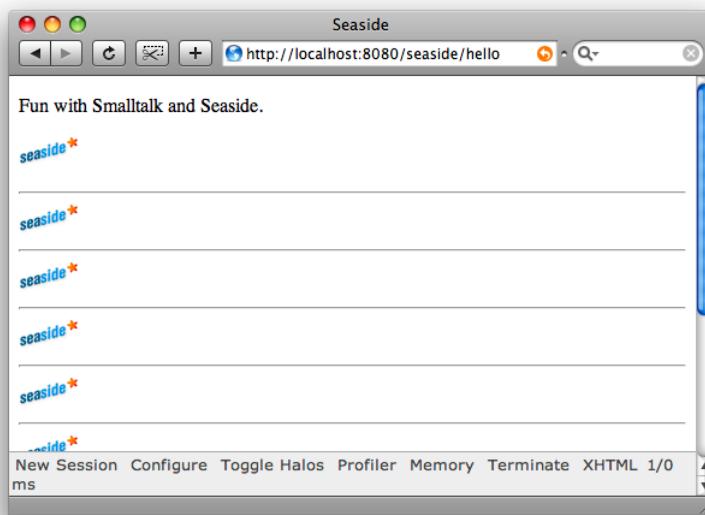


Figure 7.7: Rendering images vertically.

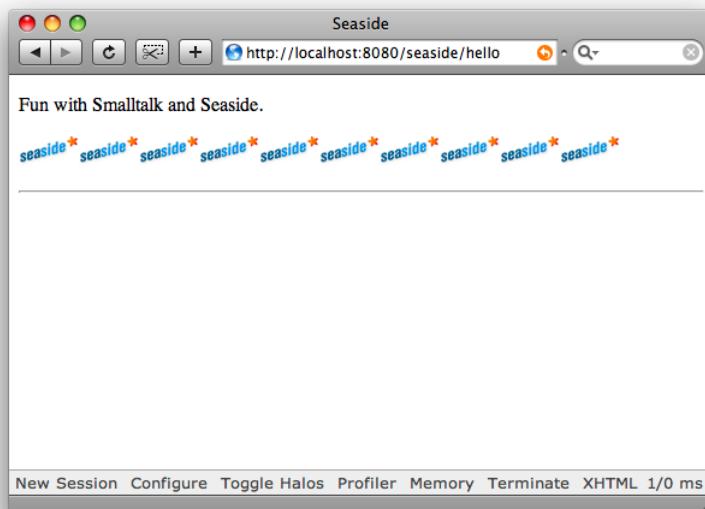


Figure 7.8: Rendering images horizontally.

```
ScrapBook>>renderContentOn: html
  html paragraph: 'A plain text paragraph.'.
  html paragraph: [
    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 7.5 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, see Figure 7.9. 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 12.

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



Figure 7.9: Rendering object with the #render: method.

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 7.10.

Two equivalent methods.

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

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

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

is equivalent to

```
html brush: anObject
```

Figure 7.10: Two equivalent forms.

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 Pharo and GemStone the method `text:` will call the method `Object>>asString`, while VisualWorks uses the method

`Object>>displayString`. 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!

7.5 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 Figure 7.11.

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::`.

Note

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*.

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

```

Figure 7.11: Select brush, configure it, and render it.

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:`.

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 Figure 7.12. There is a style checking tool called Slime that checks for such cases. Slime is explained in Chapter 14.

<pre>html brush with: []</pre> <p>is equivalent to</p> <pre>html brush</pre>	<pre>html brush with: [html render: anObject]</pre> <p>is equivalent to</p> <pre>html brush: anObject</pre>
---	---

Figure 7.12: Brush Simplifications.

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

7.6 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 <code>WAHeadingTag</code>	Description
<code>level:</code>	Specify <code>anInteger</code> as the heading level for this brush.
<code>with:</code>	Render this heading tag with <code>anObject</code> 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
a	anchor	WAAnchorTag
a	map	WAIImageMapTag
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	WAEeditTag
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	WAIImageTag
input	cancelButton	WACancelButtonTag
input	checkbox	WACheckboxTag
input	fileUpload	WAFileUploadTag
input	hiddenInput	WAHiddenInputTag
input	imageButton	WAIImageButtonTag
input	passwordInput	WAPasswordInputTag
input	radioButton	WARadioButtonTag
input	submitButton	WASubmitButtonTag
input	textInput	WATextInputTag
ins	inserted	WAEeditTag
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	WARubyTextTag
rtc	rubyTextContainer	WAGenericTag
ruby	ruby	WAGenericTag
samp	sample	WAGenericTag
script	script	WAScriptTag
select	multiSelect	WAMultiSelectTag
select	select	WASelectTag
small	small	WAGenericTag

span	span	WAGenericTag
strong	strong	WAGenericTag
sub	subscript	WAGenericTag
sup	superscript	WAGenericTag
table	table	WATableTag
tag:	tag:	WAGenericTag
tbody	tableBody	WATableTag
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

Note

Smalltalk 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 `WABrush`. 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 Smalltalk 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
    WAAuthorTag
    WAImageMapTag
    WAPopupAnchorTag
    WAAudioTag
    WABasicFormTag
    WAFormTag
    WABreakTag
    WACanvasTag
    WACollectionTag
    WADatalistTag
    WAListTag
      WAOorderedListTag
      WAUnorderedListTag
    WASelectTag
    WAMultiSelectTag
  WACommandTag
  WADetailsTag
  WADivTag
  WAEeventSourceTag
  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
WAIImageButton
WAGenericTag
    WAEeditTag
WAHeadingTag
WAHorizontalRuleTag
WAIframeTag
WAIImageTag
WAKeyGeneratorTag
WALabelTag
WAMenuTag
WAMeterTag
WAOBJECTag
WAOPTIONGroupTag
WAOPTIONTag
WAParameterTag
WAProgressTag
WARubyTextTag
WAScriptTag
WASourceTag
WATableCellTag
    WATableColumnGroupTag
        WATableColumnTag
    WATableDataTag
        WATableHeadingTag
WATableTag
WATimeTag
WAVideoTag
```

7.7 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 Figure 7.13.

As good Smalltalkers 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 ] ]
```

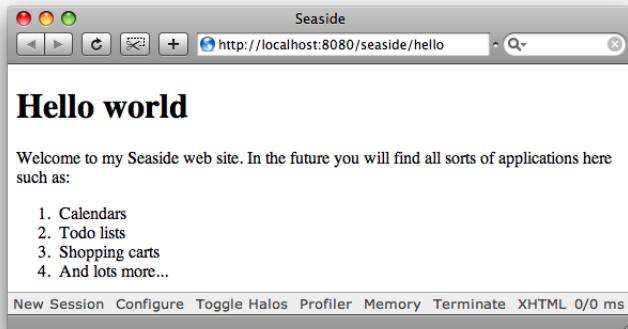


Figure 7.13: A list of items.

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.

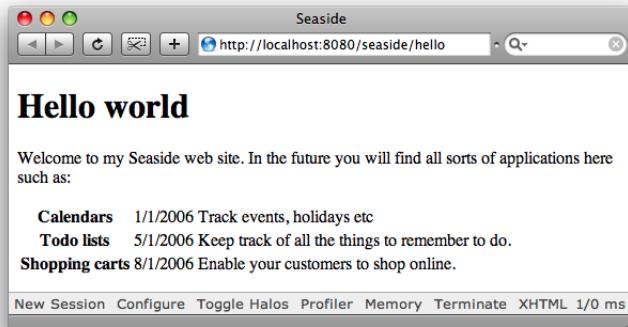


Figure 7.14: A table of items.

7.8 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.
- 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 8 or file library style sheet as described in Chapter 17.

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 Chapter 8 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 17.
- 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.

7.9 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 8

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.

8.1 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 8.1. 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.

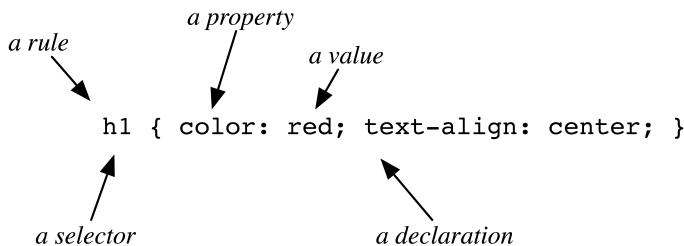


Figure 8.1: Essential CSS structural elements.

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

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 Section 23.3.6. For rapid prototyping, you can define the CSS of a component by overriding its `WACOMPONENT>>style` method.

8.2 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.

8.2.1 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; }
```

8.2.2 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 ]
```

8.2.3 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.

8.2.4 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. See Part V on Javascript programming.

Important

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.

8.3 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 `div` 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.highlight`. 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 `div` tag that have the CSS class `error`. Elements that have the class `error` but no `div` 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 `div` 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.

8.4 Summary

Styling web applications is a broad topic. This chapter has shown you the most important things to get started. For most people this will be enough to get a decent looking prototype up and running. For commercial applications you will often hire a graphic designer with experience of designing web pages. Once such designers have worked with you on the desired look and feel of your pages, they can provide you with a purpose built style-sheet. Using the techniques described in this chapter you will be able to integrate such style-sheets with the content generated by your application to make it look beautiful.

We have found the following two documents to be helpful in learning more about XHTML and CSS:

XHTML Specification <http://www.w3.org/TR/xhtml1/>

CSS Level 2 Specification <http://www.w3.org/TR/REC-CSS2/>

There are many CSS resources on the web. We've accumulated a long list but here are a few that stand out as sites that we repeatedly visit.

CSS Zen Garden <http://www.csszengarden.com/>

A List Apart <http://www.alistapart.com/>

Blueprint CSS <http://www.blueprintcss.org/>

The Layout Reservoir <http://www.bluerobot.com/web/layouts/>

Subtraction <http://www.subtraction.com/>

Chapter 9

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 (Smalltalk 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.

9.1 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: ''
  poolDictionaries: ''
  category: '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 Figure 9.1.

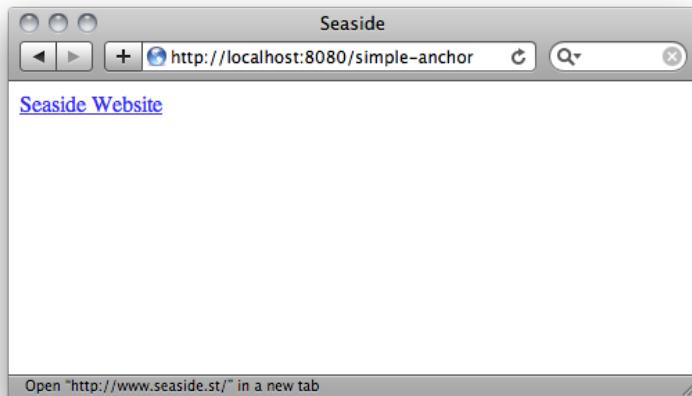


Figure 9.1: A simple anchor.

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

9.2 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 Smalltalk 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 *evaluates the block* in Smalltalk terminology).

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: ''
  poolDictionaries: ''
  category: '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 `WAAuthorTag` which is then used to set the callback (via the method `callback:`) and text for this anchor (via the message `with:.`).

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 9.2. Clicking on the link will increment the count.

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

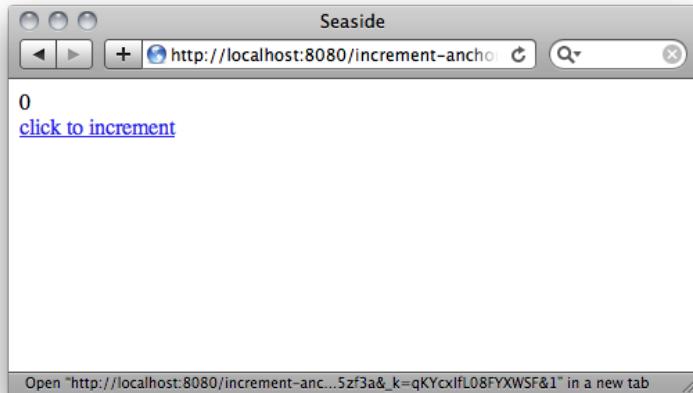


Figure 9.2: Using a callback.

Important

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.

9.3 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).

Important

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:

Important

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.

9.4 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 category `iAddress`, and creating in this category a class whose instances will represent the contacts in our address book.

```
Object subclass: #Contact
  instanceVariableNames: 'name emailAddress'
  classVariableNames: 'Database'
  poolDictionaries: ''
  category: '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 category.

```
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
```

9.5 Listing the Contacts

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

```
WAComponent subclass: #ContactListView
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'iAddress'
```

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

In some Smalltalks, saving this method will raise a warning, telling you that the selector `renderContact:on:` is unknown. This is because we haven't yet defined the method with that name, but you should confirm that this name is correct because we will define it in the next snippet:

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

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 9.3.

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

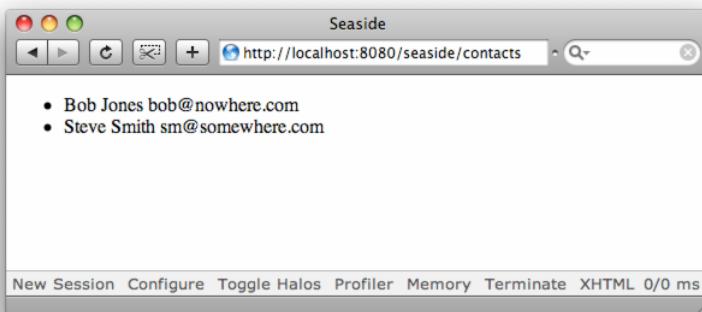


Figure 9.3: Displaying the contact database.

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.

Advanced

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.

9.6 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 con-

tact':

```
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 Figure 9.4.

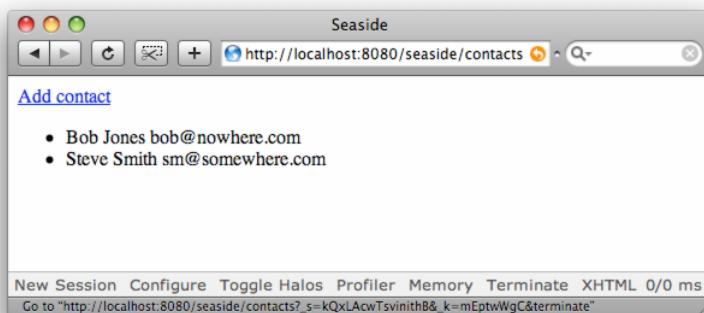


Figure 9.4: Contact list with Add contact link.

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 Part III. Note that a real application would present a form with several fields to be filled up by the user.

9.7 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 9.5.

```
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
```

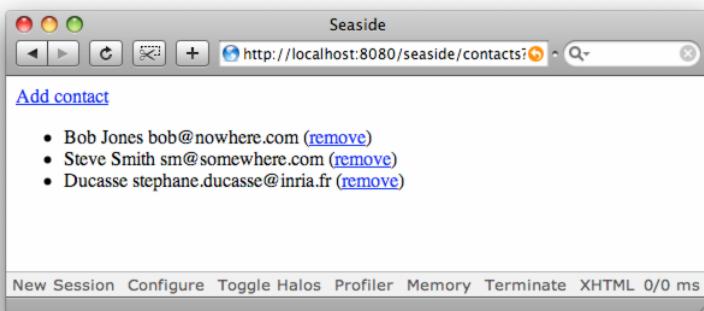


Figure 9.5: Contacts can now be removed.

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 9.4.

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 Figure 9.6).

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.



Figure 9.6: With a removal confirmation.

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 Part III.

9.8 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 Section 9.1, 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 9.7.

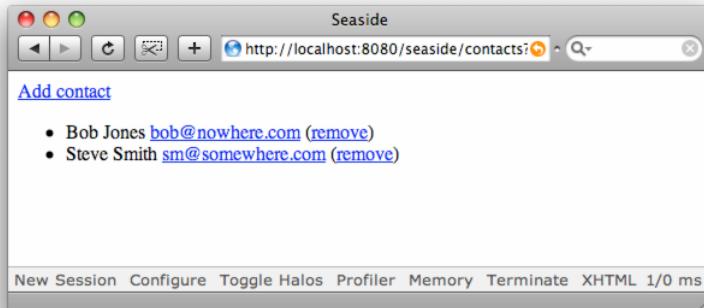


Figure 9.7: With mailto anchor.

Note that rather than manipulating strings in this way, experienced Smalltalkers 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 `WAEEmailAddress` which may be used for this very purpose.

9.9 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 Smalltalk code in a callback. In the next chapter, we

will continue to enhance the iAddress application to show you how to handle forms.

Chapter 10

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 `form` element. In this chapter we show how to create this `form` element and how to use these common controls.

10.1 Text Input Fields and Buttons

Let's continue with the same `Contact` domain model class that we used in Chapter 9. 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: ''
  poolDictionaries: ''
  category: '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.

Note

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.

Important

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 textField
            callback: [ :value | self contact name: value ];
            value: self contact name.
        html break.
        html text: 'Email address: '.
        html textField
            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
```

Note

In Seaside 3.0, 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 2.4.5 for details. Point your web browser to `http://localhost:8080/contact` and you should see the form as shown in Figure 10.1. 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.

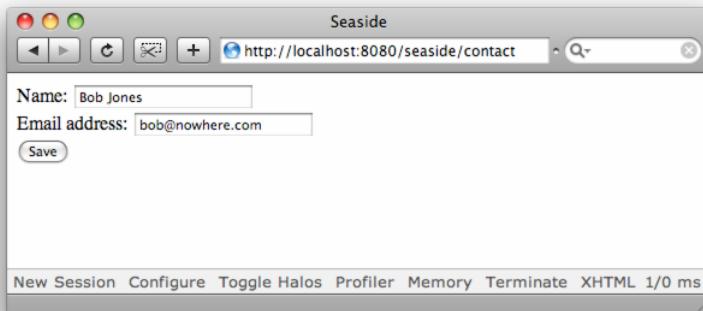


Figure 10.1: Filling up our contact view.

Methods on <code>WATextInputTag</code>	Description
<code>callback: aBlock</code>	Specify a single argument callback block which is passed the string value entered by the user.
<code>value: aString</code>	Specify an initial value for the text input.
<code>on: aSymbol of: anObject</code>	This is a convenience method explained in the next section.
Methods on <code>WASubmitButton</code>	Description
<code>callback: aBlock</code>	Specify a zero-argument callback block which is invoked when the user clicks this button.
<code>value: aString</code>	Specify a label for this submit button.
<code>on: aSymbol of: anObject</code>	This is a convenience method explained in the next section.

10.2 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 textField on: #name of: self contact
```

instead of:

```
html textField
  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'
```

Note

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 textField on: #name of: self contact.
        html break.
        html text: 'Email address:'.
        html textField 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:`.

Important

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).

10.3 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'
    poolDictionaries: ''
    category: '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 10.2.

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

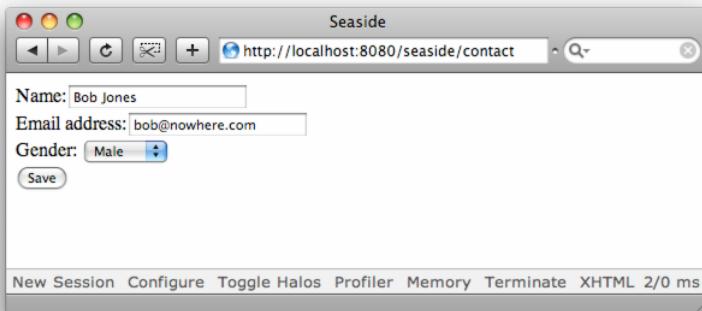


Figure 10.2: Gender as a drop-down menu.

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

Advanced

Experienced Smalltalkers will be getting concerned at the length of this method by now. Generally it is considered good practice in Smalltalk 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 Figure 10.3.

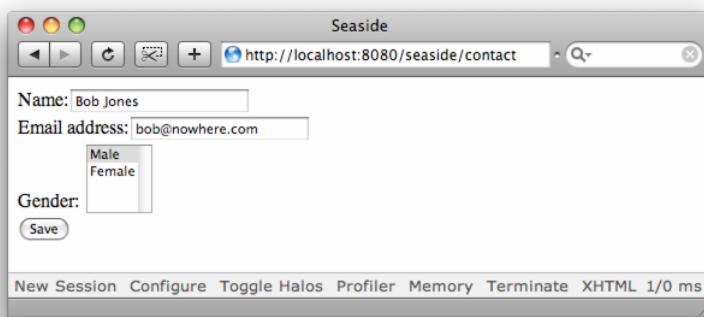


Figure 10.3: Gender as a list.

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

Methods on <code>WASelectTag</code>	Description
<code>list: aCollection</code>	Specify the list of options from which to select.
<code>selected: anObject</code>	Specify the object which should be shown as selected by default.
<code>callback: aBlock</code>	Specify a single-argument callback block which will be passed the object selected by the user.
<code>size: anInteger</code>	Specify the number of rows of the list that should be visible. Note, if you don't specify a size, the default on most browsers will be to use a drop-down menu. If you specify a size then most browsers will present the options in a list box.
<code>on: aSymbol of: anObject</code>	This is a convenience method as explained previously.

10.4 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.

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

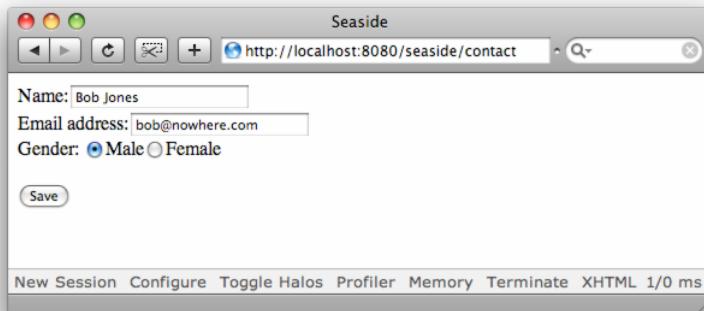


Figure 10.4: With radio buttons.

Methods on <code>WARadioButtonTag</code>	Description
<code>group: aRadioButtonGroup</code>	Specify the radio group to which this button belongs.
<code>selected: aBoolean</code>	Specify a boolean value that indicates whether this radio button is initially selected.
<code>callback: aBlock</code>	Specify a zero argument callback block which is called if this button is selected when the submit button is pressed.

10.5 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'
  poolDictionaries: ''
  category: '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 textField on: #name of: self contact.
    html break.
    html text: 'Email address:'.
    html textField 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 10.5 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:

```
html textField 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

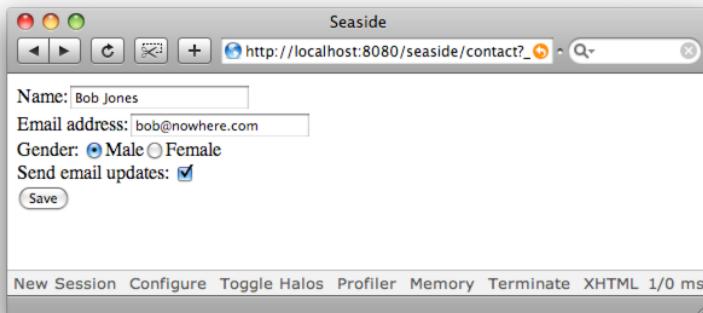


Figure 10.5: With the checkbox.

`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 <code>WACheckboxTag</code>	Description
<code>callback: aBlock</code>	Specify a one-argument callback block which will be passed true or false.
<code>on: aSymbol of: anObject</code>	This is a convenience method.
<code>onTrue: aBlock onFalse: aBlock</code>	Specify two zero-argument blocks. One is performed if the box is checked when the form is submitted, and the other if the box is not checked.
<code>value: aBoolean</code>	Specify the initial value for the checkbox.

10.6 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 textField on: #name of: self contact.
    html break.
    html text: 'Email address:'.
    html textField 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
```

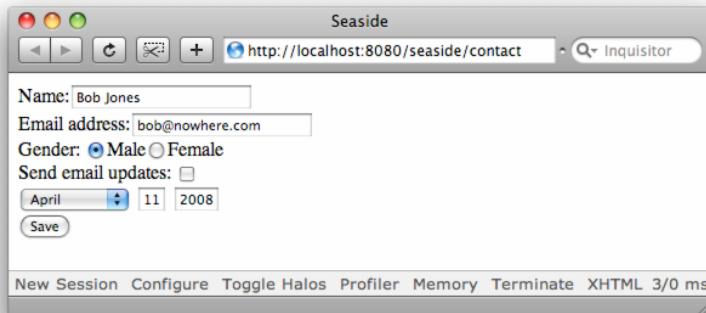


Figure 10.6: Full contact view.

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 Figure 10.6.

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

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

10.7 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 Figure 10.7. Some browsers may also display a text input field so you can enter the file path.

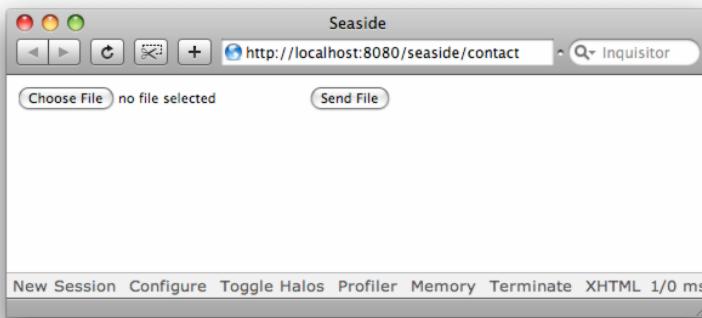


Figure 10.7: File uploads.

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 `WAFfile` object that represents your file. As an example, the method below will save the file in a directory called `uploads` in Pharo.

```
UploadForm>>receiveFile: aFile
| stream |
stream := (FileDirectory default directoryNamed: 'uploads')
assureExistence;
forceNewFileName: 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.

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 ]
```

10.8 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 Chapter 9, 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 Chapter 10 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 11

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.

11.1 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.

11.2 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 Figure 11.1), and when the user selects that link, display the `ContactView` on that Contact. We accomplish this using the `call:` message.

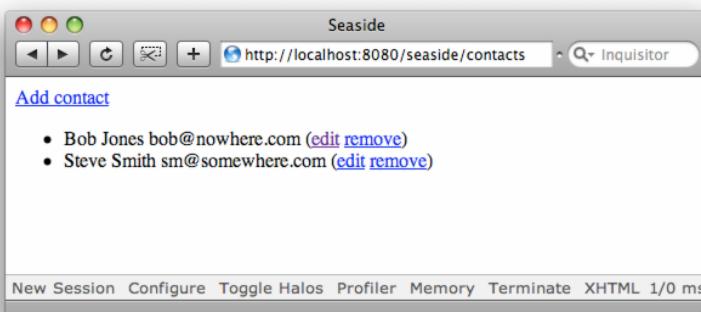


Figure 11.1: New version of the `ContactListView`.

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.

11.3 Call/Answer Explained

Figure 11.2 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.

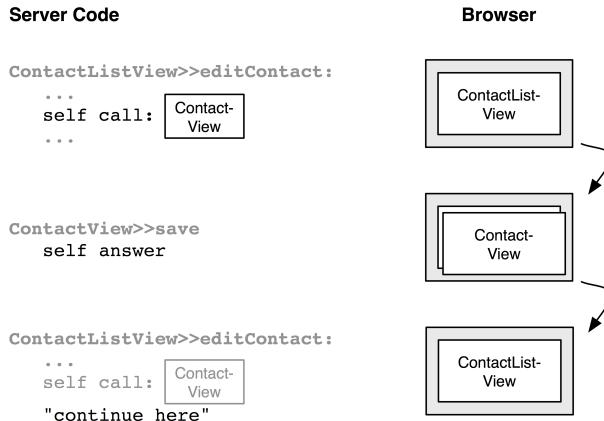


Figure 11.2: Call and Answer at Work.

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 as 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.

11.4 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 Section 11.8. For now just try it and confirm that things work as you'd expect.

11.5 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 Figure 11.3.

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
```

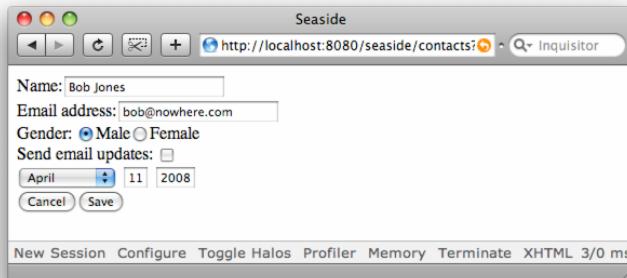


Figure 11.3: Contact edition with a cancel button.

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

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 ]
```

Note

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.

11.6 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 Chapter 12.

Important

Don't `call:` a component from `renderContentOn:.` only call components from callbacks or from `WATask` subclasses.

11.7 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.

11.8 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.

11.9 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 Section 11.9.1).

11.9.1 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 lead to messy code pretty quickly.

11.10 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 12

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 Section 11.1 that components can be sequenced. In this chapter we show how to embed one component inside another component. In Section 12.6, 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.

12.1 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 established, 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 stored.
3. The parent component renders its child components in the method `renderContentOn:` using `html render: aComponent`.

12.2 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.

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.

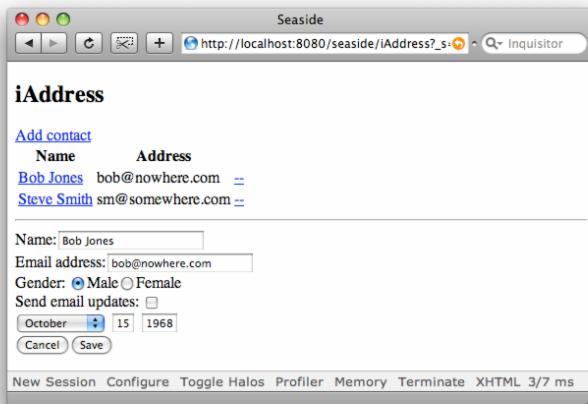


Figure 12.1: Embedding a ContactView into another component.

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

```
WAComponent subclass: #IAddress
  instanceVariableNames: 'editor'
  classVariableNames: ''
  poolDictionaries: ''
  category: '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 Section 12.7.

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 ]
```

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.

```
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 ]
```

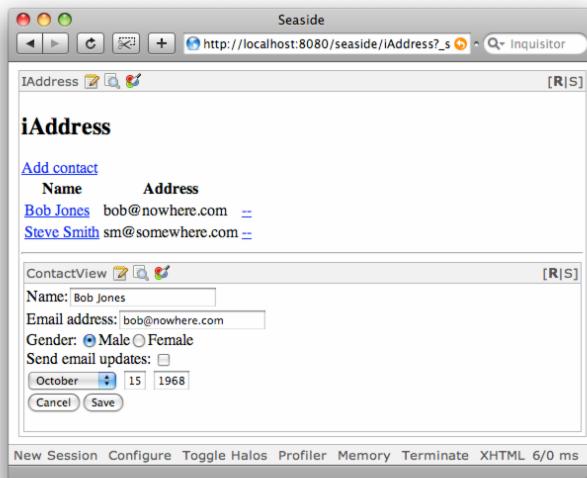


Figure 12.2: Embedding a ContactView into another component with Halos.

```
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 Figure 12.2. It is very

helpful to inspect the state of a component in a running application (or view the rendered HTML.)

Note

Our simple implementation of `IAddress>>editContact:` will save changes even when you press *cancel*. See Section 11.5 to understand how you can change this.

12.3 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 12.3 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.

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: ''
    poolDictionaries: ''
    category: '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.

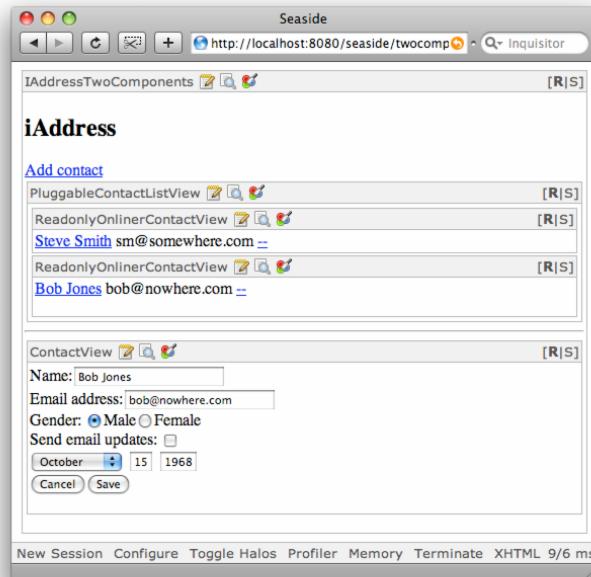


Figure 12.3: With components all the way down.

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

12.4 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.

12.5 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`.

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 Section 12.6.

```
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 Chapter 26. 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.

12.6 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



Figure 12.4: A readonly view of the ContactView.

rather as a *Chain of Responsibility*. This means that decorations form a chain 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.

12.6.1 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 Figure 12.5.

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

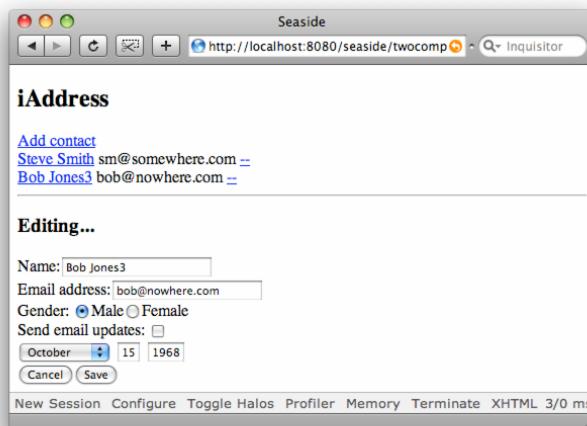


Figure 12.5: Adding a message around a component.

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 Figure 12.6.

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

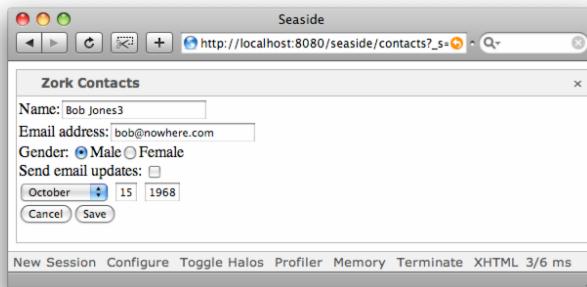


Figure 12.6: Decorating a component with a window.

Note

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.

```
WAPPlugin>>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 superfluous rendering calls from `renderContentOn:`, to leave simpler code and an unchanged application (see Figure 12.7).

```
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.

12.6.2 Behavioral Decorations

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.

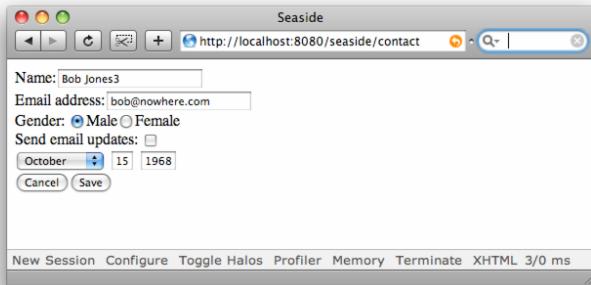


Figure 12.7: Using a decoration to add buttons and form to a ContactView.

```
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.

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

12.7 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 Chapter 18.

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: ''
poolDictionaries: ''
category: '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 `unsubscribe: anObject`.

Note

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.

12.8 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 13

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.

13.1 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.

Note

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: ''
poolDictionaries: ''
category: '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 13.1 shows a typical execution.

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

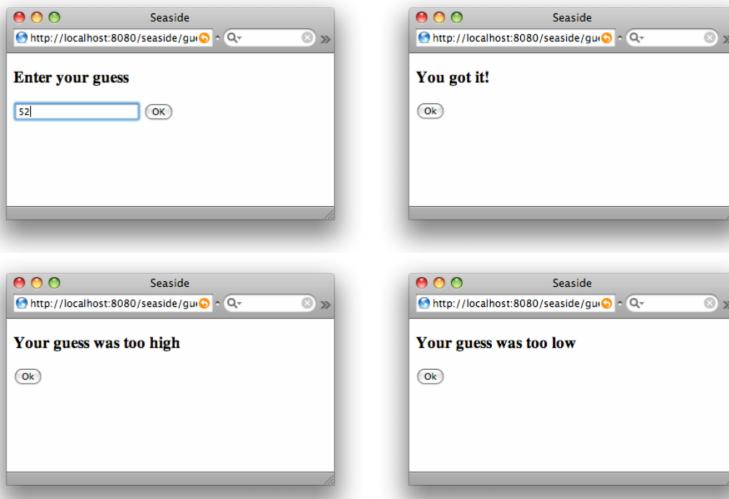


Figure 13.1: Guessing Game interaction.

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.

13.2 Hotel Reservation: Task vs. Component

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: ''
  poolDictionaries: ''
  category: '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 Figure 13.2.

```
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 , ''
```

13.3 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

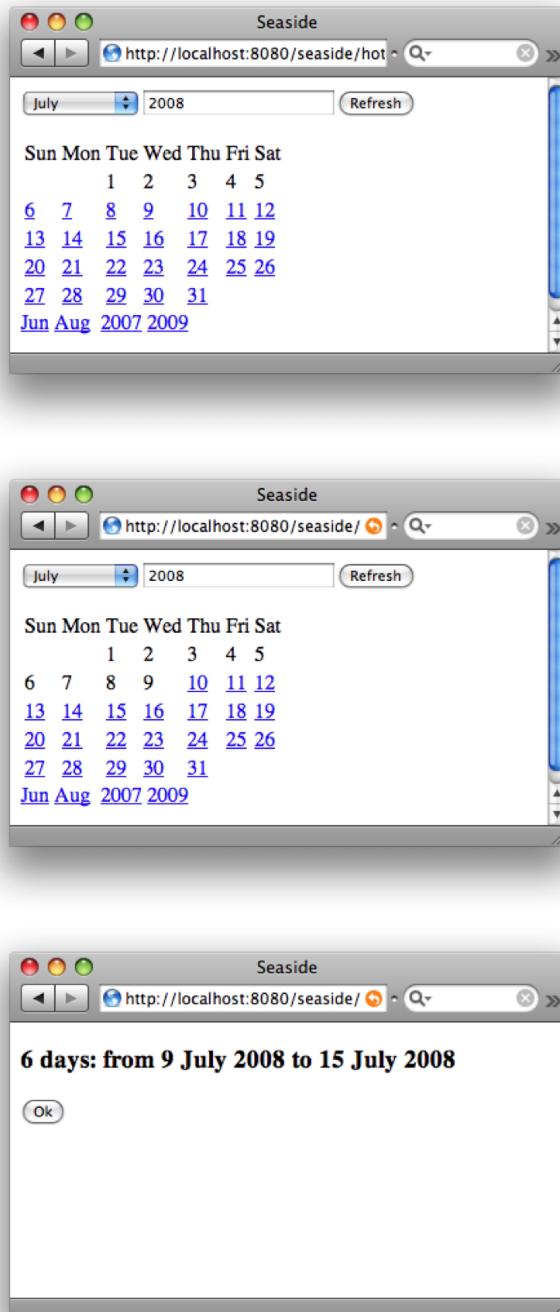


Figure 13.2: A simple reservation based on task.

always have the two mini-calendars visible on the same page and provide some feedback to the user as shown by Figure 13.3.

```
WAComponent subclass: #MiniInn
    instanceVariableNames: 'calendar1 calendar2 startDate endDate'
    classVariableNames: ''
    poolDictionaries: ''
    category: '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 , ' ' ]
```

13.4 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

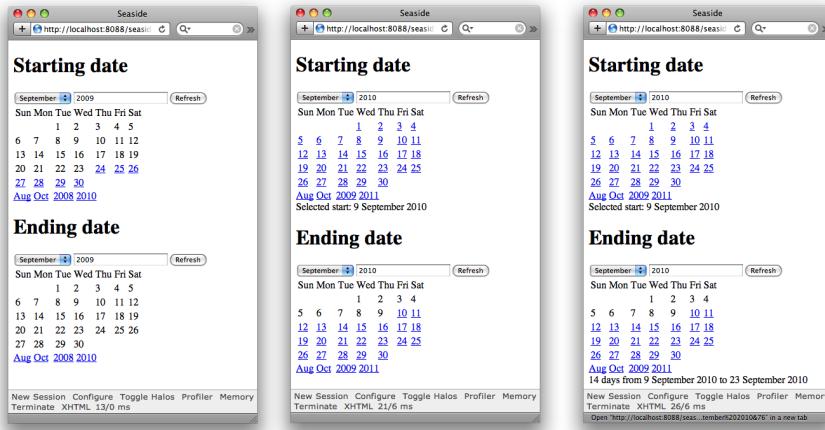


Figure 13.3: A simple reservation with feedback.

on the composition of other components. We saw that the composition is expressed in plain Smalltalk.

Chapter 14

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 7. 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.

14.1 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.

14.2 Slime at Work

Slime is not available on all Smalltalk platforms. To run Slime on Pharo follow these steps:

1. Open a scoped Browser. In most cases, you don't want to run Slime in the default System Browser, as this would run the checks on the complete image. To open Slime on a specific package, you need to open a scoped browser. Click on a class of your package and select *refactoring scope*, then select the menu item *package*.

You should obtain a browser that only shows the contents of the package. Any tool or analysis of the refactoring browser is scoped to the visible context. Other than that, this is a normal code browser that you can use to edit your code. Figure 14.1 shows that the analysis will be performed on the classes contained in the *Store* package.



Figure 14.1: A scoped browser onto the *Store* package.

2. Start the Code Checker. In the scoped browser, select *refactor* and then *code critics*. This opens a new window that starts to run all the Code Critics and Slime rules on the selected code.

The progress of the search is shown in the title bar of the Code Critics browser, depending on the size of the selected code the analyze might take a while. After a while the tool should update and display some categories and rules in bold, showing the number of detected problems as shown in Section 14.1.

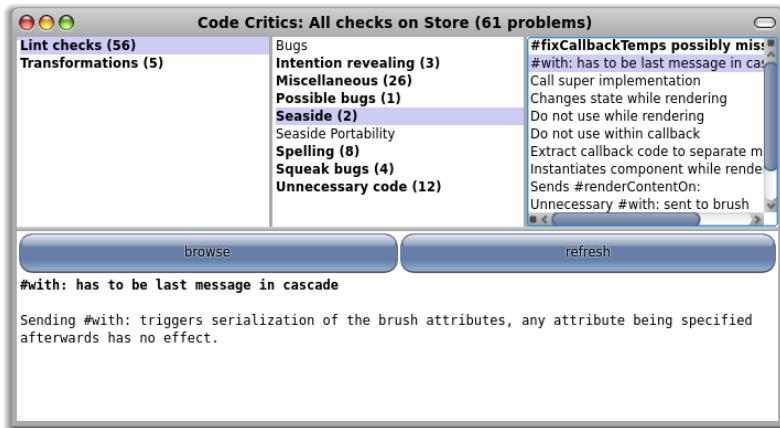


Figure 14.2: The Code Critics Result Browser.

3. Displaying the Problems. You can now start to walk through the list of detected problems. Note that this does not only list the Seaside specific problems, but also other more general problems. Most rules have a detailed description explaining the issue. When you select a bold entry and click on the open *open*, you can navigate to the actual problem in the code.

14.3 Summary

Slime offers the validation of your code and it will verify some coding practices. Once again, we suggest you run this tool often. This tools as well as your unit tests and the debugger are your best friends to produce good quality code.

Part IV

Seaside In Action

This part develops two little applications – a todo list manager and a sudoku player. Then it presents how to serve files in Seaside as well as character encodings and how to customize sessions to hold application centric information.

Chapter 15

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.

15.1 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 15.1.

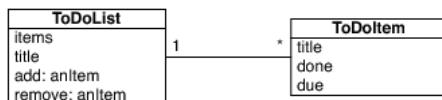


Figure 15.1: A simple model with items and an item container.

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: ''
  poolDictionaries: ''
  category: '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.
```

Important

A word about `initialize` and `new`. Squeak/Pharo is the only Smalltalk dialect that performs 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 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'
  poolDictionaries: ''
  category: '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.

15.2 Defining the 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: ''
poolDictionaries: ''
category: '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'
```

The screenshot shows a web browser window with the title "Seaside". The main content area displays a table of registered applications and dispatchers:

browse	Application	Configure Copy Remove
comet	Dispatcher	Configure Copy Remove
config	Application	Configure Copy Remove
examples	Dispatcher	Configure Copy Remove
files	File Library	Configure Copy Remove
tests	Dispatcher	Configure Copy Remove
todo	Application	Configure Copy Remove
tools	Dispatcher	Configure Copy Remove

Figure 15.2: The application is registered in Seaside.

You can see that the todo application is now registered by pointing your browser to `http://localhost:8080/config/` as shown in Figure 15.2.

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 Figure 15.3.



Figure 15.3: Our application is there, but nothing is rendered.

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

15.3 Rendering and Brushes

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

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

Note

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 Figure 15.4.

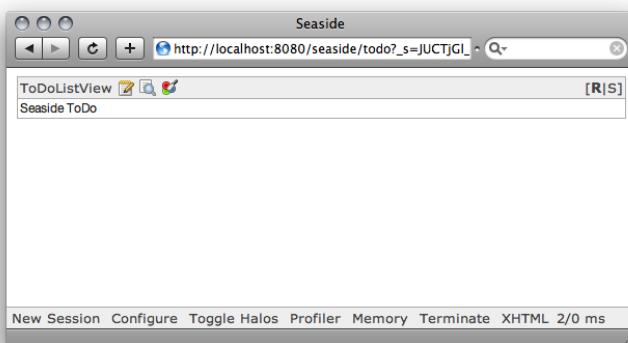


Figure 15.4: Our todo application simply displaying its title.

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

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

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

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 8 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 Figure 15.5.

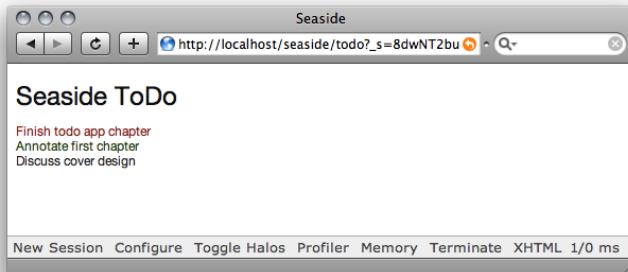


Figure 15.5: Our todo application, displaying its title and a list of its items colored according to status.

15.4 Adding Callbacks

As we saw in Chapter 9, Seaside offers a powerful way to define a user action: *callbacks*. We can use callbacks to make our items editable. To do this, we render two additional links with every item.

```

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 Figure 15.6.



Figure 15.6: TodoWithAnchors.

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 anchor under the title (see Figure 15.7):

```
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)
```

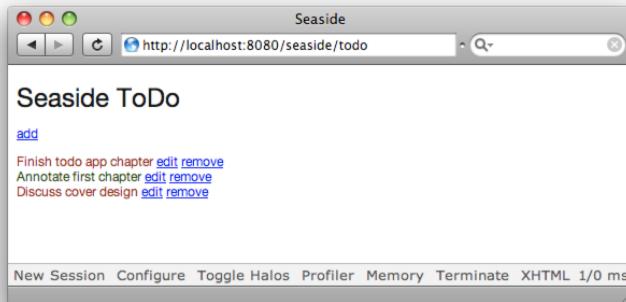


Figure 15.7: Our todo application with add functionality.

15.5 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 Chapter 10). Here is our updated `renderContentOn:` 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 Figure 15.8.

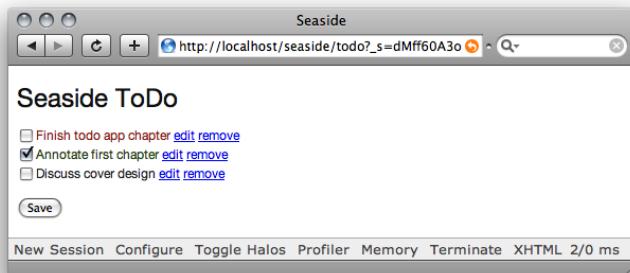


Figure 15.8: Our todo application with checkboxes and save buttons.

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' ]
```

15.6 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: ''
    poolDictionaries: ''
```

```
category: '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 Part III.

```
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;
}'
```

15.7 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`.

Advanced

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 Section 11.5.

```
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
```

Advanced

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 (that we present in Chapter 26) 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.

15.8 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.

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 Figure 15.9.

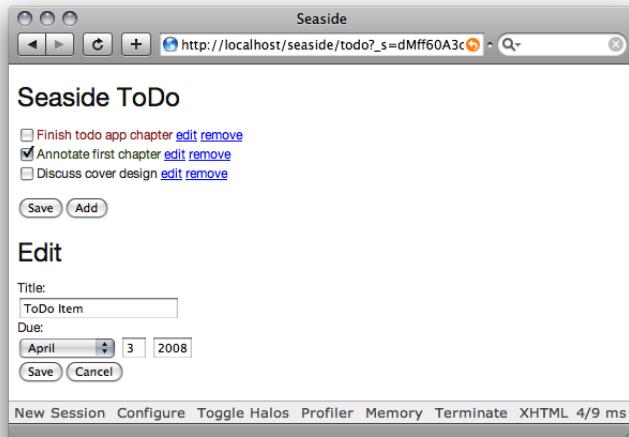


Figure 15.9: Getting an editor to edit new item.

We begin by adding an instance variable named `editor` to the `ToDoListView` class as follows:

```
WACOMPONENT subclass: #ToDoListView
  instanceVariableNames: 'editor'
  classVariableNames: ''
  poolDictionaries: ''
  category: '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
```

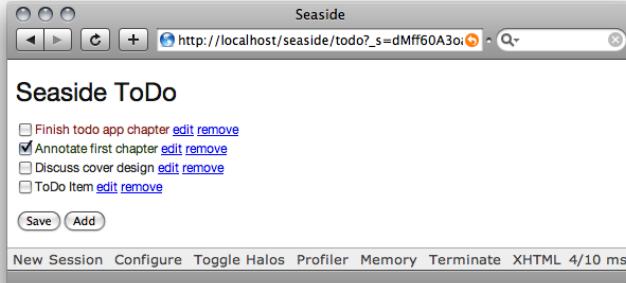


Figure 15.10: With an item added.

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 attaching a handler from the parent once the child component is instantiated. 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 ]
```

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 Part V).

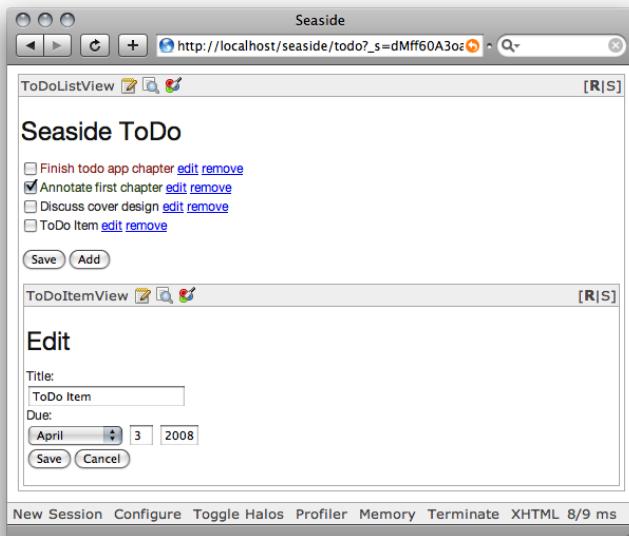


Figure 15.11: TodoFinal.

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 Figure 15.11.

15.9 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 16

A Web Sudoku Player

In this chapter we will build a Sudoku web application as shown in Figure 16.1. 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 16.1: Just started playing.

16.1 Sudoku 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.

16.2 Sudoku Component

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: ''
  poolDictionaries: ''
  category: '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;
}'
```

16.3 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 16.2: The column labels look like this.

If you run the application again, you should see the column labels as shown in Figure 16.2.

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 16.3.

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:..`

	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 16.3: Row labels are letters, column labels are numbers.

```
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 Figure 16.4.

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 possi-

	1	2	3	4	5	6	7	8	9
A									
B									
C									
D									
E									
F									
G									
H									
I									

Figure 16.4: The Sudoku board with the style sheet applied.

bility 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 numberOfRows = 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 Figure 16.5.

	1	2	3	4	5	6	7	8	9
A	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
B	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
C	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
D	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
E	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
F	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
G	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
H	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
I	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789

Figure 16.5: The Sudoku grid is showing the possible values for each cell.

16.4 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: [
```

```

htmltextInput
  size: 2;
  callback: [ :value |
    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 Figure 16.6. As you enter new values, you will see the possibilities for cells automatically be automatically reduced.

	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 16.6: A partially filled Sudoku grid.

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 16.7 shows the result of clicking on Solve.

	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 16.7: A solved Sudoku grid.

16.5 Back Button

Now let's play a bit. Suppose we have entered the values 1 through 6 as shown in Figure 16.1 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>>setCurrentCell: 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 Figure 16.8.

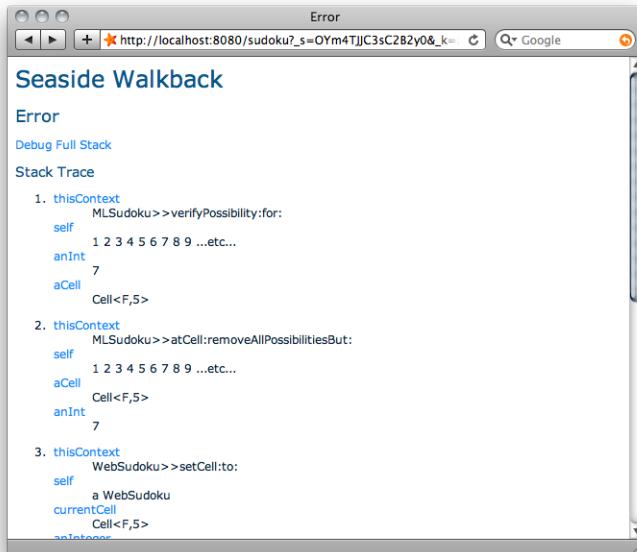


Figure 16.8: Error when trying to replace 6 by 7.

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
```

16.6 Summary

While the Sudoku solver introduces some subtleties because of its back-tracking 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 17

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 23.

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 23 we explain how to serve static files in a more efficient way using Apache.

17.1 Images

We illustrate the inclusion of static resources by displaying an external picture within an otherwise empty component. 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'
```



Figure 17.1: Including an external picture into your components.

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'
```

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`. 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.



Figure 17.2: Setting the Resource Base URL of your application.

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:`.

Another interesting way to serve a picture is to use a dynamically generated picture from within your image. In Pharo 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 `WAIImageTag>>document:` as follows:

```
html image document: EllipseMorph new
```

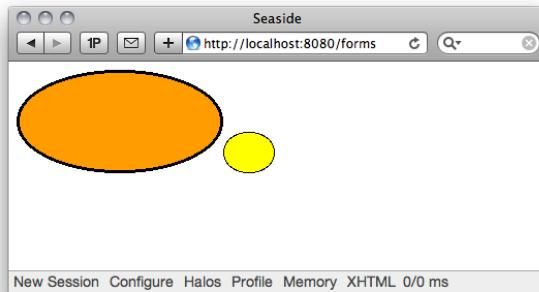


Figure 17.3: Displaying Pharo Morphs.

If you are using Pharo, have a look at the example implemented in the class `WAScreenshot`. It demonstrates a much more sophisticated use of `WAIImageTag>>form:` and presents the Pharo desktop as part of a web application. Furthermore it allows basic interactions with your windows from the web browser.

17.2 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: ''
poolDictionaries: ''
category: '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 Figure 17.4:

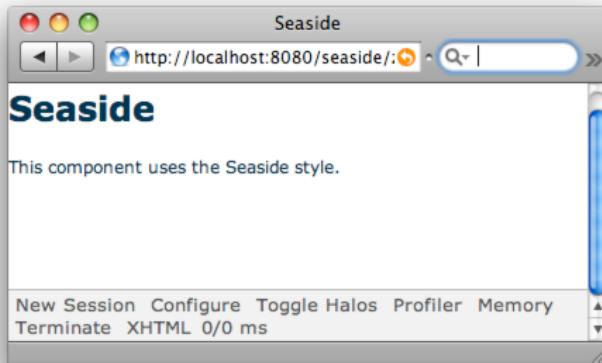


Figure 17.4: Application with enabled style sheet.

Now we will show how you can replace the stylesheet using the *FileLibrary*.

17.3 Working With File Libraries

Since version 2.7, Seaside has included 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 23 to read about Apache configuration for static file serving.

17.3.1 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 17.6. This will show which file libraries are available.

Click the configure link for `MyFileLibrary` as shown in Figure 17.6 right.

There you can add a file by uploading it (select the file, then click the *Add* button as shown by Figure 17.7).

Important

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 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,

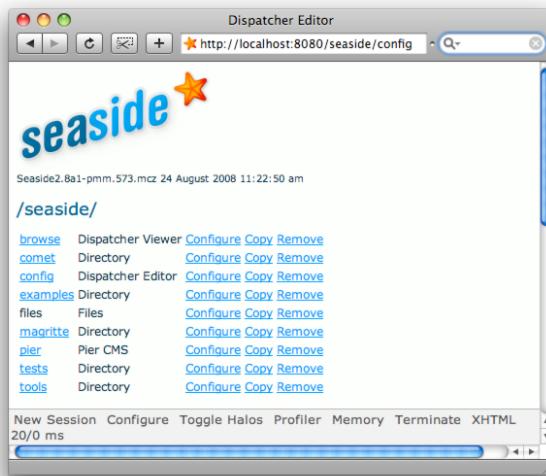


Figure 17.5: Configuring file libraries through the web interface: clicking on files - configure.

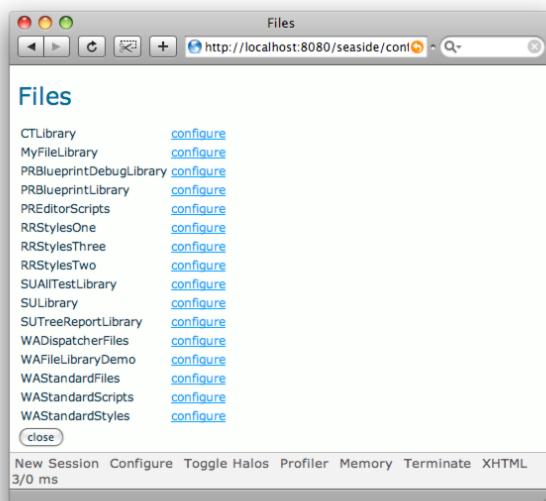


Figure 17.6: File libraries.



Figure 17.7: Adding file to MyLibrary.

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.

17.3.2 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 `WAFfileLibrary 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.

17.4 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: ''
  poolDictionaries: ''
  category: 'Test'
```

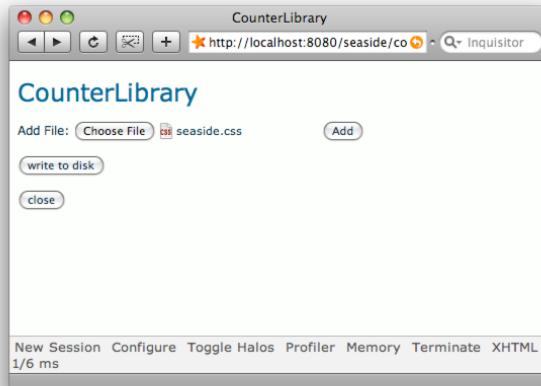


Figure 17.8: An empty CounterLibrary.

We will follow the steps presented in the previous section and associate two resources to our library. One is an icon named `seaside.png` and the other is a

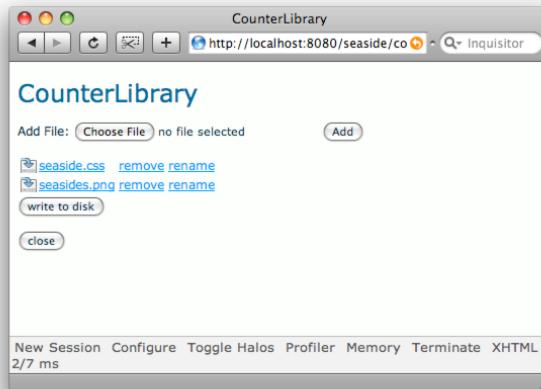


Figure 17.9: Adding files to the CounterLibrary.

CSS file named `seaside.css` – you can download the ones from the Seaside website we mentioned before:

seaside.png

`http://www.seaside.st/styles/logo-plain.png` (rename once downloaded).

seaside.css

`http://seaside.st/styles/main.css`

Important

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 Figure 17.10.



1

++ --

Figure 17.10: Counter with the `updateRoot:` method defined.

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)">
    
    <h1>0</h1>
    <a href="http://localhost:8080/WebCounter?_s=UwGcN6vwGVmj9icD&_k=D6Daqxer
        &1">++</a>&nbsp;
    <a href="http://localhost:8080/WebCounter?_s=UwGcN6vwGVmj9icD&_k=D6Daqxer
        &2">--</a>
...
...
```

17.5 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 Chapter 17.
- Otherwise read Section 23.3 about Apache file serving and configuration.

17.6 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.

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).

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.

17.6.1 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.

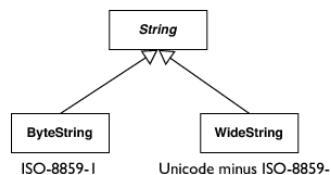


Figure 17.11: The Pharo String Hierarchy.

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 `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). Such classes are about character sets and not encodings.

17.6.2 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.

17.6.3 In Seaside and Pharo

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 `ByteString`.

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 Squeak 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.

Important

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 Squeak). 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 18

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.

18.1 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: (WAIInspector 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.

18.2 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
```

18.3 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 13 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: ''
  poolDictionaries: ''
  category: '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 Part V); and its general configuration such as its root component (see Chapter 23).

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 Figure 18.1. Now *Save* your changes.

General			
Deployment Mode	false	override	inherited from WAGlobalConfiguration
Error Handler	WAWalkbackErrorHandler	override	inherited from WARenderLoopConfiguration
Main Class	WARenderLoopMain	override	inherited from WARenderLoopConfiguration
Redirect Continuation Class	WARedirectContinuation	override	inherited from WASessionConfiguration
Redirect Handler	WARedirectHandler	override	inherited from WASessionConfiguration
Render Continuation Class	WARenderContinuation	override	inherited from WASessionConfiguration
Root Component	Mininn	clear	
Session Class	InnSession	revert to: WASession overridden from WASessionConfiguration	
Session Expiry Seconds	600	override	inherited from WASessionConfiguration
Use Session Cookie	false	override	inherited from WASessionConfiguration

Figure 18.1: 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: [ 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 ]

```

Figure 18.2, Figure 18.3 and Figure 18.4 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.

18.4 Lifecycle of a Session

It is important to understand the lifecycle of a session to know which hooks to customize. Figure 18.5 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

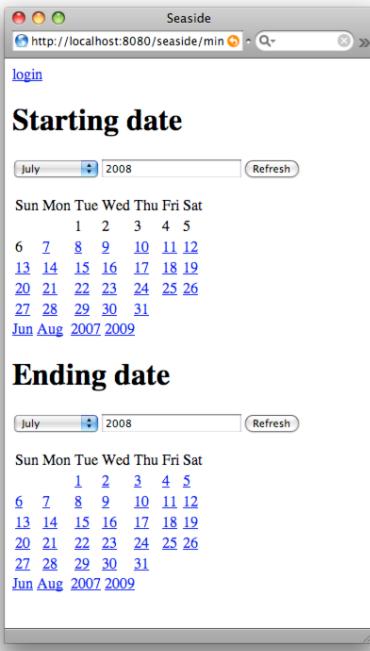


Figure 18.2: With Session.

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.
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.

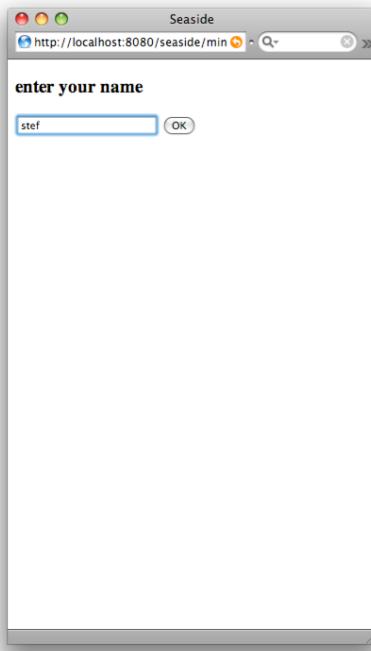


Figure 18.3: With Session: Enter your name.

18.5 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 (see Section 3.6.6), 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 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



Figure 18.4: With Session: Starting Date and Ending Date.

this is not really necessary, but to illustrate the functionality 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.

18.6 Recovering from Expired Sessions

The simplest way to change the default behavior of session expiry is to make your application bookmarkable. This involves serializing part of the

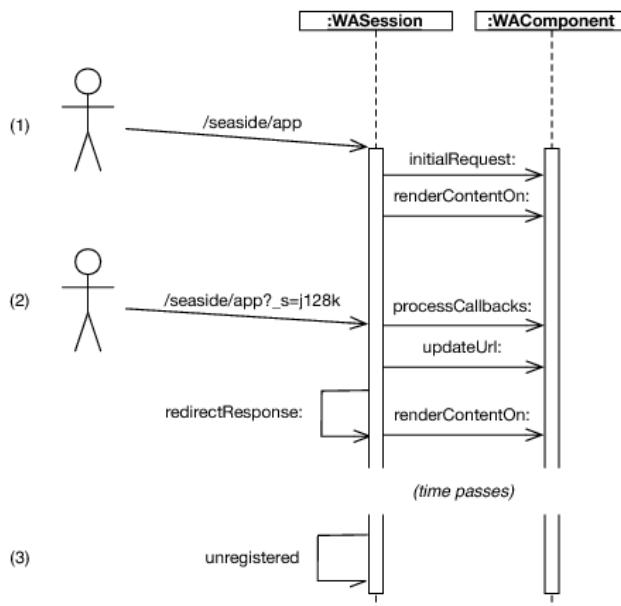


Figure 18.5: Lifetime of a session.

application state into a URL so that at any later point in time it can be retrieved, even if the session has expired. This is also a useful feature if you want that your application to be indexed by search engines or if you want to allow the possibility of bookmarking certain states of the application.

Normally as a Seaside application developer we don't worry about URLs. This is the only section of the whole book where we do, because we want to remember some of the application state. This is only because we want to be able to retrieve it later in case the session expired. Again we are using the MiniInn application as our running example.

Seaside provides the method `WAComponent>>updateUrl:` as a hook method that is called whenever the page is rendered. It allows one to modify the automatically generated URL that is displayed in the address bar of the web browser, so let's override it in our class:

```
MiniInn>>updateUrl: anUrl
  super updateUrl: anUrl.
  startDate isNil
    ifFalse: [ anUrl addField: 'startDate' value: startDate ].
  endDate isNil
    ifFalse: [ anUrl addField: 'endDate' value: endDate ]
```

In the above example we add both the value of `startDate` and `endDate` as a parameter to the URL. Have a look at the methods in `WAUrl` to see other

possibilities on how to modify the URL differently:

- `WAUrl>>addField:value`: Append the value with the key to the list of parameters. The key should be a string and not start with an underscore, such keys are reserved for internal matters by Seaside. The value will be converted to a string.
- `WAUrl>>addToPath`: Append the argument as a new path element. If the argument contains slashes the string is split into multiple elements.
- `WAUrl>>fragment`: Set the fragment part of the URL. This is the part at the very end of the URL separated by #.

Note that we could also have added the currently authenticated user to the URL. Essentially anything that can be meaningfully transformed to a string can be appended. It is the responsibility of the developer though to decide what application state is meaningful. As URLs are strings accessible to your users, you should expect that they might try to manipulate the URL manually. Thus, it is better to avoid putting any security related information in there. Also try to avoid putting too much information into the URL, as some web browsers and servers have problems with URLs that are more than 2048 characters.

When running the modified application there is not much difference. If you select a date it should appear as an URL parameter and stay as long as you don't change it again.

How can we now benefit from this additional information in the URL? Well, if the session expires we can have a look at the request parameters and we might find some information there that we can restore. To do this Seaside provides another hook method called `WAComponent>>initialRequest`: as presented earlier in this Chapter.

```
MiniInn>>initialRequest: aRequest
    super initialRequest: aRequest.
    aRequest fields
        at: 'startDate'
        ifPresent: [ :value | startDate := value asDate ].
    aRequest fields
        at: 'endDate'
        ifPresent: [ :value | endDate := value asDate ]
```

When a new session is started, Seaside calls the method `initialRequest:` on all initially visible components. Other than that, the method is never called. This allows us to have a look at the `WARequest` object and check if any of our URL parameters are present. If so, we convert the strings to a date and assign it in our model. We successfully restored part of our application state.

To test we need a way to flush all Seaside sessions. First start a session, login and select a start and end date. Then use the following expression to expire

all active sessions in your image.

```
WAAdmin clearSessions
```

When clicking on any link or simply pressing the refresh button, you will notice that the authenticated user was dropped. However, the start and end date is still persistent and you can interact with your application from within a new session.

More sophisticated examples of the interplay between `updateUrl:` and `initialRequest:` are included with your Seaside distribution. Browse for implementors of these two messages.

18.7 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 automatically 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.

18.8 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.

Part V

Web 2.0

Web 2.0 is a buzzword that characterizes the trend to make web applications more accessible. It is about opening up your data and services to your customers. It is about supporting standard protocols and allowing your application to run on many different devices, not just a web browser on a desktop computer. It is about letting your users interact with each other. It is about letting your application evolve with the needs of your users. That's why many Web 2.0 projects are constantly evolving.

In this chapter we will present the different solutions offered by Seaside to support Web 2.0. We will extend the todo application we built in Chapter 15 with Web 2.0 technology. The definition of Web 2.0 itself is still open. Every day there are new features popping up, calling themselves the new leader in the Web 2.0 movement. Seaside is very supportive of the Web 2.0 philosophy.

- Seaside ensures that the XHTML it produces is valid. It encourages developers to use meaningful XHTML markup and frees them from worrying about the final look of the application. Seaside lets designers define the colors, fonts and layout of the page through *Cascading Style Sheets* (CSS) as shown in Chapter 8.
- Seaside embraces the evolution of web applications. In contrast to file-based web frameworks, Seaside allows one to take advantage of the full power of the Smalltalk development environment. The ability to refactor, test and navigate an application with the same set of tools makes it extremely powerful and easy to change and adapt the application to new needs. On-the-fly debugging is a huge time saver.
- Seaside integrates easily with other output formats. Microformats are the easiest way to go. Just add specific classes and attributes to your XHTML to make your application generate a valid microformat. In Chapter 19 we demonstrate how to add an RSS (Really Simple Syndication) stream of todo items to your users.
- While the core of Seaside does not rely on JavaScript, this client side technology might help you to increase the appeal and the usability of your web application. AJAX (Asynchronous JavaScript and XML) enables you to build user interactions without having to reload the whole page with every click. Best of all, the application will feel much more responsive. In Chapter 20 and Chapter 21 we will learn how to extend the todo application by communicating asynchronously with the server and adding JavaScript gimmicks without writing a single line of JavaScript code.
- Comet takes AJAX to the next level. While traditional web servers always wait for the client to request data, Comet allows you to actively push changes from the server to the client. In Chapter 22 we will use this technology to let other people observe how our todo list updates.

Chapter 19

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 done using a dedicated tool called *feed reader* or *aggregator*, but most web browsers integrate this capability as part of their core functionality. If a web site offers an RSS feed, this is depicted with an icon like the one in below.



Figure 19.1: The Really Simple Syndication icon.

The RSS XML format is very much like XHTML, but much simpler. As standardised in the RSS 2.0 Specification, 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. In Figure 19.2 we 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>
```

19.1 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. 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: ''
poolDictionaries: ''
category: '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.

Note

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>
```

19.2 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.

19.3 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.

19.4 Subscribe to the Feed

Now we have done all that is required to let users subscribe. Below you can see how the feed is presented to the user in the feed reader when the URL was added manually.

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.

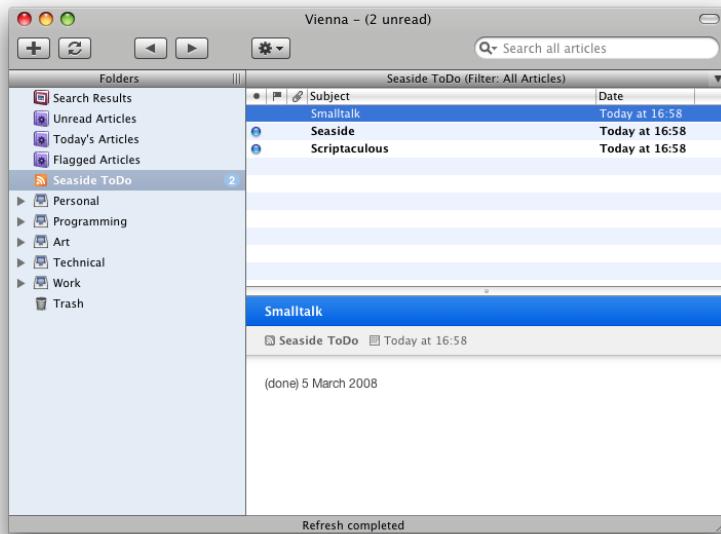


Figure 19.2: The ToDo Feed subscribed.

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'
```

19.5 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 20

Dynamic Content with Scriptaculous

While a simple web application requires communication with the server for each update of the display, refresh or any action, JavaScript-enabled applications can allow some part of the computation to be done in the client without requiring the server to recreate and resend the complete XHTML defining the page. This allows you, for example, to have UI updates without forcing the user to explicitly click on a link or press a button.

JavaScript running in the web browser can also communicate with the web server without the need to reload the whole page. This provides you with a lightweight way to provide updates to the contents of your application's pages, such as stock tickers.

The use of these techniques allows you to build highly dynamic and interactive web applications that behave like desktop applications rather than traditional web pages. Google's web mail client is a great example of how well this approach can work.

In this chapter, we give a brief description of the JavaScript frameworks Seaside supports. Then we explain how you can add JavaScript effects to your applications and show how you can take advantage of AJAX to support the communication between the client side and the server.

20.1 Prototype and script.aculo.us

Prototype (<http://www.prototypejs.org>) is a free, open-source JavaScript framework, that aims to ease JavaScript programming. The framework was created by Sam Stephenson and enhances the JavaScript experience with utilities to perform AJAX requests and to do DOM manipulations without having to worry about web browser incompatibilities. Although the framework is called Prototype (a reference to JavaScript's prototype-based inheritance model), it implements its features in a traditional class-based model.

Many of Prototype's extensions to JavaScript are inspired by Smalltalk and Ruby. For example, after loading the Prototype library, a JavaScript Array is extended with methods that have familiar Smalltalk names: `select`, `reject`, `collect`, and `detect`. Up until recently, these similarities make Prototype the framework of choice to integrate with Seaside and Smalltalk. Nowadays the JQuery (<http://www.jquery.com/>) framework described in Chapter 21 is a nice alternative to the Prototype library.

The Prototype framework aims to make JavaScript programming simpler, and it does not have any features to make the user interface richer. `script.aculo.us` (<http://script.aculo.us>) is a free, open-source JavaScript framework, built on top of Prototype, providing visual effects, drag and drop and several ready-made user interface controls. The author of `script.aculo.us` Thomas Fuchs summarizes: "It's about the user interface, baby!"

Seaside provides a complete integration of Prototype and `script.aculo.us` called "Scriptaculous". This means that you can access all aspects of these frameworks from Smalltalk by writing Smalltalk code only. Every JavaScript class has a counterpart in the Smalltalk world and can be used without having to know the details of the underlying JavaScript implementation. The key feature is that Scriptaculous lets you write Smalltalk code that will generate JavaScript snippets embedded in the XHTML stream created by Seaside.

20.1.1 Installation

As a first step we need to make sure that the Smalltalk Scriptaculous package is loaded. Most prebuilt images such as the one-click image already come with this package included. Note that despite the Scriptaculous name, the package includes the JavaScript source and integration code for both the Prototype and the `script.aculo.us` frameworks.

Make sure to have the packages `Javascript-Core`, `Prototype-Core`, `Scriptaculous-Core` and `Scriptaculous-Components` loaded. For examples



Figure 20.1: Scriptaculous Demo and Functional Test Suite.

and functional tests also load the test packages `Javascript-Tests-Core`, `Scriptaculous-Tests-Core` and `Scriptaculous-Tests-Components`.

20.1.2 Adding the Library

The principle behind using JavaScript is that your application (and the XHTML served by the server) will contain some JavaScript invocations to JavaScript libraries. Therefore you have to mention to your application that it has to include the associate JavaScript libraries.

Seaside 3.0 has reorganised and modularised the Javascript packages, making it easier to load only those parts you need. Furthermore Seaside 3.0 gives you the possibility to chose between a full version for development, a minimized and compressed version for deployment, and a minimized and compressed version served through the Google AJAX Libraries API high-performance servers. In any case the end result should be the same.

Before being able to use any of the functionality provided by the Scriptaculous package, you need add the classes `PTDeploymentLibrary` and `SUDevelopmentLibrary` to our application. These are the normal file libraries that automatically includes the necessary JavaScript sources into the XHTML head of our application. There are two alternative ways of adding the Javascript libraries:

- Add the library using the application configuration interface by selecting `PTDeploymentLibrary` and `SUDevelopmentLibrary` from the list and clicking on `Add`.
- Add the library with the Seaside API, preferably in the `initialize` method on the class side of the root component. Don't forget to evaluate the method after adding or changing it.

```
ToDoListView class>>initialize
  (WAAdmin register: self asApplicationAt: 'todo')
    addLibrary: PTDeploymentLibrary;
    addLibrary: SUDeploymentLibrary
```

Important

If you fail to specify `PTDeploymentLibrary` and `SUDeploymentLibrary` in your application, you will not get a Seaside error message, but you will get a Javascript error message that is sometimes hard to diagnose, depending on the web browser you are using.

20.2 Snippets and Brushes

Adding JavaScript code to your Seaside application is not much different from rendering plain XHTML. You need to

1. ask the rendering canvas `html prototype` to instantiate a Prototype or `script.aculo.us` brush,
2. configure the newly created brush with a cascade of configuration messages, and
3. add the brush to the XHTML output. This will embed the JavaScript snippet at the desired place into the XHTML output stream.

In the following paragraphs we are going to have a in-depth look at these 3 steps and all the possibilities that Scriptaculous is providing. To directly dive into a running example, skip this section and continue with Section 20.3. To learn about the details you can always come back later.

20.2.1 Instantiate a Brush

The Prototype and Scriptaculous package extends the Seaside class `WARenderCanvas` with the methods `prototype` and `scriptaculous` that both return a factory object for JavaScript brushes. These factories are instances

of PTFactory. These brushes are responsible for creating well defined snippets of JavaScript code. A full listing of the available brushes is presented below.

JavaScript Class	Factory Selector	Smalltalk Class
\$ (Element)	element	PTElement
\$ (Form)	form	PTForm
\$ (Form.Element)	formElement	PTFormElement
\$\$ (Selector)	selector	PTSelector
Ajax.Autocompleter	autocompleter	SUAutocompleter
Ajax.InPlaceCollectionEditor	inPlaceCollectionEditor	SUIInPlaceCollectionEditor
Ajax.InPlaceEditor	inPlaceEditor	SUIInPlaceEditor
Ajax.PeriodicalUpdater	periodical	PTPeriodical
Ajax.Responders	responders	PTResponders
Ajax.Request	evaluator	PTEvaluator
Ajax.Request	request	PTRequest
Ajax.Updater	updater	PTUpdater
Control.Slider	slider	SUSlider
Draggable	draggable	SUDraggable
Droppables	droppable	SUDroppable
Effec	effect	SUEffect
Event	event	PTEvent
Insertion	insertion	PTInsertion
Sortable	sortable	SUSortable
Sound	sound	SUSound

For some Prototype and script.aculo.us functionality there are several aliases available. Seaside always tries to generate the shortest possible variation, however for reference it is often useful to know the long form as well. For example writing `$('foo').toggle()` is a synonym for the slightly longer version `Element.toggle('foo')`.

20.2.2 Using a Brush

Unlike a normal XHTML brush, we have to explicitly add the JavaScript snippet to the XHTML output stream. There are essentially four different places a JavaScript snippet can be added: (1) right at the current position in the XHTML output stream into a `script` tag, (2) into a method that will be automatically evaluated when the page has completely loaded, (3) to a DOM element, or (4) as an event handler to a DOM element.

1. Adding to a script tag. You can add the brush right at the current place into the XHTML output stream. This is done by creating a `script` tag using an expression such as `html script: aJavaScriptBrush`. This technique is simple and straightforward, however it has the disadvantage that the JavaScript code will be evaluated right away when the web browser parses the file. This might happen even before the whole page is read in the web browser and might cause JavaScript errors, as there are no guarantees that the DOM elements you use are already available. In most cases this is not the preferred way to go.

2. Adding to a list of load-scripts. You can add the brush to a list of load scripts. These load scripts will be evaluated after the page has finished loading. This solution is usually preferred over the previous one. The following expression is an example:

```
html document addLoadScript: aJavaScriptBrush
```

3. Adding to a DOM element. Often JavaScript brushes need to know the ID of the element they are supposed to operate on. For example if we want to apply an effect to an element we need to pass the ID of that particular element to the JavaScript brush. Luckily there is an easy way to do this automatically and at the same time add the JavaScript brush to the list of load scripts. Every XHTML brush understands the message `script:` which will (1) ensure that tag has an unique ID, (2) pass the ID of the tag to the JavaScript brush, and (3) add the JavaScript brush to the list of load scripts. Here's an example:

```
html div script: aJavaScriptBrush; with: 'Hello World'
```

4. Adding as a DOM event handler. So far the JavaScript code is executed unconditionally when it is encountered or after the page has loaded. Luckily one can assign JavaScript snippets to events on XHTML DOM nodes. Similar to the technique above, a JavaScript snippet that has no specific ID will operate on its owning XHTML element.

The most common events are `onClick` and `onchange`. For example,

```
html div onClick: aJavaScriptBrush; with: 'Hello World'
```

will execute the JavaScript code when the div element is clicked. You can find a full list of DOM events in the table below. Keep in mind that not all events are supported by all XHTML tags. For example the `onchange` event will never be triggered on a div tag, since it only applies to form elements that can be changed. The support and handling of some of the events (foremost among these are `onblur`, `ondblclick`, `onfocus`, `onload`, and `onunload`) differ significantly among the web browser implementations.

DOM Event	Seaside Selector	Description
<code>onblur</code>	<code>onBlur:</code>	When the element that is in focus, loses the focus.
<code>onchange</code>	<code>onChange:</code>	When a select input element has a selection made or when a text input element has a change in the text.
<code>onclick</code>	<code>onClick:</code>	When the mouse button is clicked over an element.
<code>ondblclick</code>	<code>onDoubleClick:</code>	When the mouse button is double clicked over an element.
<code>onfocus</code>	<code>onFocus:</code>	When an element receives focus either by the mouse or by tabbing navigation.
<code>onkeydown</code>	<code>onKeyDown:</code>	When a key is pressed down over an element.
<code>onkeypress</code>	<code>onKeyPress:</code>	When a key is pressed and released over an element.
<code>onkeyup</code>	<code>onKeyUp:</code>	When a key is released over an element.
<code>onload</code>	<code>onLoad:</code>	When the user agent finishes loading a window.
<code>onmousedown</code>	<code>onMouseDown:</code>	When the mouse button is pressed over an element.
<code>onmousemove</code>	<code>onMouseMove:</code>	When the mouse is moved while it is over an element.
<code>onmouseout</code>	<code>onMouseOut:</code>	When the mouse is moved away from an element.
<code>onmouseover</code>	<code>onMouseOver:</code>	When the mouse is moved onto an element.
<code>onmouseup</code>	<code>onMouseUp:</code>	When the mouse button is released over an element.
<code>onreset</code>	<code>onReset:</code>	When a form is reset.
<code>onselect</code>	<code>onSelect:</code>	When a user selects some text in a text field.
<code>onsubmit</code>	<code>onSubmit:</code>	When a form is submitted.
<code>onunload</code>	<code>onUnload:</code>	When the user agent removes a document from a window.

20.2.3 Configure a Brush

There is no point in listing all possible configuration messages that can be sent to brushes. However, both underlying JavaScript frameworks have excellent documentation that can be directly mapped to the Smalltalk world. Use the table above to find out the JavaScript class name of the item in question and look it up on either <http://www.prototypejs.org/api> or <http://wiki.github.com/madrobby/scriptaculous/>. Some of the documentation available on these sites is also part of the class and method comments in your Smalltalk image.

20.3 Adding an Effect

Now we will demonstrate some applications of the principles just mentioned. It is straightforward, for example to add an effect to the heading of the ToDo application. To highlight the title when clicked, let us modify the method `renderContentOn:` as follows.

```
ToDoListView>>renderContentOn: html
    html heading
        onClick: html scriptaculous effect highlight; " <-- added "
        with: self model title.
    html form: [
        html unorderedList
            id: 'items';
            with: [ self renderItemsOn: html ].
        html submitButton
            text: 'Save'.
        html submitButton
            callback: [ self add ];
            text: 'Add' ].
    html render: editor
```

Before we do something more sophisticated, let's experiment a bit with this code. The first thing to try is to look at the generated source code (by using *Toggle halos* and pressing the *source* link). Obviously Seaside automatically transformed the JavaScript snippet that we specified and assigned it to the heading tag:

```
<h1 onclick="new Effect.Highlight(this)">
  Seaside ToDo
</h1>
```

Nice! Let's experiment a bit with this effect. For example we can change the default yellow highlight color to a flashing blue:

```

ToDoListView>>renderContentOn: html
    html heading
        onClick: (html scriptaculous effect
            highlight;
            startColor: Color blue); " <- added "
        with: self model title.
    html form: [
        html unorderedList
            id: 'items';
            with: [ self renderItemsOn: html ].
        html submitButton
            text: 'Save'.
        html submitButton
            callback: [ self add ];
            text: 'Add' ].
    html render: editor

```

In this case Seaside generates the following JavaScript expression:

```

<h1 onclick="new Effect.Highlight(this, {'startcolor': '#0000FF'})">
    Seaside ToDo
</div>

```

You might also want to try some other effects such as `SUEffect>>pulsate` or `SUEffect>>switchOff`. These are fun to play with, but not particularly useful for our example. Let's do something slightly more useful and display some help text or a copyright notice when the heading is clicked. You can see the result in Figure 20.2. Note that we extracted the rendering of the title and help text into a separate method and that we improved the style-sheet as well.

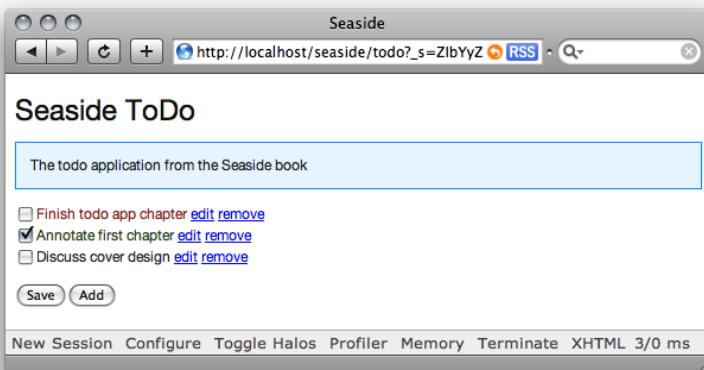


Figure 20.2: The todo application with help text faded in.

```
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
```

```
ToDoListView>>renderHeadingOn: html
| helpId |
helpId := html nextId.
html heading
    class: 'helplink';
    onClick: (html scriptaculous effect
        id: helpId;
        toggleAppear);
    with: self model title.
html div
    id: helpId;
    class: 'help';
    style: 'display: none';
    with: 'The todo application from the Seaside book'
```

```
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;
}'
```

This code requires some explanation:

- First we extract the heading code from `renderContentOn:` into its own method `renderHeadingOn::`. This makes it easier for us to change its behavior.
- In `renderHeadingOn:` we first ask Seaside to generate a new unique ID. In this case it would be possible to use our own ID string, however this is considered bad practice. If you want to use the component in a different context, the ID might conflict with existing code. Also when hardcoding IDs it is, for exactly the same reason, not possible to have two instances of the same component visible on a page. We also add some class information to the heading tag, to allow our CSS to cause the cursor to change when the mouse moves over the heading.
- Next we create an effect called `SUEffect>>toggleAppear` and assign it to the `onclick` event of the heading. Since we don't want to toggle the appearance of the heading itself (which would be the default), we pass it the generated ID. Two other interesting toggle-effects we could have used are `SUEffect>>toggleBlind` and `SUEffect>>toggleSlide`.
- Last but not least we add the div element we would like to toggle on and off. Obviously this is the element with our automatically generated ID, so we assign it here. Since the div element should not be visible in the beginning we hide it with an inline style. Moreover we assign the CSS class `help`, so that we are able to change its look from the style-sheet.

If we have a look at the running ToDo application, it works as proposed. However there is one thing we might want to improve: whenever we have a full refresh, for example when clicking on a link or button, the help text disappears again. The reason is that we only specified client side behavior: whenever the heading is clicked the JavaScript code is executed that has been generated by Seaside. And the JavaScript code doesn't talk back to the server, it just toggles the visibility of a DOM node on the client. Luckily there is AJAX that enables us to talk back to the server, and this is exactly what we are going to do in the next section.

20.4 AJAX: Talking back to the Server

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`. Thankfully the Prototype framework and its integration into Seaside makes it a cakewalk to use in your applications.

20.4.1 Defining a Callback

Our goal is to get notified whenever the heading is clicked. We add `html request callback: aBlock` to the click handler, where `aBlock` is a Seaside callback block, like the one we use for an anchor. We add the snippet as another click handler to the heading and add an instance variable to remember the visibility of our help text. This time we only hide the div tag with an inline style, if the element should be invisible.

Add the `visible` instance variable to the `ToDoListView` class, and then add the following:

```
ToDoListView>>initialize
super initialize.
visible := false
```

```
ToDoListView>>renderHeadingOn: html
| helpId |
helpId := html nextId.
html heading
    class: 'helplink';
    onClick: (html scriptaculous effect
        id: helpId;
        toggleAppear);
    onClick: (html scriptaculous request  "<-- added "
        callback: [ visible := visible not ]);
with: self model title.
html div
    id: helpId;
    class: 'help';
    style: (visible ifFalse: [ 'display: none' ]);  "<-- added "
with: 'The todo application from the Seaside book'
```

Don't forget to start a new session before playing with the new functionality, otherwise the instance variable won't be initialized. Have a look at the

generated source code. Seaside quietly combines the two AJAX snippets in the click handler of the tag:

```
Effect.toggle('id1', 'Appear');
new Ajax.Request('http://localhost:8080/todo',
{'parameters': ['_s=woCPiIxqSIGkDMqu', '_k=4FL61fCv', '2'].join('&'))}
```

As you can see, Seaside takes care of a lot of low level details for us. When we try the application it is actually quite hard to see that it works any differently than before; certainly for the end user it looks exactly the same as before. Behind the scenes it's a different story: in the background our additional JavaScript code triggers a request that goes to the server and evaluates our callback block. You can put a logging statement `Transcript show: visible; cr` or a `self halt` into that block to see that it really evaluates the code.

Another very useful tool to observe how AJAX requests are passed between your web browser and Seaside is Firebug, a Firefox extension. We will have a look at this tool in the next section in great detail.

20.4.2 Serializing a Form

An annoying behavior of our application is that the user has to click *save* to submit changes in the todo list. This is not really user-friendly and also dangerous, because it is too easy to forget to hit the save button after toggling the checkboxes. With AJAX we can improve this, and let the browser worry about saving our changes. Let's use AJAX to automatically submit the form whenever a checkbox is clicked:

```
ToDoListView>>renderItem: anItem on: html
    html listItem
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: [
            html checkbox
                onChange: (html scriptaculous request "!-- added "
                    triggerForm: (html scriptaculous element up: 'form'));
                value: anItem done;
                callback: [ :value | anItem done: value ].
            html text: anItem title.
            html space.
            html anchor
                callback: [ self edit: anItem ];
                with: 'edit'.
            html space.
            html anchor
                callback: [ self remove: anItem ];
                with: 'remove' ]
```

Once you've made this change, load the application in your browser and click on one of the checkboxes. Without pressing *save*, refresh your browser window, and you should see the new value has 'stuck'.

Again we are using the `request` object, but this time instead of a callback we tell it to trigger a form using `PTAjax>>triggerForm:`. This method expects a DOM ID or DOM node of a form as an argument. We could use the same trick that we used previously and let Seaside generate a unique ID and assign it to the form and to our JavaScript snippet. This technique has the disadvantage that we introduce a new dependency between the method that renders the form and the one that renders the checkbox, which might not be desired.

```
html scriptaculous element up: 'form'
```

walks up the DOM tree and returns the first form it encounters. In our case this is the surrounding form of our checkbox. The starting point of this lookup is the current DOM node (our checkbox) since we didn't specify a different element using `PTElement>>id:`. If we only used `PTElement>>up`, we would have got the parent element of the checkbox, which in this example is the `div` element. There are some other methods defined in `PTElement` that navigate the DOM tree in other directions: `PTElement>>down`, `PTElement>>next`, and `PTElement>>previous`.

Serializing your data. Once the form element has been found in the DOM, we then want to specify what to do with it:

```
html request
  triggerForm: (html element up: 'form')
```

Here, the `triggerForm:` says we want the page to 'serialize' the form, that is, to convert the DOM representation of the form (and all its included fields) into a string that can be passed back to the server when required. We pass this serialised version into the `request` method of the `html` object, which causes the required JavaScript code to be generated in the page in your browser.

Because all of this happens as an argument to `html checkbox onChange:`, the script will be generated into the `onchange` attribute of the checkbox, giving you something like the following:

```
<input class="checkbox" type="checkbox" name="4" checked="checked"
  onChange="new Ajax.Request(
    'http://localhost:8080/todo',
    {'parameters': ['3', $(this).up('form')).serialize()]).join('&')}")/>
```

So the effect of this code is that when you click on the checkbox, its `onChange` event gets fired. This creates an Ajax request that is sent back to your server with your serialized form. The Scriptaculous code in Seaside will then process

all the callbacks defined in your form, using the data it retrieved from the serialized form.

Obviously the *save* button is not needed in our example any more and we can remove it, but you could decide to keep the button so that your application would continue to work for people who have JavaScript disabled. Since we still have the button, our application can be used with and without JavaScript support.

20.4.3 Updating XHTML

The most important feature of AJAX is the ability to update parts of a page, without having the browser request and parse a whole new page from scratch. This is important since in most web applications only small parts of a page change with each user interaction.

If you look carefully at your todo list page in the browser, you'll notice that the colour of your todo items doesn't change when you mark them as completed. It would be nice if we could change the colour of a todo item depending on its state and give the user visual feedback. Let's make the following change to our code:

```
ToDoListView>>renderContentOn: html
    self renderHeadingOn: html.
    html form: [
        html unorderedList
            id: (listId := html nextId); "!-- ensure you have this"
            with: [ self renderItemsOn: html ].
        html submitButton
            text: 'Save'.
        html submitButton
            callback: [ self add ];
            text: 'Add' .
    html render: editor
```

```
ToDoListView>>renderItem: anItem on: html
    html listItem
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: [
            html checkbox
                onChange: (html scriptaculous updater "!-- added"
                    id: listId;
                    triggerForm: (html scriptaculous element up: 'form');
                    callback: [ :ajaxHtml | self renderItemsOn: ajaxHtml ]);
                value: anItem done;
                callback: [ :value | anItem done: value ].
            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 asked Seaside for a unique ID using `html nextId`. We store this ID in an instance-variable and assign it to the unordered list in `renderContentOn:` to be able to refer to it in `renderItem:on:` as a target of an AJAX update action.

In `renderItem:on:` we replaced the `requestor` method with an `updater`, which understands many of the same messages as a `requestor`, but can also *update* the page once the form's callbacks have been processed. Since we still want the server to process the changes to the form, we keep the line that triggers the form. To allow the update to happen, we need to give the updater two new pieces of information.

First, we pass the ID of the DOM element that we want it to update. When the update happens, that whole section of your page will be removed, and replaced by some new content that you must specify.

Second, in order to specify the new content, we create a callback block which will be triggered to render. Notice that we pass a new renderer `ajaxHtml` to the callback block; the block uses this renderer to render the list of items. Also notice that the checkbox has two callbacks now: one responsible for the state of the checkbox, the other one responsible to update the HTML.

Important

It is important that you use only the renderer passed as argument to the AJAX callback block. Do *not* to use the `html` canvas of the outer context. `html` is invalid at the time the AJAX callback is triggered since it already has been sent out to the web browser when the full page request was processed.

You may have a look at the classes `PTAjax` and its subclasses `PTUpdater` and `PTRequest`. You will notice that they have many common messages they understand.

If you play with the application you will see that the state of the checkboxes is submitted now. You can observe that the colors of the individual items change as you click the checkboxes, this is because the updater re-renders the listing with the changed items.

20.4.4 Behind the curtains

Now it is time to explain a bit the logic behind AJAX. Below you see a sequence diagram of the interaction between the web browser and the server during an AJAX updater action.

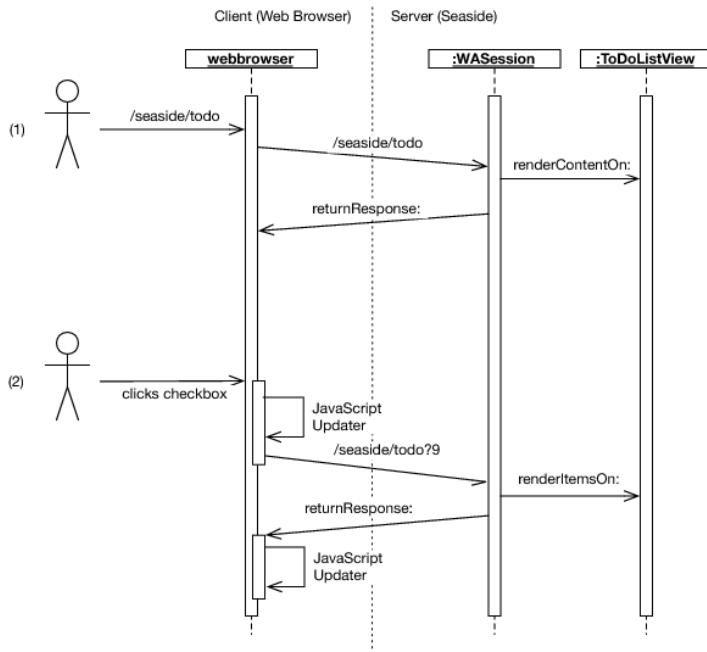


Figure 20.3: A full request (1) and an AJAX request (2) updating part of the page.

1. Full Request. Every web application, even a completely AJAX driven one, starts out with a full request. The web browser sends off a request to the desired URL, in our case `/todo`. The request is processed by the web server and passed to the Seaside dispatcher, which passes it on to the correct session and triggers the components to render. Eventually Seaside calls `returnResponse:` that returns the fully rendered page back to the web browser. As soon as the web browser receives the response it starts to parse the XHTML. During the parsing the browser may load some additional resources such as images, style-sheets or JavaScript files, and eventually displays the result to the user.

2. AJAX Request. What happens when the user triggers an AJAX updater?

1. In case of an AJAX update action it is the JavaScript engine of the web

browser that sets off the request. It is important to note that this request is made asynchronously (note the different arrow in the diagram), so the web browser remains fully functional while the AJAX request is processed by the server. Also note, that this means the JavaScript statement following the AJAX updater is immediately executed. The JavaScript engine does not block until the server returns a response.

2. Similar to the full request, Seaside receives the request and determines the correct session to handle it. Instead of calling `renderContentOn:` of the root page to generate a complete XHTML page, Seaside evaluates the callback block. In our case the callback block sends the message `renderItemsOn:,` that renders the list of todo items. The partial response is then passed back to the web browser using `returnResponse:..`
3. Finally the JavaScript engine gets notified that a response is ready to be processed (remember, the request was sent asynchronously). The XHTML snippet is parsed and inserted at a specific location in the DOM tree.

So far we only had a look at the AJAX *requestor* and *updater*. The requestor is used to serialize forms and trigger events on the server. The updater goes one step further. It allows one to update a specified DOM node with newly rendered content. There are two powerful AJAX features that we are going to look at now, the *periodical updater* and the *evaluator*.

Periodical Updater. The periodical updater `PTPeriodical` is an updater that periodically performs a normal AJAX update action. This is commonly used by all sorts of polling mechanisms. Note that in Chapter 22 we will have a look at a different way to continuously update contents on a web page, that doesn't use polling. We can add the following code snippet to any application to continuously display the current time:

```
html div
  script: (html scriptaculous periodical
    frequency: 1 second;
    callback: [ :ajaxHtml | ajaxHtml render: Time now ];
    with: Time now
```

Note that in this example we don't need to specify an ID, since we assign the script directly to a DOM element. In this case Seaside automatically connects the updater with the owning XHTML element. If we wanted to update a different DOM element, we would have to specify the ID of this element using `id:..` We set the update frequency to every second. This is more a period than a frequency, but that's what it is called in the Prototype framework. The callback block is then evaluated every second and renders the current time.

Evaluator. The evaluator `PTEvaluator` is the most complicated, but also most powerful AJAX mechanism. Instead of updating a specific DOM element,

it injects JavaScript code into the browser of the client. This is extremely powerful and allows one to update multiple DOM elements and play effects in one request. Again the evaluator looks very similar to the normal updater, but instead of passing a XHTML canvas into the callback block it gives you a script object where you insert JavaScript snippets. To see some sophisticated examples of its use have a look at `SUTabPanel` or `SUAccordion` that come with the Scriptaculous package. Both widgets update multiple parts of the page and change several CSS classes of the widget at the same time.

20.4.5 Wrap Up

Before using AJAX you have to make several decisions. First you must decide which AJAX strategy you want to use.

- `PTFactory>>request` – The requestor does not send anything back to the client. It solely sends a request and pushes data to the server.
- `PTFactory>>updater` – The updater updates a single part of the page. You need to provide the ID of the DOM element to update and a callback block that expects one parameter to render the partial XHTML on.
- `PTFactory>>periodical` – The periodical updater is an updater that is periodically executed. Additionally you need to specify an update frequency.
- `PTFactory>>evaluator` – The evaluator injects new JavaScript code into the web browser. Its callback block expects one parameter that will accept new JavaScript snippets.

After having chosen the AJAX strategy you might want to specify additional options and declare state that should be transmitted to the server. The most common ones are:

- `PTAjax>>triggerForm:` – Serialize a complete form and trigger all its associated callbacks. Note that the callbacks of submit buttons are ignored to preserve consistency, use the callback to trigger specific code evaluation.
- `PTAjax>>triggerFormElement:` – Serialize a form "element" such as a text input field and triggers its associated callback. Note that this does not work for all form elements. For example, check-boxes depend internally on other hidden form elements. Submit-button callbacks are ignored.
- `PTAjax>>callback:value:` – Serialize the result of evaluating the value as a JavaScript expression on the client and pass it as an argument into the callback block.

Furthermore you might want to register some events to get notified about the state of the AJAX action. The most common events are listed below.

- `PTAjax>>onSuccess:` – Invoked when a request completes and its status code is undefined or belongs to the 200 family.
- `PTAjax>>onFailure:` – Invoked when a request completes and its status code exists but is not in the 200 family.
- `PTAjax>>onComplete:` – Invoked at the very end of a request's life-cycle, once the request completed, status-specific callbacks were called, and possible automatic behaviors were processed.

The above events and some other AJAX options can also be globally set using `PTResponders`. This is useful to define an error handler once for the whole page in case the session expires. The following code snipped displays an error message and triggers a full refresh whenever that happens:

```
html document
addLoadScript: (html prototype responders
    onFailure: (html javascript alert: 'Session Expired') ,
                (html javascript refresh))
```

20.5 Drag and Drop

The `script.aculo.us` library comes with sophisticated support for drag and drop. You can define DOM elements to be draggable and (other) DOM elements to accept drags. `script.aculo.us` provides something called *sortables* which are very easy to use. They enable you to specify XHTML elements whose children can be sorted. As seen in the figure below, we are going to use sortables to enable end-users to reorder their todo items.

To implement sortable todo items we need to change two parts of the application. First we need to assign a *passenger* to every list-item. Think of every item in our todo list being represented by an XHTML list-item and that we have to tell Seaside how to make that mapping so that it can automatically track the order of the items:

```
ToDoListView>>renderItem: anItem on: html
    html listItem
        passenger: anItem; "<-- added"
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: [
            html checkbox
                onChange: (html scriptaculous updater
                    id: listId;
                    triggerForm: (html scriptaculous element up: 'form');
                    callback: [ :ajaxHtml | self renderItemsOn: ajaxHtml ]);
```

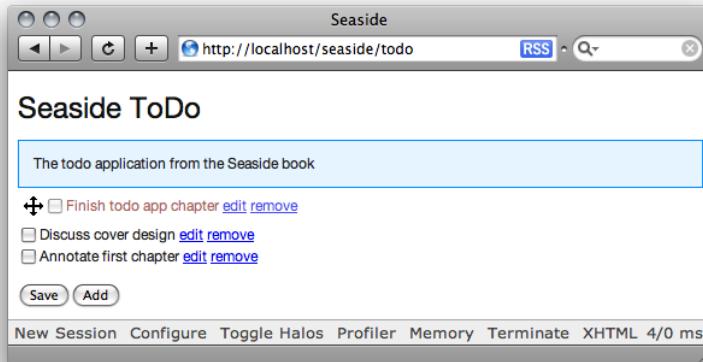


Figure 20.4: Reorder the items in the todo list.

```
value: anItem done;
callback: [ :value | anItem done: value ].
html text: anItem title ]
```

In our example this is the unordered list, that should be made sortable. We attach the sortable script to the list and assign an `SUSortable>>onUpdate:` handler that uses an AJAX request to serialize the changed order of list-items and triggers the associated Seaside callback.

```
ToDoListView>>renderContentOn: html
    self renderHeadingOn: html.
    html form: [
        html unorderedList
            id: (listId := html nextId);
            script: (html scriptaculous sortable "!-- added"
                onUpdate: (html scriptaculous request
                    triggerSortable: listId
                    callback: [ :items | self model items: items ]));
            with: [ self renderItemsOn: html ].
        html submitButton
            text: 'Save'.
        html submitButton
            callback: [ self add ];
            text: 'Add'.
        html render: editor
```

There are a number of other options available on the sortable.

- `SUSortable>>constraint:` – Set it to `#horizontal` or `#vertical` to constrain dragging to be in the horizontal or vertical direction only.

- `SUSortable>>ghosting` – If set to true, the dragged element will appear faded and a clone will stay at the old place, instead of directly dragging the original element.
- `SUSortable>>handle:` – Restrict the selection of child elements to those with the given CSS class.
- `SUSortable>>tag:` – Sets the kind of tag (of the child elements of the container) that will be made sortable. The default is `li` (as it is in our example), you have to provide the XHTML tag if you use something other than an ordered or unordered list.

For more advanced uses of drag and drop have a look at the examples that come bundled with Seaside. `SUSortableTest` demonstrates a single sortable list. `SUSortableDoubleTest` demonstrates two sortable lists side by side, where you can reorder the individual items as well as move items from one side to the other. `SUSortableDoubleTest` demonstrates a small shop, where you drag items into a cart, and from to cart to a trash bin. This example also shows how to combine drag and drop operations with graphical effects to give users feedback about their actions.

Note

Note that drag and drop operations are not yet that common in the context of web applications. End users might not discover them unless you give them an explicit indication that they can drag and drop parts of your user-interface, for instance you can try adding the following to your `style` method:

```
li { cursor: move; }
```

20.6 JavaScript Controls

script.aculo.us comes with a collection of JavaScript widgets bundled that can be used from Seaside. In this section we are going to add an in-place editor to the `ToDo` application that allows us to edit the title of the todo item right in the list with the checkboxes by simply clicking on the item, as seen below.

To get the in-place editor up and running we only have to change the method `renderItem:on:` by replacing

```
html text: anItem title.
```

with

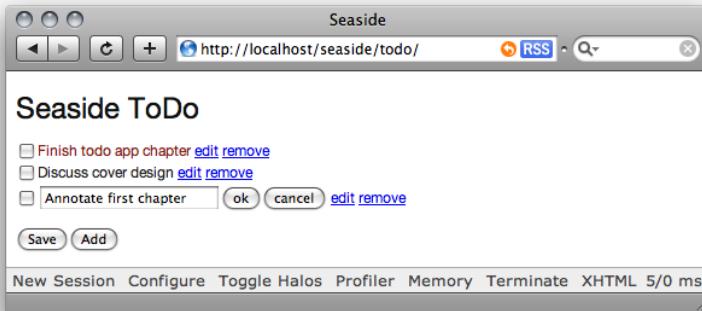


Figure 20.5: In-place item editor.

```
html span "!-- added"
script: (html scriptaculous inPlaceEditor
    cancelControl: #button;
    triggerInPlaceEditor: [ :value | anItem title: value ];
    callback: [ :htmlAjax | htmlAjax render: anItem title ];
    with: anItem title.
```

Here we add a XHTML span element to specify exactly what part we want to turn editable. Then we need to specify two callback blocks: the first one for `SUIInPlaceEditor>>triggerInPlaceEditor:` to store the edited value back into the todo item, and the second one for `SUIInPlaceEditor>>callback:` to update the changed item. Furthermore we specify in this example a `SUIInPlaceEditor>>cancelControl:` button to be used to cancel editing. We could also write `#link` or `false` here, to put a link or to disallow cancelling altogether.

Your complete method should now look like this:

```
renderItem: anItem on: html
    html listItem
        passenger: anItem;
        class: 'done' if: anItem isDone;
        class: 'overdue' if: anItem isOverdue;
        with: [
            html checkbox
                onChange: (html scriptaculous updater
                    id: listId;
                    triggerForm: (html scriptaculous element up: 'form');
                    callback: [ :ajaxHtml | self renderItemsOn: ajaxHtml ]);
                value: anItem done;
                callback: [ :value | anItem done: value ].
            html span "!-- added"
            script: (html scriptaculous inPlaceEditor
```

```

cancelControl: #button;
triggerInPlaceEditor: [ :value | anItem title: value ];
callback: [ :htmlAjax | htmlAjax render: anItem title ];
with: anItem title.
html space.
html anchor
callback: [ self edit: anItem ];
with: 'edit'.
html space.
html anchor
callback: [ self remove: anItem ];
with: 'remove' ]

```

In its current state, the in-place editor is fully functional. There is a small glitch in the visual presentation, because script.aculo.us temporarily introduces a `form` element into the DOM tree that causes two ugly line-breaks. We can fix that by adding the following style to our style-sheet:

```

li form {
  margin: 0;
  display: inline;
}

```

The in-place editor supports a wide variety of options and events. We are going to point out the most important ones. To see all possibilities, check out the class `SUIInPlaceEditor`.

- `SUIInPlaceEditor>>cancelText:` – The text of the button or link that cancels editing.
- `SUIInPlaceEditor>>highlightColor:` – The color to be used to highlight the editable area when the user hovers the mouse over it.
- `SUIInPlaceEditor>>okControl:` – `#button` or `#link`, if the ok command should be displayed as a button or a link. `false` if there should be neither and the editor can only be closed by pressing the enter key.
- `SUIInPlaceEditor>>okText:` – The text of the submit button that submits the changed value to the server.
- `SUIInPlaceEditor>>rows:` – The number of rows the input field should use, anything greater than 1 uses a multiline text area for input instead of a text-input.
- `SUIInPlaceEditor>>submitOnBlur:` – Set this to `true` to automatically submit the editor when it loses focus.

There are several other JavaScript controls available. Similar to the `SUIInPlaceEditor` we've seen above, there is an `SUIInPlaceCollectionEditor`. The in-place collection editor displays a drop down list, instead of a text input field, when the editing is triggered. You can see a combined example

when browsing `SUInPlaceEditorTest`. The `SUAutoCompleter` and its example `SUAutoCompleterTest` shows how to add autocompletion functionality to a text-input or text-area field. This allows users to start typing something while the JavaScript library will asynchronously ask the server for possible tokens that are then offered to the user within a drop-down list as possible completion matches. Another interesting control is `SUSlider` that is demoed in `SUSliderTest`. The slider offers a sophisticated scroll-bar implementation that is otherwise missing in XHTML.

20.7 Debugging AJAX

For now we have a fully functional todo application. However, before we continue to improve the application further, we would like to have an in-depth look at how to debug an AJAX application. The challenge here is that there are many different technologies being combined together.

On one side we have Smalltalk and Seaside, on the other hand you have the web browser and JavaScript. We assume that you are familiar with the tools available in Smalltalk to debug code, to set breakpoints, to inspect objects or to log output. These tools behave the same way when you are using AJAX. What should you do, when the error does not happen in Smalltalk but on the JavaScript side? What should you do if the browser shows an error message? How can you investigate the situation if the browser does not do what you expect?

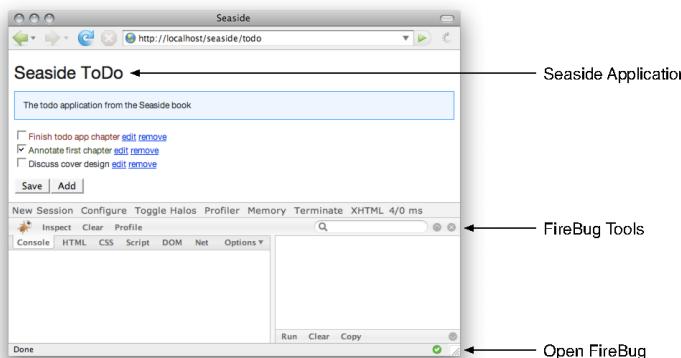


Figure 20.6: Firebug: Web development evolved.

There are tools available for most modern web browsers to help you to debug JavaScript code and to inspect XHTML DOM nodes. Here we list some of the most popular tools:

Firefox	Firebug
Opera	Dragonfly Introduction to dragonfly
WebKit (Safari, Chromium)	Web Inspector
	Drosera Debugger
Microsoft Internet Explorer	Developer Toolbar
	Script Debugger

In this section we are going to concentrate on using Firebug, an extremely powerful plugin for the Firefox browser.

Launching Firebug. While viewing the todo application, open the Firebug console. There are two ways to do this: by clicking on *Tools > Firebug > Open Firebug*; or by clicking on the Firebug icon on the status bar – this will be a green checkmark or a ‘bug’ icon depending on your version of Firebug. If you have a big screen you might want to detach Firebug from the main window, to have more space for your application.

Console. Click on the *Console* tab. This may prompt you to enable it before proceeding; if this is the case, you should also reload your page. Now click in the todo application on the checkboxes, and observe how the asynchronous requests are logged in the console. By clicking on the log entries you can further inspect the header fields, and the contents of the request and response objects. Note that this tool is indispensable for observing how AJAX requests are sent, since there is no other way to observe it on the client side.

Another interesting feature of the console is the ability to display custom information. For example you can temporarily replace the code that renders the checkbox with the following code:

```
html checkbox
onChange:
  (html logger info: 'Before AJAX'),
  (html updater
    id: listId;
    triggerForm: (html element up: 'form');
    callback: [ :ajaxHtml | self renderItemsOn: ajaxHtml ];
    onComplete: (html logger warn: 'After Update') ),
  (html logger error: 'After AJAX');
value: anItem done;
callback: [ :value | anItem done: value ].
```

Note that we are using the `,` operator to concatenate multiple script snippets. Of course this also works with other JavaScript snippets, not just with logging statements. So we display the strings *Before AJAX* and *After AJAX* right before and after the updater is executed. Furthermore we added another logging statement to the `onComplete:` event handler of the updater. Clicking on a checkbox produces the 3 logging statements in the sequence you see below. This demonstrates nicely that the AJAX request is really processed asynchronously: the text *After Update* appears last in the list of log messages.

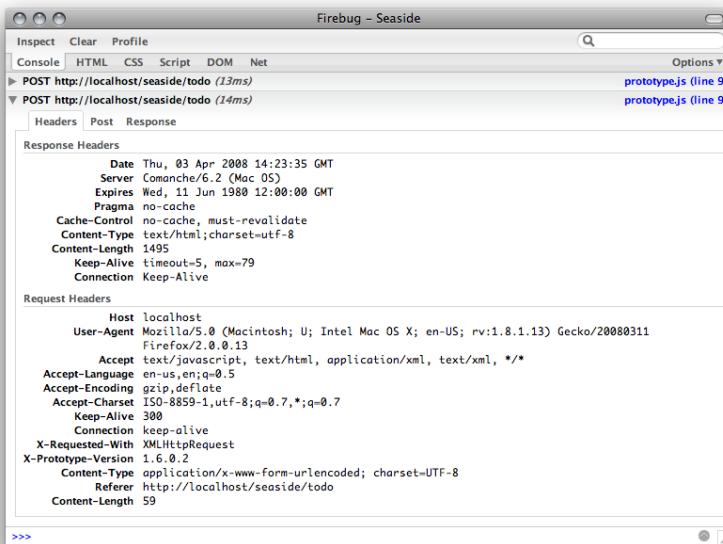


Figure 20.7: The console shows the AJAX requests and responses exchanged between Seaside and the web browser.

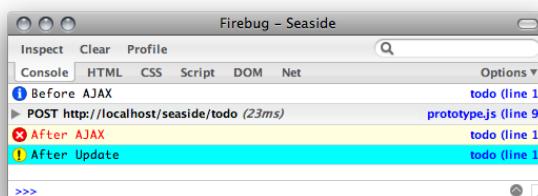


Figure 20.8: Logging to the Firebug console.

Important

Logging. The logging facility only works when using Firebug or the Safari Web Inspector. It might cause JavaScript errors and completely break your code if you try to execute the logging statements within other browsers. Other browsers do not provide the necessary interface to emit log statements.

HTML. Another useful tool is the Firebug HTML tab. The view here is somehow similar to what you see in the source view of the Seaside halos. In Firebug however, you always see the up-to-date DOM tree, even if it has been transformed by JavaScript. For example, if you click a checkbox in the todo application, you will see which parts of the DOM tree change. Make sure that you choose Firebug to highlight and expand changes in the options, so that you don't miss an update operation of your JavaScript library.

In the same window, it is also possible to modify the XHTML nodes and instantly see the effect on the page. You can create, delete and edit attributes and see how the visual appearance of the page changes. If you are looking for the XHTML node of a specific part in your application, click on *Inspect* (or the box-and-pointer icon in recent versions) and select a checkbox in our todo application. Firebug will display the code that defines this control.

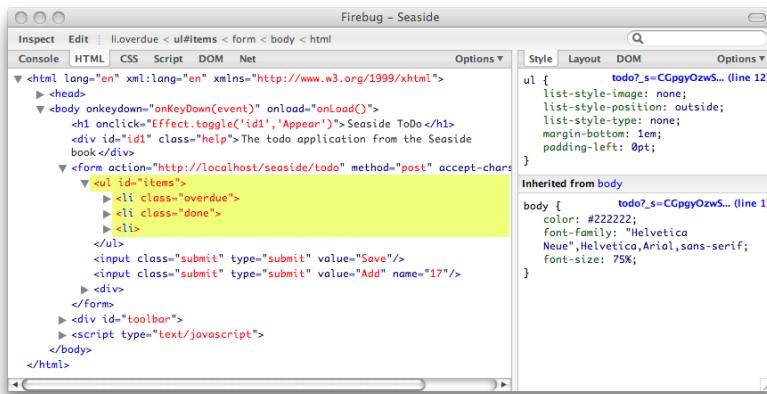


Figure 20.9: The inspector displays an up-to-date view on the DOM tree. Changes are automatically highlighted.

CSS. The CSS tab in Firebug is another valuable tool, that graphic designers find particularly useful. There you will see all the CSS rules that are used on the current page and you can edit and configure them on the fly.

Script. The Script tab is more interesting for programmers. Here you can find the JavaScript files that are used by your application. You can set breakpoints

and step through the code. If you encounter a JavaScript error, you will notice that the Firebug icon in the status bar turns red. This means that you will find a detailed error description in the console. If you click the description, you end up in the debugger investigating the exact cause of the problem. Again, you are able to look at (and change) variables and step through the code.

DOM. The DOM tab shows the complete object graph of the JavaScript engine. You can use it to walk through the objects that the web browser and the JavaScript libraries offer you.

Net. Last but not least the net tab displays a list of all the files your web application depends on. These files consist of html files, style-sheets, JavaScript files, and images and other resources. The tab gives detailed information about how long it took to download the data, which is essential information if you want to optimize your application.

This section has only given you a brief overview of the features available in Firebug to help you inspect and debug your applications. Visit the Firebug website at <http://getfirebug.com/> for more information on how to take advantage of all the capabilities of this tool.

20.8 Summary

As you have seen, introducing JavaScript functionality can allow you to greatly improve the user experience of your applications. Seaside offers a tight integration of JavaScript and AJAX functionality into the core of the framework. It allows you to define JavaScript behaviour using standard Smalltalk, and hides the necessary details from you.

Chapter 21

jQuery

jQuery 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 plugins available that extend the base framework with new functionality. One of the most popular of these plugins is jQuery UI. 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 Smalltalk side of the integration is automatically built from the excellent jQuery documentation, so you can be sure that the integration is up-to-date and feature-complete.

21.1 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.

In order 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;

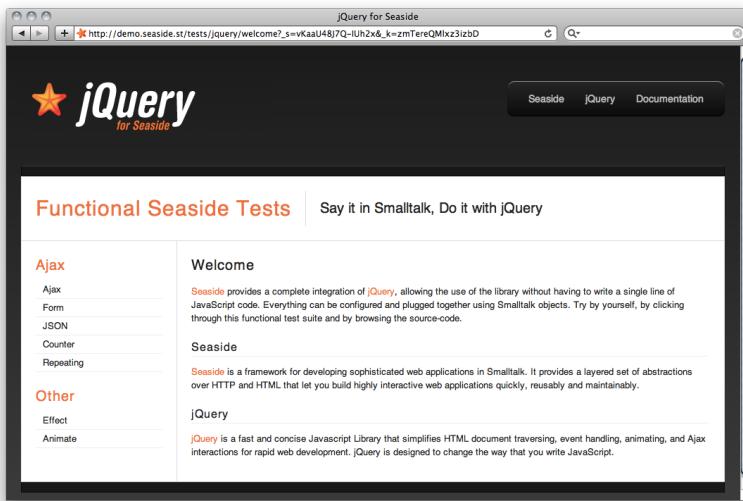


Figure 21.1: jQuery Demo and Functional Test Suite.

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
JQGoogleLibrary	JQuery	Google
JQUIDevelopmentLibrary	JQuery UI	Full
JQUIDeploymentLibrary	JQuery UI	Compressed
JQUIGoogleLibrary	JQuery UI	Google

Advanced

For many of the most popular jQuery plugins there are ready-made Smalltalk wrappers in the Project JQueryWidgetBox on SqueakSource available.

21.2 jQuery Basics

jQuery has a simple but powerful model for its interactions. It always follows the same pattern depicted in Figure 21.2.

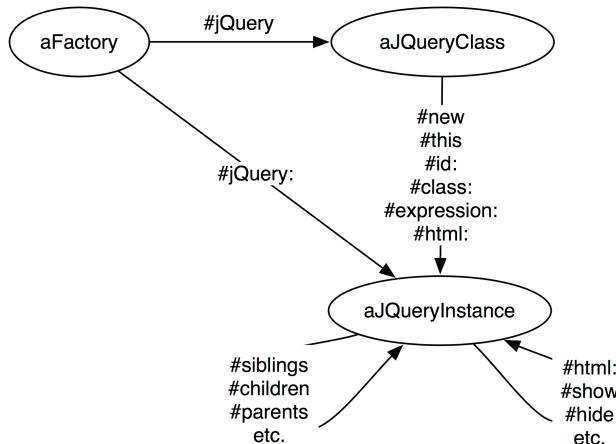


Figure 21.2: jQuery Lifecycle in Smalltalk.

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 Smalltalk instance. `html jQuery` returns an instance of `JQueryClass`.

1. 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'
```

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 21.2.1.

2. 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 21.2.2.

3. 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` class in Smalltalk, and by visiting the jQuery documentation at <http://api.jquery.com/>.

You find more details on performing actions in Section 21.2.3.

21.2.1 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').
```

21.2.2 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.

```
qQuery 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'.
```

21.2.3 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 Content 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 Content 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.
```

21.3 Adding jQuery

After creating a jQuery object on the Smalltalk 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

```
html anchor
  onClick: (html jQuery: 'div')
    remove;
  with: 'Remove DIVs'
```

```
html anchor
  onClick: (html jQuery this)
    remove;
  with: 'Remove Myself'
```

Execute at Load-Time

- Forget about `$(document).ready(...)`
- Seaside has its own mechanism there

```
html document addLoadScript: (html jQuery: 'div') remove
```

21.4 Ajax

Loading

```
aQuery load html: [ :r | r div: Time now ].
```

No Query

```
html jquery ajax.
```

Generators

```
anAjax html: [ :r | r div ].
anAjax script: [ :s | s alert: 'Hello' ].
```

Triggering Callbacks

```
anAjax serialize: aQuery.
anAjax trigger: [ :p | ... ] passengers: aQuery.
anAjax callback: [ :v | ... ] value: anObject.
```

21.5 How To

21.5.1 Click and Show

```

html anchor
  onClick: (html jQuery: 'div.help') toggle;
  with: 'About jQuery'.

html div
  class: 'help';
  style: 'display: none';
  with: 'jQuery is a fast and ...'

```

21.5.2 Replace a Component

```

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'

```

21.5.3 Update Multiple Elements

```

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'

```

21.5.4 Open a Lightbox

```

| 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'

```

21.6 Enhanced ToDo Application

jQuery is an increasingly popular Javascript library. Let's port the the ToDo application to use jQuery for the Javascript functionality, instead of Scriptaculous.

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.

21.6.1 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 renderItemOn: 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 highlights 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.'
```

Note

We need to add the CSS (the same as in the SU example).

```
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;
}'
```

21.6.2 Callbacks Redux

```

ToDoListView>>renderHeadingOn: html

| helpId |
helpId := html nextId.
(html heading)
    class: 'helplink';
    onClick: ((html jQuery id: helpId)
        slideToggle: 1 seconds);
    onClick: ((html jQuery ajax)
        id: helpId;
        callback: [visible := visible not]);
    with: self model title.
(html div)
    id: helpId;
    class: 'help';
    style: (visible ifFalse: ['display: none']);
    with: 'The Enhanced ToDo application.'
```

```

ToDoListView>>renderItem: anItem on: html

(html listItem)
    passenger: anItem;
    class: 'overdue' if: anItem isOverdue;
    class: 'done' if: anItem isDone;
    with:
        [(html checkbox)
            onChange: (((html jQuery id: listId) load)
                serializeForm;
                html: [:ajaxHtml | self renderItemsOn: ajaxHtml]);
            value: anItem done;
            callback: [:value | anItem done: value].
        (html span)
            script: ((html scriptaculous inplaceEditor)
                cancelControl: #button;
                triggerInPlaceEditor: [:value | anItem title: value];
                callback: [:htmlAjax | htmlAjax render: anItem title]);
            with: anItem title. "<-- added"
        html space.
        (html anchor)
            callback: [self edit: anItem];
            with: 'edit'.
        html space.
        (html anchor)
            callback: [self remove: anItem];
            with: 'remove']
```

21.6.3 Drag and Drop

```
ToDoListView>>renderContentOn: html  
  
    self renderHeadingOn: html.  
    html form:  
        [(html unorderedList)  
            id: (listId := html nextId);  
            script: ((html jQuery new sortable)  
                onStop: (html jQuery ajax  
                    callback: [:items | self model items: items]  
                    passengers: (html jQuery this find: 'li'));  
                axis: 'y');  
            with: [self renderItemsOn: html].  
            "html submitButton text: 'Save'."  
        (html submitButton)  
            callback: [self add];  
            text: 'Add'].  
    html render: editor
```

21.6.4 Summary

Chapter 22

Comet

HTTP is unidirectional by design. It is always the client (web browser) that submits or requests information. The web server just waits at the other end and processes whatever requests it receives. For many applications it would be beneficial if the server could propagate events to the clients as they happen. For example a chat application would like to push new messages to all connected users, or our todo application would like to push updates of the model to other people working on the same list.

Even today, the most common solution to this problem is to use polling. That is, the client regularly queries the server for new information. The only way to decrease the latency is to increase the polling frequency, which causes a significant load on the server. In practice one would like to avoid both a high server load and high latency.

In March 2006 Alex Russell coined the term *Comet* in a blog post to describe an alternative approach: one where the server keeps its connection to the client open, and continues to send updates to the client through that connection. The idea that Alex describes in *Comet* was previously known by a variety of terms like ‘reverse AJAX’, ‘HTTP push’, ‘server streaming’, etc, but Alex’ post popularised the concept.

Comet was incorporated into Seaside only a few months after it was introduced.

22.1 Inside Comet

Before we dive into a comet application, let us have a look at precisely how Comet works. The figure depicts the basic interaction between the web

browser and the Seaside server in a Comet setup. The first round-trip between the web browser and Seaside is a normal HTTP request that returns a full XHTML page. This first page includes a small JavaScript snippet, that is executed once the page has finished loading. This starts a new *asynchronous connection* to the server. This connection now persists for a much longer time, essentially as long as the web browser is listening. This connection can then be used by the server to send data to the web browser at any time.

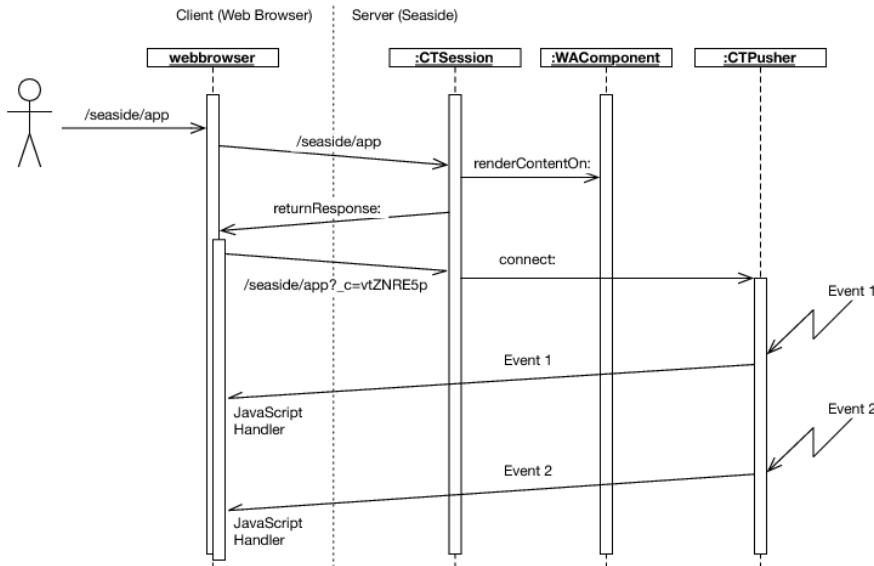


Figure 22.1: Comet enables the server to push data to the client, without that the client has to poll the server for fresh content.

When using Comet, keep in mind that HTTP was not designed to push data from the server to the client. Comet is a hack, even if the details are hidden in Seaside. Luckily the implementation works on all modern web browsers, but there is no guarantee that these tricks will continue to work with the next generation of web browsers. It should be noted that upcoming generations of web browsers will very likely support this type of inverse communication, and there is work on an emerging Comet standard.

Enough ranting, let's give it a quick try.

22.2 Getting Started

The Seaside Comet functionality comes packaged in the package named Comet. Most prebuilt images come with this package preloaded. Point your

browser to `http://localhost:8080/comet/counter`, which is a simple Comet based counter application.



Figure 22.2: Comet requires a server adapter that supports streaming of the response.

Most likely the first time you try to use a Comet application you will end up with an error message as seen above. What does *Streaming-server required* mean? On most Smalltalk platforms Seaside does not stream responses by default. This means that the response is sent to the client only after the complete body is generated. Obviously this does not work for Comet, since we need to continually send data to the client.

The default server adapter on Squeak does not support streaming and neither does Swazoo. Instead, you need to use a special wrapper around Comanche, called `WAListenerAdaptor`. Evaluate `WAListenerAdaptor startOn: 8888` to start the server. Note that we are using a different port number here, to avoid a conflict with the normal server that continues to work on port 8080.

If you try to access your application again (with the new port number), you will be able to see the counter application. Open multiple browser windows, preferably with different browsers and using different machines. As you can see below, clicking on the links in one application propagates immediately to the other windows. Note that every window has its own Seaside session, it is just the global count variable that is synced with all the currently connected sessions.

Let's have a look at the implementation and study the differences from our first counter implementation at the beginning of this book.

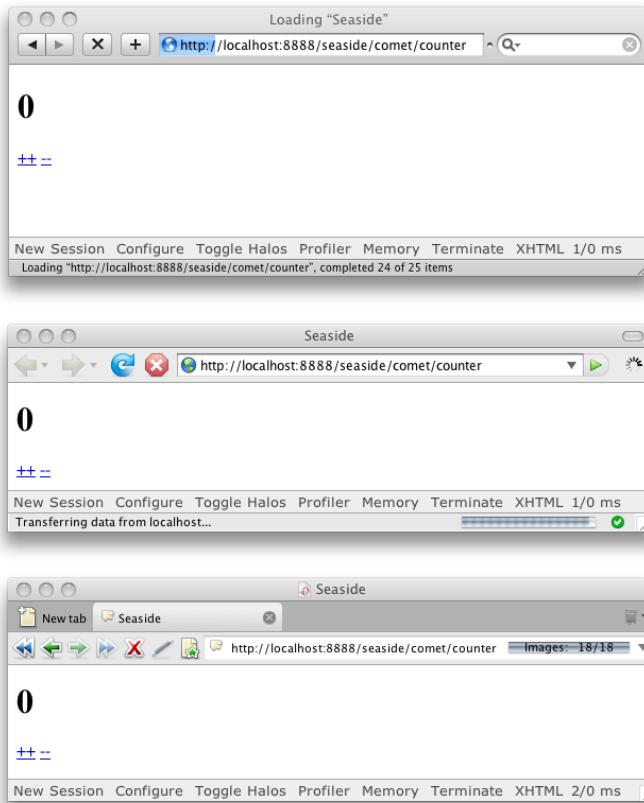


Figure 22.3: The counter application synchronized along different web browsers.

22.3 The Counter Explained

`CTCounter` is a subclass of `WAComponent`:

```
WAComponent subclass: #CTCounter
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Comet-Examples'
```

It has an `initialize` method to register it as an application.

```
CTCounter class>>initialize
| application |
application := WAAdmin register: self asApplicationAt: 'comet/counter'.
application addLibrary: JQDeploymentLibrary; addLibrary: CTLibrary
```

There are two JavaScript libraries included here. The first is `JQDeploymentLibrary` which is the `jQuery` library that we saw in Chapter 21. The second library is the Comet JavaScript library `CTLlibrary`. `CTLlibrary` does not depend on functionality provided by the `jQuery` library, but we will use some of that functionality later to update the value of the counter.

Unlike the counter application we saw in previous chapters, the Comet counter requires global state. That is, the model is not part of the component which is local to a single session but global to all sessions so that it is shared among all users. To do this, we keep the model – an instance of `CTCounterModel` – on the class side of the `CTCounter` component. Furthermore we need a dedicated pusher object – an instance of `CTPusher` – that is responsible for managing the communication channel between the server and many clients.

```
CTCounter class
    instanceVariableNames: 'model pusher'
```

```
CTCounter class>>model
    ^ model ifNil: [ model := CTCounterModel new ]
```

```
CTCounter class>>pusher
    ^ pusher ifNil: [ pusher := CTPusher new ]
```

Important

Its worth re-emphasizing that the pusher requires global state so that all browsers connecting to the page share the same model. Thus we store the pusher and its model on the class-side.

The method `renderContentOn:` is unremarkable. First we render the current count in a heading we give the ID `count`. Then we have two anchors with JavaScript actions attached to the click event, that call the methods `increase` and `decrease`. Finally, we append a small script to the bottom of the component that connects the pusher we defined on the class side to this session and component.

```
CTCounter>>renderContentOn: html
    html heading
        id: 'count';
        with: self class model count.
    html anchor
        onClick: (html jQuery ajax
            callback: [ self class decrease ]);
        with: '--'.
    html space.
    html anchor
        onClick: (html jQuery ajax
            callback: [ self class increase ]);
        with: '++'.
```

```
html script: (html comet
  pusher: self class pusher;
  connect)
```

Where does all the magic happen now? Where are all the connected components updated? Obviously this must happen in the methods `increase` and `decrease`, which both call the method `update`.

```
CTCounter>>increase
  self class model increase.
  self update
```

```
CTCounter>>decrease
  self class model decrease.
  self update
```

```
CTCounter>>update
  self class pusher javascript: [ :script |
    script << (script jquery: #count)
    text: self class model count ]
```

Nothing surprising happens in `increase` and `decrease`, both methods just delegate the action to their model. The interesting thing happens in `update` right after the model has been changed. As you can see, we tell the pusher that we want to push a script to the client. We do this by sending a block to the method `javascript:` that gives us a `script` object. This script updates the element with the ID `count` to the current count of the element. The pusher ensures that the script is automatically sent to all connected components.

22.4 Summary

In this chapter we saw how we can use bleeding edge Web 2.0 technology in Seaside. The Comet technology circumvents the common restrictions of the asymmetric HTTP protocol, and allows one to change a website by initiating the update event from the server. Keep in mind that Comet is a browser hack and that you might run into scalability problems quickly. The *Web Hypertext Application Technology Working Group* proposes a standard as part of the HTML 5 draft recommendation. Rest assured that as soon as major web browsers implement an official standard for server push, Seaside will be among the first to support the new technology.

Part VI

Advanced Topics

This part presents some aspects that you face when you deploy Seaside applications for real. It presents how to configure and deploy an application.

While Seaside keeps as much state on the server side as possible, it supports the creation and integration of REST web services as well. This part covers how to write a REST API and integrate it with an existing application.

Although Seaside does not offer a built-in persistency framework, the issue of how to manage your data is a common concern when building web applications. This part also covers a selection of approaches to managing data persistency in Smalltalk, and how these work with Seaside.

Finally, we present Magritte, which is a metadata framework with Seaside integration: using Magritte allows you to generate forms on the fly without hard-coding HTML or generating XML files.

Chapter 23

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 Section 23.1 we are going to have a look at some best practices before deploying an application. Then in Section 23.2 we introduce Seaside-Hosting, a simple and free hosting service for non-commercial Seaside applications. Next, in Section 23.3 we present how to setup your own server using Apache. Last but not least, in Section 23.4, we demonstrate ways to maintain a deployed image.

23.1 Preparing for Deployment

Because Smalltalk 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.

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 Core or Pharo Kernel 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
```

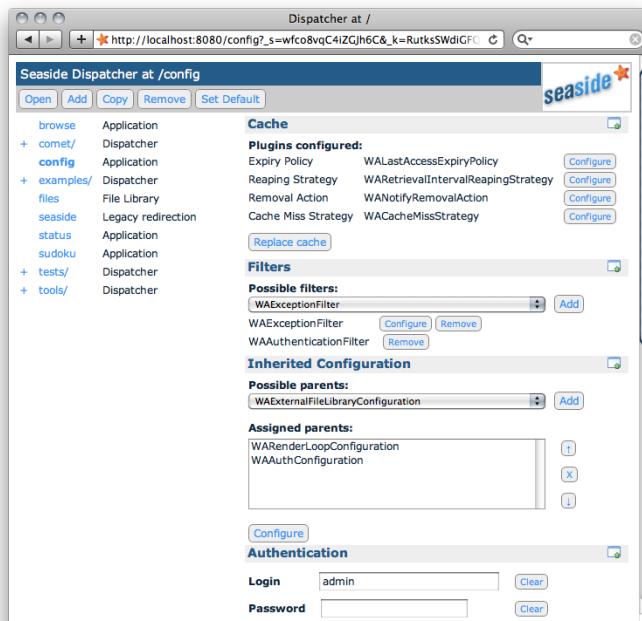


Figure 23.1: Configure an application for deployment.

```

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:

```

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.

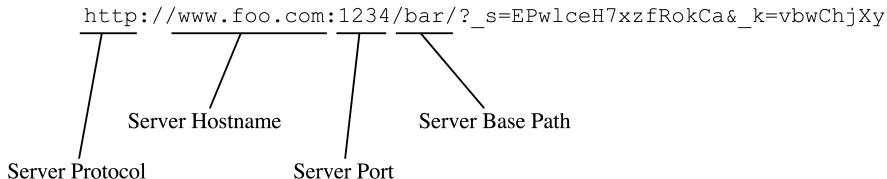


Figure 23.2: Configuration options for absolute URLs.

Have a look at Figure 23.2 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: '/'
```

23.2 Seaside-Hosting

To deploy an application on a public server, Seaside-Hosting (<http://www.seasidehosting.st>) is undoubtedly the easiest way to get started. Seaside-Hosting is a free hosting service for Seaside applications sponsored by netstyle.ch GmbH and ESUG. The highest-profile example of a website running on Seaside-Hosting is www.seaside.st itself. Furthermore the portal and management application of Seaside-Hosting is hosted in itself.

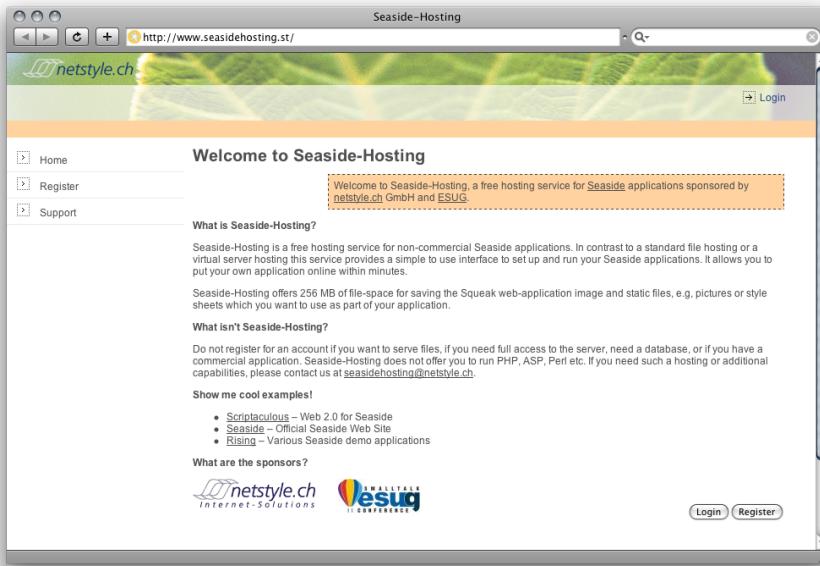


Figure 23.3: Seaside-Hosting Portal.

Before getting started you need to be aware of some key conditions that apply when using Seaside-Hosting:

- Seaside-Hosting is for non-commercial applications only. If you plan to make money with your application, you need to look somewhere else.
- Seaside-Hosting only provides hosting for Seaside applications that are developed in Pharo.
- Seaside-Hosting does not provide access to any external databases or to the underlying operating system. Many sites are hosted on the same machine and therefore there are some restrictions to avoid people harming the servers and other hosted applications.

The advantage, on the other hand, is that you can put your application online within minutes. There is no need for you to worry about the server, about installing a web server, about serving static files or about getting a domain name. You upload the application and you are online instantly. For up-to-date information on Seaside-Hosting, please check out the FAQ on the web-site.

Get an account. To get started with Seaside-Hosting register for a free account. In the process you will be able to choose a domain name of the form `yourname.seasidehosting.st`, where you can choose `yourname` freely. Click on the link in the confirmation mail to validate your e-mail address.

Upload an image. Next, upload your required *image* and *changes* files through the web interface or use FTP as described in the *Filesystem* section of the portal. When using the file upload through the web, you are also able to upload a zip archive of your files and automatically decompress them on the server.

If you have static files to serve, such as style sheets, javascript files, images or other resources that are not part of the uploaded image, you can put these into the folder `/resources`. Make sure to set the setting *Resource Base Path* to `http://yourname.seasidehosting.st/resources` so that you can easily refer to these files from within your application.

Starting your application. To start your application go to *Status*, select the image you uploaded, choose the server adapter and click on *Start*. This will launch a new Squeak VM on the server running your image. Your application should now be reachable from the world wide web. Point your browser to `http://yourname.seasidehosting.st/<your_appname>` to test the application. Note that even if you started a server on a different port in your image (for example port 8080), Seaside-Hosting conveniently serves your application on the default port. So when browsing your application there is no need to type a port number.

Configure Path. As a last step you might want to make your application be served from the root of your domain-name, so that people only have to type `http://yourname.seasidehosting.st/`. In the Seaside configuration interface select your preferred application as default entry point. Then go

to the configuration of this application and change the base-path to `/`. The application is now reachable without typing a path.

Getting your own domain. It is possible to use your own top level domain name on Seaside-Hosting. Go to *Status* and select *Server Alias* from the toolbar. You'll be asked for the new domain name and then instructions are given on how to update the DNS entry to point to Seaside-Hosting. This can be done through the registrar of the domain name. The Seaside application is now reachable through `seasidehosting.st` and your own personal domain.

An annoyance of Seaside-Hosting is that whenever you want to update your web-application you need to upload an image. This is especially an issue if you have a slow internet connection and your images are relatively big. To avoid this issue you might want to look at the class `WAVersionUploader` that is normally installed at `/tools/versionuploader`. It is a simple web interface to the Monticello versioning system and allows one to remotely update and load packages into a running image. Try it out locally first, to see if it suits your needs. Again, make sure to password protect the application, if you plan to use it in a public deployment.

23.3 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.

23.3.1 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.
```

23.3.2 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 seamless 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
```

23.3.3 Installing the Squeak VM

Depending on the Smalltalk dialect you are using, the installation of the VM is different. Installing Squeak on a Debian system is simple. Install Squeak 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.

23.3.4 Running the VMs

Before we hook up the Smalltalk 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 Section 23.1 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"
```

Again, we are using Squeak in this example. You need to check out the documentation of your Smalltalk VM, if you are running with a different dialect. Let's have a quick look at the different parts of the script.

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 one 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.

Important

Squeak and Pharo both display 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 Section 23.4.2.

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.

23.3.5 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 Section 23.1

`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.

23.3.6 Serving files 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 17 such files can be easily served through the image, however the same drawbacks apply here as those listed in Section 23.3.

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:

<code>favicon.ico</code>	A shortcut icon, which most graphical web browsers automatically make use of. The icon is typically displayed next to the URL and within the list of bookmarks.
<code>robots.txt</code>	A robots exclusion standard, which most search engines request to get information on what parts of the site should be indexed.
<code>resources/</code>	A subdirectory of resources used by the graphical designer of your application.
<code>resources/css/</code>	All the CSS resources of your application.
<code>resources/script/</code>	All the external Javascript files of your application.

23.3.7 Load Balancing Multiple Images

There are several ways to load balance Seaside images, one common way is to use `mod_proxy_balancer` 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 noclobber=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.

This is it, the cluster is ready to go. If enabled the server manager can be found at <http://localhost/balancer-manager>, see Figure 23.4, and the server status can be queried on <http://localhost/server-status>. Note that before going to

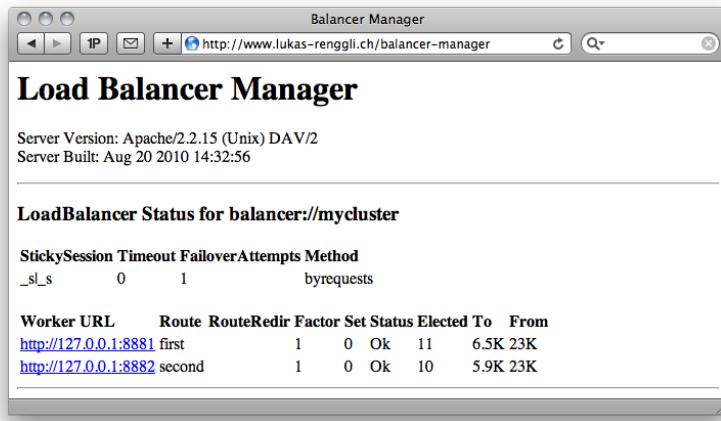


Figure 23.4: Apache Load Balancer Manager.

production these two admin applications need to be properly protected from unauthorized access.

23.3.8 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_ajp 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 23.3.5. 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.squeaksources.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 `AJPPPharoAdaptor` on the correct port from within your image (8003 in this example) and your web application is up and running with AJP.

23.4 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 Smalltalk VM. There are several possibilities to do that, we are going to look at those in this section.

23.4.1 Headful System

Instead of running the VM headless as we did in Section 23.3.4, 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.

23.4.2 Virtual Network Computing

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.

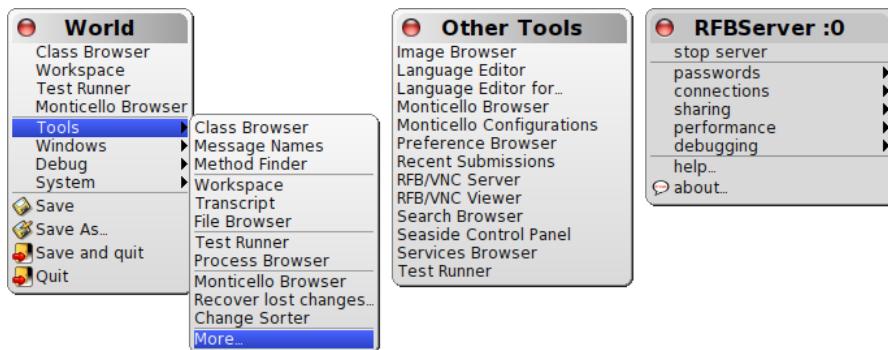


Figure 23.5: Starting the RFB Server in Pharo.

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/>.

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.

23.4.3 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 Figure 23.6), the virtual machine, Seaside, the garbage collector and the running processes.



Figure 23.6: 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.

Important

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.

23.4.4 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.

Advanced

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: ''
    poolDictionaries: ''
    category: '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 Smalltalk 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=SystemVersion current`. 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 23.1.

Chapter 24

REST Services

Seaside is not built around REST services by default, to increase programmer productivity and make 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 15. 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.

24.1 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 Architectures. 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 Figure 24.1.

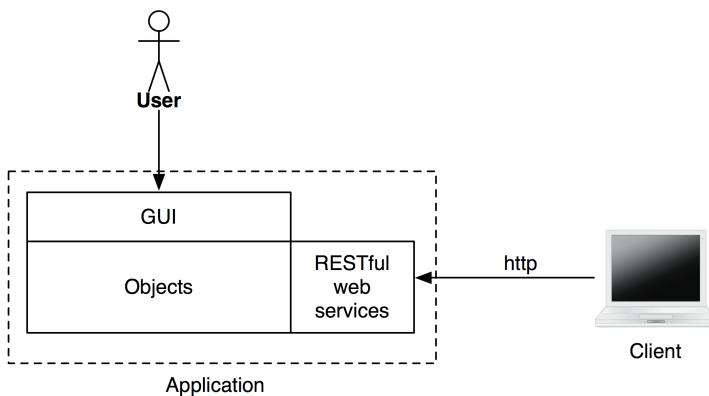


Figure 24.1: First architecture: adding REST to an existing application.

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 Figure 24.2.

This second approach offers a low coupling and eases deployment. Load-balancing 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.

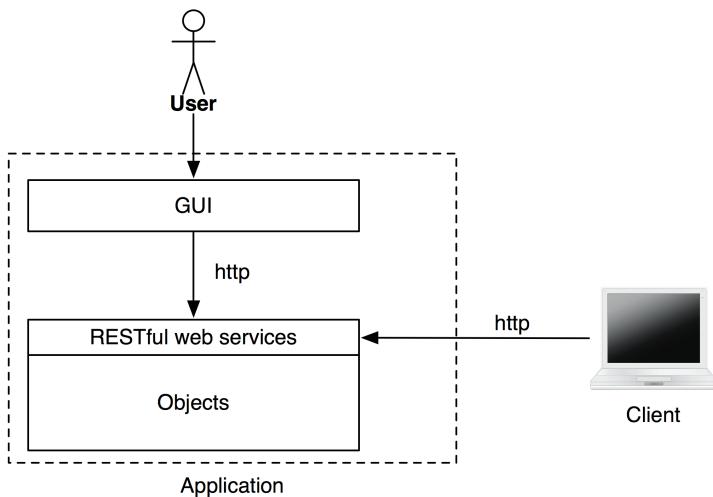


Figure 24.2: Second architecture: REST centric core.

24.2 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 them easily with the following Gofer script:

```

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.

24.2.1 Defining a Handler

We are going to extend the todo application from Chapter 15 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: ''
  poolDictionaries: ''
  category: 'ToDo-REST'
```

Note

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 handler 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'
```

24.2.2 Defining a Service

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 Section 24.3, 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 application. 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 ] ]
```

24.3 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 Section 24.3.1 we are going to look at matching other request types, such as POST and PUT. In Section 24.3.2 we are going to see how to serve different content types depending on the requested data. In Section 24.3.3 we will see how to match path elements and in Section 24.3.4 how to extract query parameters.

24.3.1 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

24.3.2 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:

```
$ curl -H "Accept: text/json" http://localhost:8080/todo-api
[{"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>
    ...
  </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
```

24.3.3 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 Section 24.2.2 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
```

24.3.4 Query Parameters

So far we used the request type (Section 24.3.1), the content type (Section 24.3.2) and the request path (Section 24.3.3) 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
```

24.3.5 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.

24.4 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: ''
  poolDictionaries: ''
  category: 'ToDo-REST'
```

A filter cannot be registered as 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 to 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 Figure 24.3. 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

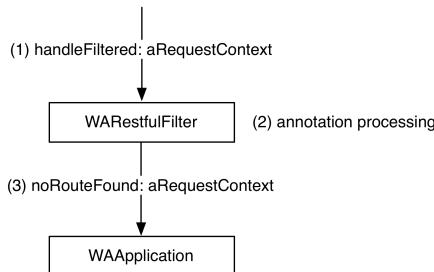


Figure 24.3: Request handling of WARestfulFilter and WAAplication.

example, the following code triggers the wrapped application whenever HTML is requested:

```

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.

24.5 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 Figure 24.4.

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.
  
```

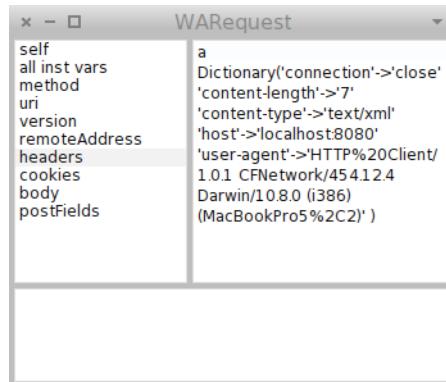


Figure 24.4: Inspecting a request.

```
^ 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`.

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).

24.6 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 can be 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 25

Some Persistency Approaches

One important question when building applications is how to save their data. This question is not directly linked to Seaside, but we want to provide some possible solutions to help you get started. Several solutions exist to save your data. We will present some of the ones that are available in Squeak and that are based on open-source software. We will cover multiple solutions based on image snapshots, active record-like behavior, and object-oriented databases. We will not cover in detail Object-Relational mapping solutions such as GLORP (an open-source object-relational mapper). The book “An Introduction to Seaside” by Michael Perscheid et al. presents a nice introduction to GLORP and we point the interested reader to this book. Cincom offers WebVelocity (see Chapter 3) to support the definition of web applications using Seaside and scaffolding of applications. For the persistency, WebVelocity uses GLORP. There are some efforts to support OpenDBX in Squeak. The list of support will certainly grow and we may extend it as well.

The purest definition of “persistency” is capturing the state of memory to secondary storage (disk) so that work on the meaningful object model may continue later; and this is the primary purpose of any persistency framework. Another useful purpose, however, is to accommodate very large models that cannot fit entirely in RAM. Capable persistency frameworks manage large models by paging portions of the model into RAM as necessary for consumption or updates, and paging it out to make room for other parts of the model. One job of an object-database is to do this as transparently to the developer as possible.

Of course, Smalltalk vendors offer specific solutions for persistency ranging from object-relational mapping to object-oriented databases. GemStone is an

object database that natively runs Seaside (see Chapter 4). The GemStone approach offers a robust and scalable object-oriented environment for Seaside. With GemStone you don't have to think about persistency at all, the fact that objects are persistent in an industrial strength database is part of the language. GemStone offers GLASS, a 4 GiB persistent image that just magically solves most of your persistency problems. GLASS can be a really important option for persistency when you really need to scale in the Seaside world; however, it requires a 64 bit server and introduces the small additional complexity of changing to a different Smalltalk and learning its class library.

A little note. Most applications don't need to scale. They need to be robust and solve client problems. Most real world applications are written to run small businesses. In all likelihood, scaling is not and probably won't ever be your problem. We might like to think we're writing the next YouTube or Twitter, but odds are we're not. You can make a career just replacing spread sheets with simple applications that make people's lives easier without ever once hitting the limits of a single Squeak image (such was the inspiration for DabbleDB), so don't waste your time scaling (yet). Spend your time on solving client problems and building robust software. If scaling ever becomes a problem, be happy, it's a nice problem and your clients will pay for it.

In contrast to other “web frameworks” Seaside does not provide a ready-made persistency solution for you. We consider this to be an advantage, Seaside lets you choose the database technology that fits your needs the best and concentrates on what it is strong at, web application development.

In this chapter we will present several persistency solutions of various degrees of scalability and simplicity. First we will describe some simple image-based and object serialization solutions. Then we present SandstoneDB which offers an *Active Record*-style API (even if it does not offer transactional semantics, concurrent accesses) and use the image as a persistency. Finally we present Magma: an object-oriented database based on a write-barrier. This approach is similar to the one of GOODS.

25.1 Image-Based Persistence

While a Smalltalk image should not really be used as an artifact of code management (you should use Monticello packages, change-sets, Store packages to manage your code elements and build your image from packages), it can be used to store objects. With some precaution you can use the image as a simple and powerful object-oriented database. Therefore you can delay the need to hook up a database during much of your development and often your deployment. The point is to find the adequate solution for your problem.

On his blog Ramon Leon advocates that not all applications need a relational-database back-end, and he explains some of the advantages of a lighter-weight approach at <http://onsmalltalk.com/simple-image-based-persistence-in-squeak/>.

Understanding the right level of database is important since it will lower the stress on the development. This is why solutions like Prevayler based on the Command design pattern have emerged over the years. Such approaches mimic the notion of the Smalltalk image, even if they offer a better store granularity.

Not directly using a relational database will also ease the evolution of your application which in case of the prototyping phase will certainly do. In addition, working with full objects all the way down is more productive. So let's have a look at image storage mechanisms.

The simplest approach is the following: you save your image. Now if the image crashes because for example your disc is full, you are in trouble. The second level is to perform several backups. Later on you can switch to an object-oriented database approach such as GOODS, Magma or GemStone. Of course saving an image does not work well if you have to share data between different applications not running in the same image. So you get the simplicity and the limits of simplicity too.

Saving an image. The expression `SmalltalkImage current saveSession` saves an image, i.e., all the objects that are accessible in your system. Now based on that we can build a small utility class. Let us define `ImageSaver` as a class for saving the image.

```
Object subclass: #ImageSaver
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ImageSaver'

ImageSaver class>>saveImage
    SmalltalkImage current saveSession
```

Now the question is when do we save our data. For the ToDo application, each time an item is changed, added or removed would be a possibility. Having an explicit save button is another solution. We let you decide for your application.

On a Mac Book pro with around ten applications running in parallel, it takes about 1100 ms to save the Seaside image which highly depends on the size of the image. Therefore, this will have an influence on choice. For two lines of code, this is a good tradeoff. Now we will use the solution proposed by Ramon Leon to improve the robustness on crashes of the approach.

Backing Up images. Using the image itself as a database is not free of problems. An average image is well over 30 megabytes, saving it takes a bit of time, and saving it while processing http requests is a risk you want to avoid. In addition you want to avoid having several processes saving the image.

ReferenceStream provides a solution to serialize objects to disk. On every change you just snapshot the entire model. Note that this isn't as crazy as it might sound, most applications just don't have that much data. If you're going to have a lot of data, clearly this is a bad approach, but if you're already thinking about how to use the image for simple persistence because you know your data will fit in ram, here's how Ramon Leon does it.

We define a simple abstract class that you can subclass for each project. With a couple of lines you get a Squeak image-based persistent solution which is fairly robust and crash proof and more than capable enough to allow you just to use the image without the need for an external database.

```
Object subclass: #SMFileDatabase
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SimpleFileDb'

SMFileDatabase class
  instanceVariableNames: 'lock'
```

All the methods that follow are class-side methods. First, we'll need a method to fetch the directory where rolling snapshots are kept. Note that we use the name of the class as the directory entry.

```
SMFileDatabase class>>backupDirectory
  ^ (FileDirectory default directoryNamed: self name) assureExistence.
```

The approach here is simple, a subclass should implement `repositories` to return the root object to be serialized. Therefore we often just return an array containing the root collection of each domain class.

```
SMFileDatabase class>>repositories
  self subclassResponsibility
```

The subclass should also implement `restoreRepositories:` which will restore those repositories back to wherever they belong in the image for the application to use them.

```
SMFileDatabase class>>restoreRepositories: someRepositories
  self subclassResponsibility
```

Should the image crash for any reason, we want the last backup to be fetched from disk and restored. So we need a method to detect the latest version of the backup file, which we will tag with a version number in when saving.

```
SMFfileDatabase class>>lastBackupFile
^ self backupDirectory fileNames
detectMax: [:each | each name asInteger]
```

Once we have the file name, we'll deserialize it with a read-only reference stream.

```
SMFfileDatabase class>>lastBackup
| lastBackup |
lastBackup := self lastBackupFile.
lastBackup ifNil: [ ^ nil ].
^ ReferenceStream
    readOnlyFileNamed: (self backupDirectory fullNameFor: lastBackup)
do: [ :f | f next ]
```

This requires you extend the class `ReferenceStream` with `readOnlyFileNamed:do:` as follows. This way you do not have to remember to close your streams.

```
ReferenceStream class>>readOnlyFileNamed: aName do: aBlock
| file |
file := self oldFileNamed: aName.
^ file isNil
    ifFalse: [ [ aBlock value: file ] ensure: [ file close ] ]
```

Now we can provide a method to actually restore the latest backup. Later, we will make sure this happens automatically.

```
SMFfileDatabase class>>restoreLastBackup
self lastBackup
ifNotNilDo: [ :backup | self restoreRepositories: backup ]
```

We provide a hook with a default value representing the number of old versions.

```
SMFfileDatabase class>>defaultHistoryCount
^ 15
```

Now we define a method `trimBackups` that suppresses the older versions so that we do not fill up the disc with more data than needed.

```
SMFfileDatabase class>>trimBackups
| entries versionsToKeep |
versionsToKeep := self defaultHistoryCount.
entries := self backupDirectory entries.
entries size < versionsToKeep ifTrue: [ ^ self ].
((entries sortBy: [ :a :b | a first asInteger < b first asInteger ])
allButLast: versionsToKeep)
do: [ :entry | self backupDirectory deleteFileNamed: entry first ]
```

Note that you can change this strategy and keep more versions.

Serializing Data. Now we are ready to actually serialize the data. Since we want to avoid multiple processes to save our data at the same time, we will invoke `trimBackups` within a critical section, figure out the next version number, and serialize the data (using the method `newFileNamed:do:`), ensure to flush it to disk before continuing. Let's define the method `newFileNamed:do:` as follows.

```
ReferenceStream class>>newFileNamed: aName do: aBlock
| file |
file := self newFileNamed: aName.
^ file isNil
    ifFalse: [ [ aBlock value: file ] ensure: [ file close ] ]
```

```
SMFileDatabase class>>saveRepository
| version |
lock critical: [
    self trimBackups.
version := self lastBackupFile isNil
    ifTrue: [ 1 ]
    ifFalse: [ self lastBackupFile name asInteger + 1 ].  

ReferenceStream
    newFileNamed: (self backupDirectory fullPathFor: self name) , '.' ,
version asString
    do: [ :f | f nextPut: self repositories ; flush ] ]
```

So far so good, let's automate it. In Squeak, we can register classes so that their method `shutDown:` is called when the image is quit and `startUp:` when the image is booting. Using this mechanism, we can make sure that when the image is saved a backup is automatically performed and will be automatically restored at startup time. This way if your computer crashes, relaunching the image will automatically load the latest backup.

We'll add a method to schedule the subclass to be added to the start up and shutdown sequence. Note that you must call this for each subclass, not for this class itself. This method also initializes the lock and must be called before `saveRepository` since this is cleaner. To achieve this behavior, we use the `addToStartUpList:` and `addToShutDownList:` messages as follows:

```
SMFileDatabase class>>enablePersistence
lock := Semaphore forMutualExclusion.
Smalltalk addToStartUpList: self.
Smalltalk addToShutDownList: self
```

So on shutdown, if the image is actually going down, we just save the current data to disk by specializing the method `shutDown:..`

```
SMFileDatabase class>>shutDown: isGoingDown
isGoingDown ifTrue: [ self saveRepository ]
```

And on startup we can restore the last backup by specializing the method `startUp:..`

```
SMFiteDatabase class>>startUp: isComingUp
    isComingUp ifTrue: [ self restoreLastBackup ]
```

Now, if you want a little extra snappiness and you're not worried about making the user wait for the flush to disk, we'll add little convenience method for saving the repository on a background thread.

```
SMFiteDatabase class>>takeSnapshot
    [self saveRepository]
        forkAt: Processor systemBackgroundPriority
        named: 'snapshot: ', self class name
```

Now for the ToDo application. We create `ToDoFileDatabase` as a subclass of the class `SMFiteDatabase`.

```
SMFiteDatabase subclass: #ToDoFileDatabase
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ImageSaver'
```

We make sure that the persistency is enabled by specializing the class `initialize` method as follows:

```
ToDoFileDatabase class>>initialize
    "self initialize"
    self enablePersistence
```

Now the list of items is the only root of our object model so we specify it as entry point for the store in the `repositories` method.

```
ToDoFileDatabase class>>repositories
    ^ ToDoList default
```

Since we need a way to change the current list of todo items we extended the class with the method `default:` that is defined as follows:

```
ToDoList class>>default: aToDoList
    Default := aToDoList
```

```
ToDoFileDatabase class>>restoreRepositories: someRepositories
    ToDoList default: someRepositories
```

We modify the method `renderContentOn:` of the `ToDoListView` to offer the possibility of saving.

```
ToDoListView>>renderContentOn: html
    html heading: self model title.
    html form: [
        html unorderedList: [ self renderItemOn: html ].
        html submitButton
```

```

    text: 'Save' ;
    callback: [ ToDoFileDatabase saveRepository].
html submitButton
    callback: [ self add ];
    text: 'Add' ].
html render: editor

```

The expression `ToDoFileDatabase restoreLastBackup` lets you restore the latest backup.

This solution offers a simple persistency mechanism that is more robust and easier than just saving an image. It works for those small projects where you really don't want to bother with a real database. Just sprinkle a few `MyFileDbSubclass saveRepository` or `MyFileDbSubclass takeSnapshot`'s around your application code whenever you feel it is important, and you're done.

On the one hand, saving the image is easy, but on the other it saves all the data. Let us now take a look at other approaches that can select what is saved.

25.2 Object Serialization

A variation on the same principle is to simply serialize your data when it changes. Here is a 6 method long solution. There are several frameworks available: SIXX saves objects in XML format, ReferenceStream (SmartRefStream) in Pharo, BOSS (Binary object storage system) in VisualWorks, and ObjectDumper in GNU Smalltalk allow one to serialize objects in a binary format. These frameworks support cyclic references.

To illustrate this simple method, we show how Conrad, the conference registration system for ESUG (<http://www.squeaksource.com/Conrad>), applies such an approach. The root of the model is `CRConference`. Whenever the application changes something in the model it calls `CRConference>>snapshot` which saves the complete model. At start up we make sure that the latest version is always loaded.

The method `CRConference class>>load:` loads a specific version. `CRConference class>>initialize` and `CRConference class>>startup` make sure that always the latest version is loaded.

Since Conrad only manages one conference, it uses an singleton that can be accessed using the message `default` and reset using the message `reset`.

```

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

```

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

```
CRConference class>>load: aString
| stream |
stream := ReferenceStream fileNamed: aString.
[ Default := stream next ] ensure: [ stream close ]
```

Afterwards we make sure that at image start up, the latest version is loaded. To do so, we specialize the class method `startUp` which is invoked when the image is starting up.

```
CRConference class>>startUp
| files |
self reset.
files := FileDirectory default fileNamesMatching: '*.obj'.
files isEmptyIfFalse: [ self load: files asSortedCollection last ]
```

```
CRConference class>>initialize
Smalltalk addToStartUpList: self
```

Now we save the data, by creating a reference stream in which we add date and time information for tracing purposes.

```
CRConference>>snapshot
| stream |
stream := ReferenceStream fileNamed: (String streamContents: [ :filename |
Date current
printOn: filename
format: #( 3 2 1 $- 1 1 2 ).
filename space; nextPutAll: (Time current print24
collect: [ :char | char = $: ifTrue: [ $- ] ifFalse: [ char ] ]).
filename nextPutAll: '.obj' ]).
[ stream nextPut: self ] ensure: [ stream close ]
```

We illustrated the principles using `ReferenceStream` but it would be the same with another object serializer.

Again with 6 methods you get a working and robust solution that has some limits: first this is the responsibility of the developer to track when to save the objects, second, saving all the data each time one single element changes is not optimal and works for small models. Third with large data, saving subpart can be tedious because you may reload parts and you may have to swap the one in the image with the one loaded.

25.3 Sandstone: an Active-Record Image-based Approach

SandstoneDB has been developed by Ramon Leon. SandstoneDB is a lightweight Prevayler style embedded object database with an ActiveRecord API. It is available for Pharo and GNU Smalltalk. SandstoneDB doesn't require a command pattern and works for small apps that a single Pharo image can handle. SandstoneDB is a simple, fast, configuration free, crash proof, easy to use object database that doesn't require heavy thinking to use. It allows you to build and iterate prototypes and small applications quickly without having to keep a schema in sync. SandstoneDB is a simple object database that uses `SmartRefStream` to serialize clusters of objects to disk (compared to the above approach it can save the model in increments, not the whole model when only something small changes).

The idea is to make a Squeak image durable and crash proof and suitable for use in small office applications. SandstoneDB and Seaside give what the Rails and ActiveRecord guys have, simple fast persistence that just works, simply. It also gets the additional benefit of no mapping and no SQL queries, instead we use plain Smalltalk iterators.

With Sandstone, data is kept in RAM for speed and on disk for safety. All data is reloaded from disk on image startup. Since objects live in memory, concurrency is handled via optional record level critical sections rather than optimistic locking and commit failures. It's up to the developer to use critical sections at the appropriate points by using the `critical` method on the record. Saves are atomic for an ActiveRecord and all its non-ActiveRecord children, for example, an order and its items. There is no atomic save across multiple ActiveRecords. A record is a cluster of objects that are stored in a single file together.

Contrary to the image-based persistence schema described at the beginning of this chapter, SandstoneDB is more like an OODB. It slices out part of the object graph and commits just that record and its children to a single temp file. Once successfully written it's renamed into place to make the commit as atomic as possible. First the new record is written to a file named `objectid.new`, then the current record which is named `objectid.obj` is renamed to `objectid.obj.version`, and the change is finally committed by renaming `objectid.new` to `objectid.obj`. The recovery process takes this into account and can tell at what point the crash occurred by what the file names are and recovers appropriately. There's a recovery process that runs on image startup to finish partial commits and clean up failed commits that may have happened during a crash. Since commits on objects are explicit, there's no need to for any kind of change notification or change tracking.

About Aggregate. The root of each cluster is an ActiveRecord. It makes

ActiveRecord a bit more object-oriented by treating it as an aggregate root and its class as a repository for its instances.

A good example of an aggregate root object is an `Order` class, while its `LineItem` class just be an ordinary Smalltalk object. A `BlogPost` is an aggregate root while a `BlogComment` is an ordinary Smalltalk object. `Order` and `BlogPost` would be ActiveRecords. This allows you to query for `Order` and `BlogPost` but not for `LineItem` and `BlogComment`, which is as it should be, those items don't make much sense outside the context of their aggregate root and no other object in the system should be allowed to reference them directly, only aggregate roots are referenced by other ActiveRecords. This will cause the entire cluster to be committed atomically by calling `anOrder commit`.

To start. To use SandstoneDB, just subclass `SDActiveRecord` and save your image to ensure the proper directories are created, that's it, there is no further configuration. The database is kept in a subdirectory matching the name of the class in the same directory as the image. Following the idea of Prevayler all data is kept in memory, then written to disk on save or commit and on system startup, all data is loaded from disk back into memory. This keeps the image small. Like Prevayler, there's a startup cost associated with loading all the instances into memory and rebuilding the object graph, however once loaded, accessing your objects is blazing fast and you don't need to worry about indexing or special query syntaxes like you would with an on disk database. This of course limits the size of the database to whatever you're willing to put up with in load time and whatever you can fit in RAM.

25.3.1 The SandstoneDB API

SandstoneDB has a very simple API for querying and iterating on the classes representing the repository for their instances:

Class Query API. The API looks a lot like the standard Smalltalk collection protocol slightly renamed to make it clear these queries *could* potentially be more expensive than just a standard collection.

- `atId:` and `atId:ifAbsent:` allow one to access an instance of the receiver based on its ID. Here is an example of `atId:` use.

```
handleAccountEnable: aRequest
| auction |
(aRequest url endsWith: '/enable-account') ifTrue:
    [ auction := CAAuction atId: (self fieldsAt: #id).
    self session pendingAction: 'User has been enabled!' ->
        [ self session user isAdmin ifTrue:
            [ auction seller
                isEnabled: false;
                commit ] ] ]
```

- `do`: iterates over all the instances of the class but does a copy in case the `do` modifies the collection.
- `find`: returns the first instance satisfying the predicate, as `do`
`find:ifAbsent`: and `find:ifPresent`: `findAll`: returns all the instances that match a predicate. Here is an example of `findAll`:

```
isValidAuctionVin: aVin
  ^ (CAAuction findAll: [ :each | each vin = aVin ])
    allSatisfy: [ :each | each isClosed ]
```

- `findAll` returns all the instances of the class.

Instance API. There's a simple API for the instance side:

- `id` returns a UUID string in base 36 which uniquely identifies the instance.
- `createdOn` and `updatedOn` return the timestamps of the creation and last update of the instance.
- `version` returns the version of the instance. The version is increased for each save. It is useful in critical sections to validate you're working on the version you expect.
- `indexString` returns all instance variable' `asStrings` as a single string for easy searching.

Instance Actions. Here is the list of actions you can perform on a record.

- `save` saves the instance but is not thread safe.
- `critical`: grabs or creates a Monitor for thread safety.
- `commit` is just a `save` in a `critical`: session.
- `commit`: is similar to `commit` but you can pass a block if you have other work you want done while the object is locked.
- `abortChanges` rolls back to the last saved version.
- `delete` deletes the instance.
- `validate` is a hook that subclasses can override to specify validation action and throw exceptions to prevent saves.

Here are some trivial examples of using an `SDActiveRecord`.

```
person := Person find: [ :each | each name = 'Joe' ].
person commit.
person delete.
user := User
  find: [ :each | each email = 'Joe@Schmoe.com' ]
  ifAbsent: [ User named: 'Joe' email: 'Joe@Schmoe.com' ].
```

```
joe := Person atId: anId.
managers := Employee findAll: [ :each | each hasSubordinates ].
```

The framework offers some hooks that you can override on record life cycle events. But pay attention to invoke the superclass methods.

- `onBeforeFirstSave`
- `onAfterFirstSave`
- `onBeforeSave`
- `onAfterSave`
- `onBeforeDelete`
- `onAfterDelete`

There is also a testing method you might find useful: `isNew` answers true prior to the first successful commit.

25.3.2 About Concurrency

Transactions are a nice to have feature, however, they are not a must have feature. Starbucks doesn't use a two phase commit and MySql became the most popular open source database in existence long before they added transactions as a feature.

In SandstoneDB, concurrency is handled by calling either `commit` or `critical:` and it's entirely up to the programmer to put critical sections around the appropriate code. You are working on the same instances of these objects as other threads and you need to be aware of that to deal with concurrency correctly. You can wrap a `commit:` around any chunk of code to ensure you have a write lock for that object like so...

```
auction commit: [
  auction addBid: (Bid
    price: 30 dollars
    user: self session currentUser) ].
```

While `commit:` saves the instance within a critical section, `critical:` lets you decide when to call `save`, in case you want other actions inside the critical section of code to do something more complex than a simple implicit save. When you're working with multiple distributed systems, like a credit card processor, transactions don't really cut it anyway so you might do something like save the record, get the authentication, and if successful, update the record again with the new authentication.

```
auction critical: [
  [ auction
  acceptBid: aBid;
  save;
  authorizeBuyerCC;
  save ] on: Error do: [ :error | auction reopen; save ] ]
```

Here is another example of the use of a use of the `critical:` method.

```
expireOpenAuctions
(CAAuction findAll: [ :each | each needsClosed ]) do:
  [ :each |
  each critical: [
    [ each bids isEmpty: [ each expireAuction ].
    each save.
    each flushNotifications ]
    on: Error
    do:
      [ :error |
      error asDebugEmail.
      each abortNotifications ] ] ]
```

Only subclass `SDActiveRecord` for aggregate roots where you need to be able to query for the object, for all other objects just use ordinary Smalltalk objects. You do not need to make every one of your domain objects into ActiveRecords, choosing your model carefully gives you natural transaction boundaries since the commit of a single ActiveRecord and all ordinary objects contained within is atomic and stored in a single file. There are no real transactions so you cannot atomically commit multiple ActiveRecords.

That's about all there is to using it, there are some more things going on under the hood like crash recovery and startup but if you really want to know how that works, read the code. It is similar to the approaches we presented before. SandstoneDB is available on SqueakSource and is MIT licensed and makes a handy development and prototyping or small application database for Seaside.

The limits or disadvantages of SandstoneDB are that you have to inherit from `SDActiveRecord`, and it goes against clean domain separation. It makes it harder to reuse your domain code. Now you can define `SDActiveRecord` as a trait and use this trait in your domain code without being forced to change your inheritance hierarchy. Another disadvantage is that SandstoneDB is designed for small projects that are satisfied with one single image. SandstoneDB neither provides distributed object access (there is always just one image that accesses the data) nor transactional semantics (two concurrent processes could create fatal conflicts in the data structures, unless the developer uses proper concurrency control himself).

OO purists wouldn't want domain objects to be linked to a persistency framework. You can see this in the design of most OODBs available, it's considered

a sin to make you inherit from a class to obtain persistence. The typical usage pattern is to create a connection to the OODB server which basically presents itself to you as a persistent dictionary of some sort where you put objects into it and then commit any unsaved changes. They will save any object and leave it up to you what your object should look like, intruding as little as possible on your domain, so they say. In Pharo and Squeak, one possible solution explored by SqueakSave (a new framework to save objects in Squeak) is to turn the root class `SDActiveRecord` into a trait (trait are compiled-time group of methods) and to apply the trait to your classes. Behind the scenes there's some voodoo going on where this persistent dictionary tries to figure out what's actually been changed either by having installed some sort of write barrier that marks objects dirty automatically when they get changed, comparing your objects to a cached copy created when they were originally read, or sometimes even explicitly forcing the programmer to manually mark the object dirty. The point of all of this complexity is to minimize writes to the disk to reduce IO and keep things snappy. This is what we will see next with Magma.

25.4 Magma: an Object-Oriented Database

Magma is an open-source object-oriented database developed entirely in Smalltalk. Magma provides transparent access to a large-scale shared persistent object model. It supports multiple users concurrently via optimistic locking. It uses a simple transaction protocol, including nested transactions, supports collaborative program development via live class evolution, peer-to-peer model sharing and Monticello integration. Magma supports large, indexed collections with robust querying, runs with pretty good performance and provides performance tuning mechanisms. Magma is fault tolerant and includes a small suite of tools. Magma can either work locally or on a remote Magma server. This means, multiple images can access the same database concurrently.

Magma provides safe access and management to multiple, large, interconnected models over the network, by multiple users, simultaneously. In keeping with the simplicity of the other frameworks, it “just works” out of the box with comparably little API and learning curve. Developers will appreciate there is no need to inherit from a special superclass and no need to signal changed-notifications anywhere. There are a lot of options but the defaults will work fine without any consideration. Servers running when the image is closed and reopened are automatically restarted, connected clients are automatically reconnected. Recovery from hardware failures also occurs automatically, guaranteeing integrity.

25.4.1 How it works

A running Magma server is never required to deal with actual instances of the domain, it is really nothing more than a buffer server. The persistent model is encapsulated into serialized object buffers that are relationally-linked by their `oid` (object-id). Here is the entire class hierarchy:

```
MaObjectBuffer #('byteArray' 'startPos')
    MaFixedObjectBuffer #()
    MaVariableBuffer #()
        MaByteObjectBuffer #()
            MaStorageObjectBuffer #()
        MaVariableObjectBuffer #()
    MaVariableWordBuffer #()
```

Every single Magma repository consists of many thousands (millions, billions, etc.) of these buffers written to various object files (note that each Magma repository is physically stored in its own directory of files, you should not remove these Magma-generated files via the OS, or add any additional files). This first-class representation of the buffers allows the Magma server to dance over any part of the domain with agility, rapidly supplying requesting clients with object-answers to their requests by way of the “Ma client server” supporting package, the high-performance client-server framework for Squeak.

After receiving the chunk of buffers from the server, Magma clients translate them into the domain instances, constructing the model and attaching it correctly to the existing cached model. Of course the model is rendered exactly true to its original shape, cycles and all. The “edges” of the cached domain model are terminated by Proxy instances which will automatically mutate that portion of the model if the program ventures there.

This constant conversion to and from the `MaObjectBuffer` instance format does mean Magma runs more slowly for a single-user than the all-in-memory frameworks. At least until the model grows beyond 100MB. At that point, every single save in the all-in-memory is writing out another 100MB, but Magma will automatically determine only the objects that changed and write out only those changes. Even further, the level of performance can be maintained indefinitely as the size of the model or number of users increases, by adding additional servers as necessary. Magma constantly monitors and captures extensive performance statistics, allowing developers to fine-tune their programs to maximize use of the available resources.

Magma is a safe place to maintain a Squeak object model important to you or your organization. It supports full-backup and DR replication for rapid failover, out of the box. The generic structure of the storage is simple, well documented, has remained compatible across many Squeak versions and will continue to do so

In Magma you have to specify the root of the world that you want to store and Magma identifies the changes that should be saved. Now we will have a look at how to use Magma to save our ToDo application.

25.4.2 Getting Started

You can install Magma using SqueakSource as indicated from <http://wiki.squeak.org/squeak/2657>. You basically get three packages, the client, the server and the tests. Typically you will load the server package, since the client package contains the code to connect to a remote server and the server package contains the client and server code.

To start we will work with a local repository. Later we describe the simple steps to do to get a remote server. Note that the difference between the two setups is quite small. We will define a small class to group all the operations to set up and manage a connection as well as accessing the session.

Setting up the Database. The first action is to set up the database using the message `#create:root:`. We have to give a file location and a root of the object graph we want to store in this location.

```
MagmaRepositoryController create: 'todo' root: ToDoList new.
```

Here the controller will create a repository in the path{/tmp/todo} folder and it takes the singleton of the `ToDoList` class since it contains all the items of our application. It is often the case that you specify a dictionary where you can define multiple roots of your application. We will use such code in the class that will manage the storage of our application.

```
Object subclass: #ToDoDB
  instanceVariableNames: 'session'
  classVariableNames: 'Default'
  poolDictionaries: ''
  category: 'ToDo-Model'
```

We define the class method `path` which returns the file location of our repository as well as the method `createDB`. Then we create the repository by executing `ToDoDB createDB`.

```
ToDoDB class>>path
  ^ 'todo'
```

```
ToDoDB class>>createDB
  MagmaRepositoryController
  create: self path
  root: ToDoList new.
```

We define an accessor method for the session and we define a `MagmaSession` using the path specified before and connect to the session using the method `connectAs:`. Note here that we send the message `MagmaSession class>>openLocal:` to the session since we are in local mode.

```
ToDoDB>>session
^ session

ToDoDB>>connect
    session := (MagmaSession openLocal: self class path)
        connectAs: 'user'
```

We make sure that the session will be correctly set in the singleton method by sending the instance message `connect` we previously defined.

```
ToDoDB class>>uniqueInstance
^ Default ifNil: [ Default := self new connect; yourself ]
```

We also define the method `release` which disconnects and closes the session. Note that the method `MagmaSession>>disconnectAndClose` deals with the fact that we are in local or remote mode.

```
ToDoDB class>>release
Default isNil ifFalse: [
    Default session disconnectAndClose.
    Default := nil ]
```

The following `root` method illustrates how we could access the session root object.

```
ToDoDB class>>root
^ self uniqueInstance session root
```

We finally provide a class method to commit changes to the database using the method `MagmaSession>>commit:`.

```
ToDoDB class>>commit: aBlock
self uniqueInstance session commit: aBlock
```

Now we are ready to test. We will create a repository, add an item to the list and commit the changes and release the connection.

```
ToDoDB createDB.
ToDoDB commit: [ ToDoDB root add: ToDoItem new ].
ToDoDB release.
```

Now using these messages we can decide when we will store the data. For example we modify the method `ToDoListView>>edit:` and wrap the change of the current item into a `commit:` so that the resulting objects get stored.

```

ToDoListView>>edit: anItem
| result |
result := self call: (ToDoItemView new model: anItem copy).
result isNil ifFalse: [
    ToDoDB commit: [
        self model items replaceAll: anItem with: result ] ]

```

Magma will detect the changes from the root object and save them if we wrap the action inside `commit:`.

Note that the Magma tutorial available at <http://wiki.squeak.org/squeak/2689> proposes ways to avoid modifying the domain objects to make them persistent.

25.4.3 Running Remotely

If you want to use Magma in remote server mode you have to execute the following piece of code in a second image. This code launches the server. You have to specify where the repository will be located and the port used for the connection.

```

MagmaServerConsole new
    open: 'todo';
    processOn: 51001;
    inspect

```

Do not close the inspector since you will use it to send the message `shutdown` to stop the server. If you accidentally close the inspect window you can execute something like this:

```

MagmaServerConsole allInstancesDo:
    [:aServerConsole| aServerConsole shutdown]

```

To connect to the server, we will have to specify the port and possibly the address. The following methods show how to define a connection to the port 51001 on localhost.

```

ToDoDB>>connect
    session := MagmaSession
        hostAddress: self localhost
        port: self defaultPort.
    session connectAs: 'user'

```

```

ToDoDB>>localhost
    ^ #(127 0 0 1) asByteArray

```

```

ToDoDB>>defaultPort
    ^ 51001

```

We just showed superficially the functionality offered by Magma. Magma offers much more such as optimized and large collections. We suggest you read the documentation of Magma that you can find at: <http://wiki.squeak.org/squeak/2665>.

25.5 GLORP: an Object-Relational Mapper

GLORP which stands for Generic Lightweight Object-Relational Persistence, is a meta-data driven object-relational mapper. GLORP is open-source (LGPL) and you can find more information at <http://www.glorp.org/>. GLORP is non-intrusive in the sense that the object model does not have to store foreign keys in objects and that you can map an arbitrary domain model to an arbitrary relational model.

GLORP allows you to manipulate databases with a low-level interface. You can create databases, create sessions, add/remove rows, select a row and so on. However this interface is low-level and GLORP offers a high-level interface. When you are using the higher-level API, GLORP watches for dirty objects (i.e., objects whose state has been changed since the transaction began), and then automatically writes those objects to the RDBMS when you commit the transaction. The required SQL is automatically generated and you don't need to ever explicitly write any SQL yourself, or otherwise need to explicitly manipulate the rows yourself. In addition GLORP preserves object identity when objects are fetched several times from the database. GLORP uses a clever object cache.

GLORP uses meta-data to define the mapping between the objects and the relational database. The meta-data is a declarative description of the correspondence between an object and its database representation. It is used to drive reads, writes, joins, and anything else that is needed. SQL code should not ever have to be explicitly created, as it is autogenerated through the mappings (i.e., through the meta-data). In GLORP queries are expressed in terms of objects and composable expressions.

We will not cover GLORP in this book. We suggest reading the book *An Introduction to Seaside* by Michael Perscheid et al. as well as the GLORP documentation.

Chapter 26

Magritte: Meta-data at Work

Many applications consist of a large number of input dialogs and reports that need to be built, displayed and validated manually. Often these dialogs remain static after the development phase and cannot be changed unless a new development effort occurs. For certain kinds of application domains such as small businesses, changing business plans, modifying workflows, etc., it usually boils down to minor modifications to domain objects and behavior, for example new fields have to be added, configured differently, rearranged or removed. Performing such tasks is tedious.

Magritte is a meta-data description framework. With Magritte you describe your domain objects and among other things you can get Seaside components and their associated validation for free. Moreover Magritte is self-described, enabling the automatic generation of meta-editors which can be adapted for building end-user customizations of application.

In this chapter we describe Magritte, its design and how to customize it. Now be warned, Magritte is a framework supporting meta-data description. As any framework, mastering it takes time. It is not a scaffolding engine, therefore Magritte may imply some extra complexity and you have to understand when you want to pay for such complexity.

26.1 Basic Principles

In this section we present the key principles. With such knowledge you can get 80% of the power of Magritte without knowing all its possible customiza-

tions. The key idea behind Magritte is the following: given one object with a set of values, and a description of this information, we will create automatically tools that treat such information and for example automatically create Seaside components. Figure 26.1 shows that a person address', John's address, instance of the class Address, is described by a description object which is attached to the class Address. A program (i.e., database query, generic UI, seaside component builder) will interpret the value of the instance by using its descriptions.

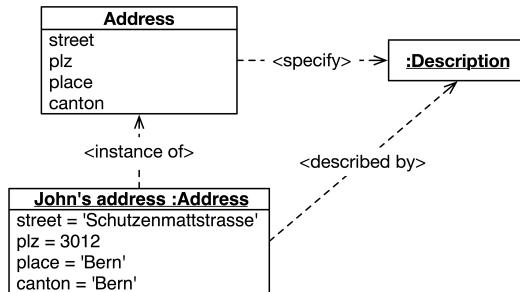


Figure 26.1: An object is described by a description which is defined on its class.

Here are the basic description assumptions:

- An object is described by adding methods named `description` (naming convention) to the class-side of its class. Such description methods create different description entities. The following `Address` class method creates a string description object that has a label 'Street', a priority and two accessors `street` and `street:` to access it.

```

Address class>>descriptionStreet
^ MASTringDescription new
  accessor: #street;
  label: 'Street';
  priority: 100;
  yourself
  
```

Note that there is no need to have a one to one mapping between the instance variables of the class and the associated descriptions.

- All descriptions are automatically collected and put into a container `description` when sending `description` to the object (see Figure 26.2).
- Descriptions are defined programmatically and can also be queried. They support the collection protocol (`do:, select:...`).

Obtaining a component. Once an object is described, you can obtain a Seaside component by sending to the object the message

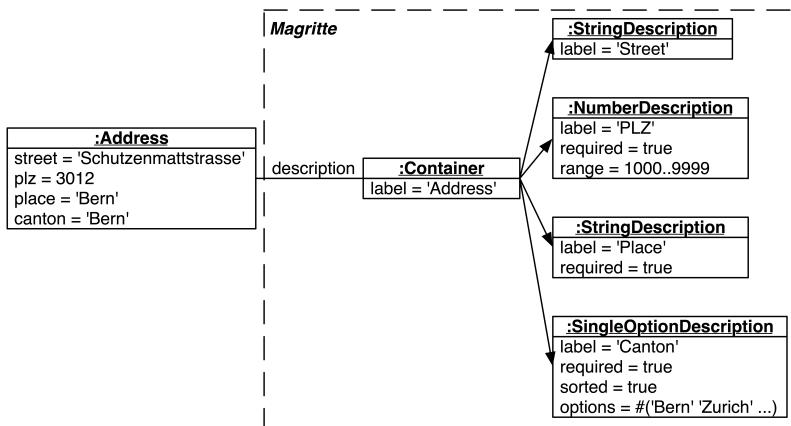


Figure 26.2: Describing an Address.

`Object>>asComponent`. For example to get a component for an address:

```
anAddress asComponent.
```

It is often useful to decorate the component with buttons like cancel and Ok. This can be done by sending the message `WAComponent>>addValidatedForm` to the component. Note that you can also change the label of the validating form using `addValidatedForm:` and passing an array of associations whose first element is the message to be sent and the second the label to be displayed.

A Magritte form is generally wrapped with a form decoration via `WAComponent>>addValidatedForm`. Magritte forms don't directly work on your domain objects. They work on a memento of the values pulled from your object using the descriptions. When you call `save`, the values are validated using the descriptions, and only after passing all validation rules are the values committed to your domain object by the momentos via the accessors.

```
anAddress asComponent addValidatedForm.
```

```
anAddress asComponent addValidatedForm: { #save -> 'Save'. #cancel -> 'Cancel' }.
```

A description container is an object implementing collection behavior (`collect:`, `select:`, `do:`, `allSatisfy:`, ...). Therefore you can send the normal collection messages to extract the parts of the descriptions you want. You can also iterate over the description or concatenate new ones. Have a look at the `MAContainer` protocol.

You can also use the message `MAContainer>>asComponentOn:` with `aModel` to build or select part of a description and get a component on a given

model. The following code schematically shows the idea: two descriptions are assembled and a component based on these two descriptions is built.

```
((Address descriptionStreet , Address descriptionPlace)
  asComponentOn: anAddress) addValidatedForm
```

26.2 First Example

Let us go over a simple but complete example. We want to develop an application to manage person, address and phone number as shown in Figure 26.3.

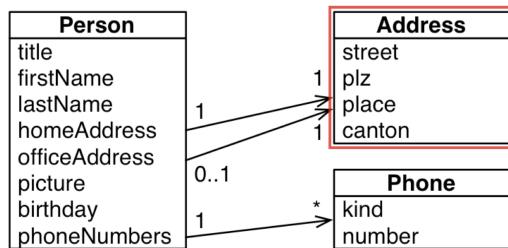


Figure 26.3: Our address.

We define a class `Address` with four instance variables and their corresponding accessors.

```
Object subclass: #Address
  instanceVariableNames: 'street place plz canton'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'MaAddress'
```

```
Address class>>example1
| address |
address := self new.
address plz: 1001.
address street: 'Sesame'.
address place: 'DreamTown'.
address canton: 'Bern'.
^ address
```

Then we add the descriptions to the `Address` class as follows: the street name and the place are described by a string description, the PLZ is a number with a range between 1000 and 9999, and since the canton is one of the predefined canton list (our address is for Switzerland so far), we describe it as a single option description.

```
Address class>>descriptionStreet
^ MAStringDescription new
  accessor: #street;
  label: 'Street';
  priority: 100;
  yourself
```

```
Address class>>descriptionPlz
^ MANumberDescription new
  accessor: #plz;
  label: 'PLZ';
  priority: 200;
  beRequired;
  min: 1000 ;
  max: 9999;
  yourself
```

```
Address class>>descriptionPlace
^ MAStringDescription new
  accessor: #place;
  label: 'Place';
  priority: 300;
  yourself
```

```
Address class>>descriptionCanton
^ MASingleOptionDescription new
  accessor: #canton;
  label: 'Canton' ;
  priority: 400;
  options: #('Bern' 'Solothurn' 'Aargau' 'Zuerich' 'Schwyz' 'Glarus');
  beSorted;
  beRequired;
  yourself
```

Now we can start manipulating the descriptions. Inspect the description object of the address object:

```
| address |
address := Address example1.
address description inspect.
```

Now we can iterate over the descriptions and get the values associated with the descriptions of our address model:

```
| address |
address := Address example1.
address description do: [ :description |
  Transcript
    show: description label; show: ':'; tab;
    show: (description toString: (address readUsing: description));
    cr ]
```

Executing the second code snippet outputs the following in the Transcript:

```
Street: Sesame
PLZ: 1001
Place: DreamTown
Canton: Bern
```

Creating a Seaside Editor. Now we are ready to create a Seaside component automatically in a similar manner.

```
WAComponent subclass: #MyContactAddress
  instanceVariableNames: 'addressForm'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'MaAddress'
```

```
MyContactAddress>>initialize
  super initialize.
  addressForm := Address example1 asComponent
```

```
MyContactAddress>>children
  ^ Array with: addressForm
```

The method `asComponent` sent to the address object automatically builds a Seaside component for us. The resulting editor is displayed in Figure 26.4.

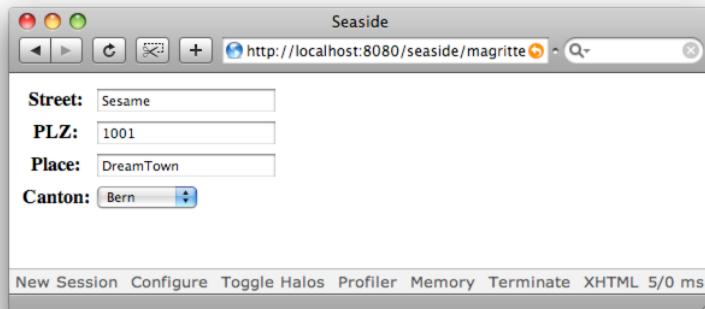


Figure 26.4: Address example1 asComponent.

To enable validation and add buttons to save and cancel the editing process is a matter of adding a decoration. The message `addValidatedForm` decorates the component with the necessary rendering and behavior.

```
MyContactAddress>>initialize
  super initialize.
  addressForm := Address example1 asComponent.
  addressForm addValidatedForm
```

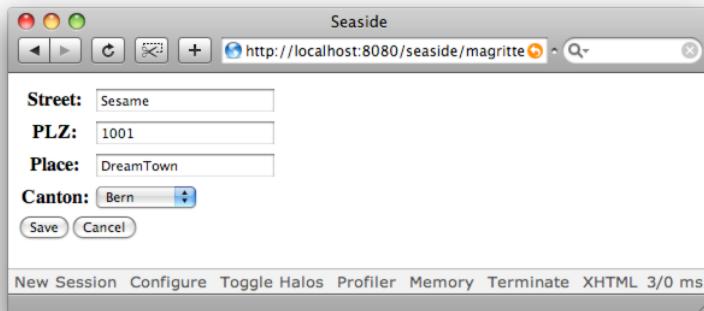


Figure 26.5: Same Magritte generated component with buttons and validation.

As a result we get a complete address editor, as seen in Figure 26.5.

In summary Magritte is really easy to use with Seaside. You put your descriptions on the class-side according to a naming-convention. You can then ask your model objects for their descriptions by sending the message `description` or alternatively you directly ask Magritte to build a Seaside editor for you by sending the message `asComponent`.

26.3 Descriptions

Descriptions, as we have seen in the above examples, are naturally organized in a description hierarchy. A class diagram of the most important descriptions is shown in Figure 26.6. Different kinds of descriptions exist: simple type-descriptions that directly map to Smalltalk classes, and some more advanced descriptions that are used to represent a collection of descriptions, or to model relationships between different entities.

Descriptions are central to Magritte and are connected to accessors and conditions that we present below.

1. **Type Descriptions.** Most descriptions belong to this group, such as the `ColorDescription`, the `DateDescription`, the `NumberDescription`, the `StringDescription`, the `BooleanDescription`, etc. All of them describe a specific Smalltalk class; in the examples given, this would be `Color`, `Date`, `Number` and all its subclasses, `String`, and `Boolean` and its two subclasses `True` and `False`. All descriptions know how to perform basic tasks on those types, such as to display, to parse, to serialize, to query, to edit, and to validate them.

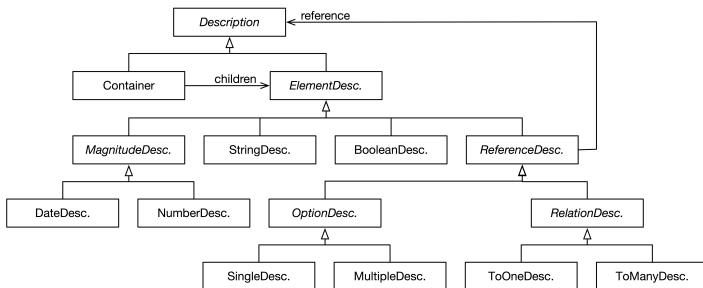


Figure 26.6: Description hierarchy.

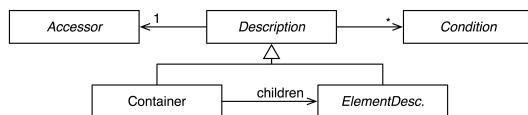


Figure 26.7: Descriptions are a composite and connected via accessors.

2. **Container Descriptions.** If a model object is described, it is often necessary to keep a set of other descriptions within a collection, for example the description of a person consists of a description of the title, the family name, the birthday, etc. The `ContainerDescription` class and its subclasses provide a collection container for other descriptions. In fact the container implements the whole collection interface, so that users can easily iterate (`do:`), filter (`select:, reject:`), transform (`collect:`) and query (`detect:, anySatisfy:, allSatisfy:`) the containing descriptions.
3. **Option Descriptions.** The `SingleOptionDescription` describes an entity, for which it is possible to choose up to one item from a list of objects. The `MultipleOptionDescription` describes a collection, for which it is possible to choose any number of items from a predefined list of objects. The selected items are described by the referencing description.
4. **Relationship Descriptions.** Probably the most advanced descriptions are the ones that describe relationships between objects. The `ToOneRelationshipDescription` models a one-to-one relationship; the `ToManyRelationshipDescription` models a one-to-many relationship using a Smalltalk collection. In fact, these two descriptions can also be seen as basic type descriptions, since the `ToOneRelationshipDescription` describes a generic object reference and the `ToManyRelationshipDescription` describes a collection of object references.

26.4 Exceptions

Since actions on the meta-model can fail. In addition, objects might not match a given meta-model. Finally it is often important to present to the end users readable error messages. Magritte introduces an exception hierarchy which knows about the description, the failure and a human-readable error message as shown in Figure 26.8.

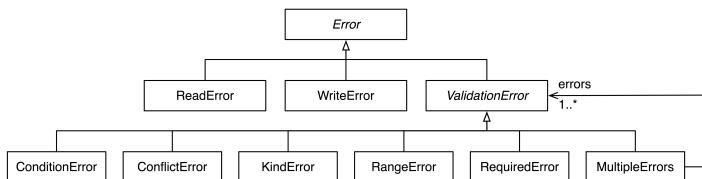


Figure 26.8: Exceptions.

26.5 Adding Validation

Descriptions can have predicates associated with them to validate the input data. Magritte calls such validation conditions associated with a description. A condition is simply a block that returns true or false. A simple one is the single field condition which is passed the pending value directly and can be attached directly to your descriptions. It is as simple as the following block which checks the length of the value once the blanks are removed.

```
aDescription addCondition: [ :value | value withBlanksTrimmed size > 3 ]
```

There are advantages to having your rules outside your domain objects, especially if you're taking advantage of Magritte as an Adaptive Object Model where users can build their own rules. It also allows the mementos to easily test data for validity outside the domain object and gives you a nice place to hook into the validation system in the components. Still you have to pay attention since it may weaken your domain model.

Multiple Field Validation. When we did the Mini-reservation example in previous chapters we wanted the starting date to be later than today. With Magritte we can express this point as follows:

```
descriptionStartDate
  ^ MADateDescription new
  accessor: #startDate;
  label: 'Start Date';
  addCondition: [ :value | value > Date today ];
  beRequired;
  yourself
```

```

descriptionEndDate
  ^ MADateDescription new
  accessor: #endDate;
  label: 'End Date';
  addCondition: [ :value | value > Date today ];
  beRequired;
  yourself

```

Now to add a validation criterion that depends on several field values, you have to do a bit more than that because you cannot from one description access the other. Suppose that we want to enforce that the endDate should be after the startDate. We cannot add that rule to the endDate description because we cannot reference the startDate value. We need to add the rule in a place where we can ensure all single field data exists but before it's written to the object from the mementos.

We need to add a rule to the objects container description like this

```

descriptionContainer
  ^ super descriptionContainer
  addCondition: [ :object |
    (object readUsing: self descriptionEndDate) >
      (object readUsing: self descriptionStartDate)]
  labelled: 'End date must be after start date';
  yourself

```

This simply intercepts the container after it is built, and adds a multi field validation by accessing the potential value of those fields. You get the memento for the object, and all the field values are in its cache, keyed by their descriptions. You simply read the values and validate them before they have a chance to be written to the real business object. Multi field validations are a bit more complicated than single field validations but this pattern works well.

26.6 Accessors and Mementos

Accessors. In Smalltalk data can be accessed and stored in different ways. Most common data is stored within instance variables and read and written using accessor methods, but sometimes developers choose other strategies, for example to group data within a referenced object, to keep their data stored within a dictionary, or to calculate it dynamically from block closures. Magritte uses a strategy pattern to be able to access the data through a common interface, see Figure 26.9.

By far the most commonly used accessor type is the `SelectorAccessor`. It can be instantiated with two selectors: a zero argument selector to read, and a one

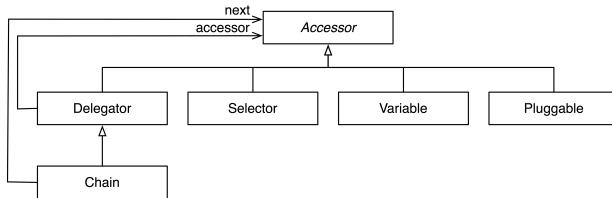


Figure 26.9: Accessors in Magritte.

argument selector to write. For convenience it is possible to specify a read selector only, from which the write selector is inferred automatically.

The `MADictionaryAccessor` is used to add and retrieve data from a dictionary with a given key. This access strategy is mainly used for prototyping as it allows one to treat dictionaries like objects with object-based instance variables.

When a memento writes valid data to its model, it does so through accessors which are also defined by the descriptions. `MAAccessor` subclasses allow you to define how a value gets written to your class.

Mementos. Magritte introduces mementos that behave like the original model and that delay modifications until they are proven to be valid. When components read and write to domain objects, they do it using mementos rather than working directly on the objects. The default memento is `MACheckedMemento`. The mementos give Magritte a place to store invalid form data prior to allowing the data to be committed to the domain object. It also allows Magritte to detect concurrency conflicts. By never committing invalid data to domain objects, there's never a need to roll anything back. So mementos are good since editing might turn a model (temporarily) invalid, canceling an edit shouldn't change the model and concurrent edits of the same model should be detected and (manually) merged. `mementoClass` is a class factory that you can specialize to specify your own memento. But normally you should not need that.

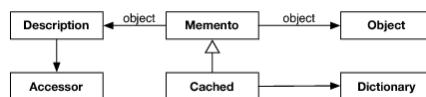


Figure 26.10: Mementos (simplified version).

26.7 Custom Views

Components control how your objects display. Some descriptions have an obvious one to one relationship with a UI component while others could easily be shown by several different components. Figure 26.11 shows some components. For example, an `MAMemoDescription` would be represented by a text area, but a `MABooleanDescription` could be a checkbox, or a drop down with true and false, or a radio group with true and false.

Each description defaults to a component, but allows you to override and specify any component you choose, including any custom one you may write using the message `Description>>componentClass::`.

```
aDescription componentClass: aClass
```

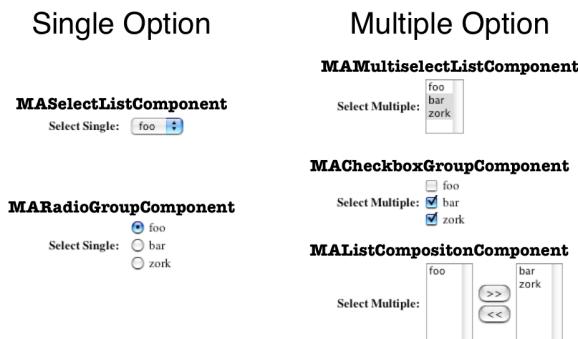


Figure 26.11: Some Magritte specific widgets.

You can also define your own view by following the steps:

- Create a subclass of `MADescriptionComponent`.
- Override `renderEditorOn:` and/or `renderViewerOn:` as necessary.
- Use your custom view together with your description by using the accessor `componentClass::`.
- Possibly add your custom view to its description into `defaultComponentClasses`.

26.8 Custom Descriptions

In some cases it might happen that there is no description provided to use with a class. If your domain manipulates money (amount and currency) or

URLs (scheme, domain, port, path, parameters) you may want to define your own descriptions (or load an extension package that already provides these descriptions) to take advantage of Magritte.

Extending Magritte is simple, create your own description but remember that Magritte is described within itself so you have to provide certain information.

- Create a subclass of `MAElementDescription`.
- On the class-side override: `isAbstract` to return `false`, `label` to return the name of the description. On the instance-side override: `kind` to return the base-class, `acceptMagritte:` to enable visiting and `validateSpecific:` to validate.
- Create a view, if you want to use it for UI building.

We suggest you have a look at existing descriptions. In addition, carefully choosing the right superclass can already provide you part of what you are looking for. Parsing, printing and (de)serialization is implemented by the following visitors: `MAStringReader`, `MAStringWriter`, `MABinaryReader` and `MABinaryWriter`.

26.9 Summary

While Magritte is really powerful, using a meta-data system will add another indirection layer to your application, so this is important to understand how to use such power and when to just use normal development techniques. A good and small example to learn from is the Conrad conference management system developed to manage the ESUG conference available at <http://www.squeaksource.com/Conrad.html>. Here Magritte is used to describe the different forms and all the data. The Pier content management system is a more complex example of using Magritte, it can be downloaded at <http://www.piercms.com>.

Index

AJAX, 255
anchors, 111
Announcement, 173
Avi Bryant, 1, 4

callback:
 callback:, 127, 135
Cmsbox, 5
Comet, 255
componentClass:
 componentClass:, 388
consumes:, 350
Contact, 128, 135
convenience methods, 127
copy, 348

delete, 348
download, 137

Form, 224
form:
 form:, 136
forms, 123
 buttons, 125–127
 check~boxes, 133, 134
 date~input, 135, 137
 drop-down menus, 128
 list boxes, 128
 lists, 131
 radio~buttons, 131, 132
 text areas, 123
 text input fields, 123

get, 346, 348

halo, 100

JavaScript, 255
 JQueryClass, 296
 all, 297

expression:, 296
html, 297
id, 296
new, 297
ready:, 297
this, 297

 JQueryInstance, 296
 children, 298
 children:, 298
 closest, 299
 closest:, 299
 contents, 298
 find:, 298
 next, 298
 next:, 298
 nextAll, 298
 nextAll:, 298
 nextUntil:, 298
 parent, 299
 parent:, 299
 parents, 299
 parents:, 299
 parentsUntil:, 299
 previous, 298
 previous:, 298
 previousAll, 298
 previousAll:, 298
 previousUntil:, 298
 siblings, 297
 siblings:, 297

Julian Fitzell, 1, 3, 4

lists
 ordered, 96, 98
Lukas Renggli, 4

MACContainer
 asComponentOn:, 379
Microformats, 255

move, 348
Object
 asComponent, 379
 asString, 89
 displayString, 90
 printOn:, 90
 renderOn:, 90
on:of:
 ==on:of:==, 127
 on:of:, 126, 127, 134
Philippe Marshall, 4
Pier, 5
post, 348
produces:, 349
Prototype, 264
PTAjax, 278
 callback:value:, 281
 onComplete:, 282
 onFailure:, 282
 onSuccess:, 282
 triggerForm:, 276, 281
 triggerFormElement:, 281
PTElement, 267, 276
 down, 276
 id:, 276
 next, 276
 previous, 276
 up, 276
PTEvaluator, 267, 280
PTEvent, 267
PTFactory
 autocomplete, 267
 draggable, 267
 droppable, 267
 effect, 267
 element, 267
 evaluator, 267, 281
 event, 267
 form, 267
 formElement, 267
 inPlaceCollectionEditor, 267
 inPlaceEditor, 267
 insertion, 267
 periodical, 267, 281
 request, 267, 281
 responders, 267
 selector, 267
 slider, 267
 sortable, 267
 sound, 267
 updater, 267, 281
PTForm, 267
PTFormElement, 267
PTInsertion, 267
PTPeriodical, 267, 280
PTRequest, 267, 278
PTResponders, 267, 282
PTSelector, 267
PTUpdater, 267, 278
put, 348
RemoveChild, 174
rendering
 lists, 96, 98
 tables, 96, 98
 text, 81
request context, 347
resourceUrl:
 resourceUrl:, 224
response, 347
REST, 343
 COPY, 348
 DELETE, 344, 348
 GET, 343, 348
 MOVE, 348
 POST, 344, 348
 PUT, 344, 348
RRComponent, 258
RRRssRenderCanvas
 author, 261
 category, 260, 261
 comments, 261
 copyright, 260
 description, 260, 261
 enclosure, 261
 generator, 260
 guid, 261
 language, 260
 lastBuildDate, 260
 link, 260, 261
 managingEditor, 260
 pubDate, 260, 261
 source, 261
 title, 260, 261
 webMaster, 260
RSS, 255, 257

script.aculo.us, 264
SqueakSource, 6
style sheets, 99
SUAccordion, 281
SUAutocompleter, 267, 287
SUAutocompleterTest, 287
SUDraggable, 267
SUDroppable, 267
SUEffect, 267
 pulse, 271
 switchOff, 271
 toggleAppear, 273
 toggleBlind, 273
 toggleSlide, 273
SUInPlaceCollectionEditor, 267, 286
SUInPlaceEditor, 267, 286
 callback:, 285
 cancelControl:, 285
 cancelText:, 286
 highlightColor:, 286
 okControl:, 286
 okText:, 286
 rows:, 286
 submitOnBlur:, 286
 triggerInPlaceEditor:, 285
SUInPlaceEditorTest, 287
SUSlider, 267, 287
SUSliderTest, 287
SUSortable, 267
 constraint:, 283
 ghosting, 284
 handle:, 284
 onUpdate:, 283
 tag:, 284
SUSortableDoubleTest, 284
SUSortableTest, 284
SUSound, 267
SUTabPanel, 281

tables, 98

upload, 137
urlOf:
 urlOf:, 231, 232

value:
 value:, 127, 135

WAAuthorTag, 113
 callback:, 113
 resourceUrl:, 321
 url:, 112, 113
 with:, 113
WAAnswerHandler, 168
WAAuthConfiguration, 320
WACheckboxTag, 134, 135
 callback: aBlock, 135
 on: aSymbol of: anObject,
 135
 onTrue: aBlock onFalse: aBlock,
 135
 value: aBoolean, 135
WAComponent, 81, 151
 addDecoration:, 168
 addValidatedForm, 379
 answer, 145
 answer:, 205, 208
 call:, 147
 children, 206
 chooseFrom:, 151
 chooseFrom:caption:, 151
 chooseFrom:default:, 151
 chooseFrom:default:caption:,
 152
 confirm:, 119, 120, 151
 inform:, 149, 151
 initialRequest:, 245, 250
 onAnswer:, 147, 153, 166
 renderContentOn:, 44, 81, 82
 request:, 118, 151
 request:default:, 151
 request:label:, 151
 request:label:default:, 151
 session, 241, 244
 show:, 153
 states, 152, 221
 style, 99, 105
 updateRoot:, 226, 233, 261
 updateUrl:, 246, 249
WACurrentSession, 242
WADateInput, 135, 137
 callback: aBlock, 137
 options: anArray, 137
 with: aDate, 137
WADecoration
 decoratedComponent, 172
WADelegation, 168

WADevelopmentConfiguration,
 320
WAFile, 138
WAFileLibrary, 228, 231
 addAllFilesIn:, 228
 addFileAt:, 228
WAFormDecoration, 168, 170, 171
WAFormDialog, 151, 171, 178
WAFormTag, 124
 with:, 124
WAGenericTag
 class:, 105
 class;if:, 106
WAHeadingTag, 92
WAHtmlCanvas
 anchor, 112, 113
 listItem:, 97
 orderedList, 97
 orderedList:, 97
 render:, 90
 tableHeading:, 98
 text:, 89
WAHtmlRoot, 226
 addScript:, 226
 addStyles:, 226
 bodyAttributes, 226
 headAttributes, 226
 javascript, 226
 meta, 226
 stylesheet, 226
 title, 226
WAIImageTag
 document:, 225
 form:, 224, 225
 resourceUrl:, 224
 url:, 223
WAInputDialog, 178
WAListTag, 97
WAMessageDecoration, 168
WARadioButtonTag, 131, 132
 callback:, 132
 callback: aBlock, 133
 group: aRadioGroup, 133
 selected:, 132
 selected: aBoolean, 133
WARenderCanvas, 81, 266
 cancelButton, 205
 checkbox, 134
 form, 124
radioButton, 131, 132
radioGroup, 132
submitButton, 125, 203, 205
textInput, 124, 125
WARequest, 250, 354
WARequestContext, 242
WARequestHandler, 341
 handleRequest:, 341
WAResponse, 355
WARestfulFilter, 353
WARestfulHandler, 345
WAScreenshot, 225
WASelectTag, 129, 131
 callback: aBlock, 131
 list: aCollection, 131
 on: aSymbol of: anObject,
 131
 selected: anObject, 131
 size: anInteger, 131
WASession, 243
 expire, 251
 unregistered, 246, 247
WASubmitButton, 126
 callback: aBlock, 126
 on: aSymbol of: anObject,
 126
 value: aString, 126
WASubmitButtonTag, 124
WATagBrush, 269
 onBlur:, 269
 onChange:, 269
 onClick:, 269
 onDoubleClick:, 269
 onFocus:, 269
 onKeyDown:, 269
 onKeyPress:, 269
 onKeyUp:, 269
 onLoad:, 269
 onMouseDown:, 269
 onMouseMove:, 269
 onMouseOut:, 269
 onMouseOver:, 269
 onMouseUp:, 269
 onReset:, 269
 onSelect:, 269
 onSubmit:, 269
 onUnload:, 269
WATask, 177, 180
 go, 177

WATextInputTag, 124, 125
 callback: aBlock, 126
 on: aSymbol of: anObject,
 126
 value: aString, 126
WAToolDecoration, 320
WAUrl, 249
 addField:value:, 250
 addToPath:, 250
 fragment:, 250
WAValidationDecoration, 168, 171
WAVersionUploader, 325
WAWindowDecoration, 168, 169
WebSudoku, 213

XHTML, 255

Yesplan, 5

Seaside is the open source framework of choice for developing sophisticated and dynamic web applications. Seaside uses the power of objects to master the web. With Seaside web applications is as simple as building desktop applications. Seaside lets you build highly dynamic and interactive web applications.

Seaside supports agile development through interactive debugging and unit testing. Seaside is based on Smalltalk, a proven and robust language implemented by different vendors. Seaside is now available for all the major Smalltalk including Pharo, Squeak, GNU Smalltalk, Cincom Smalltalk, GemStone Smalltalk, and VA Smalltalk.

Dynamic Web Development with Seaside, intended for developers, will present the core of Seaside as well as advanced features such as Web 2.0 support and deployment. In this book you will learn how to design your own components and glue them together to build and deploy powerful and reusable web applications.

Dynamic Web Development with Seaside is endorsed by ESUG, the European Smalltalk User Group. To learn more about Smalltalk and ESUG, see www.esug.org. This book is sponsored by Inceptive.be, Cincom Systems, GemStone Systems, and Instantiations.



INCEPTIVE.BE
INNOVATIVE SOFTWARE CONCEPTS



Cincom®



ISBN 978-3-9523341-1-9



90000

9 783952 334119



www.esug.org