

[Back](#)

Overview

What you are going to learn

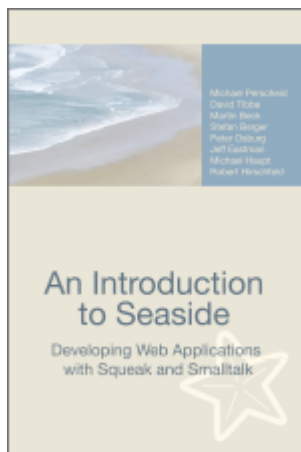
- [Welcome](#)
- [An Introduction to Seaside](#)
- [Acknowledgments](#)
- [Table of Contents](#)

Welcome

Welcome to the Seaside Tutorial by the [Software Architecture Group \(Hasso-Platter-Institut\)](#).

Primarily, the tutorial is intended for students attending the 3rd term course "Software Engineering I" to get started with their Seaside projects. Please feel free to use it for your own projects. The tutorial has 12 chapters (for the table of contents, see below) and introduces a sample application to explain the use of the main features of Seaside. For feedback, please send an email to the authors (see "About us").

An Introduction to Seaside



Book Cover

The tutorial and additional content is also available in printed form. You can order the book via Lulu: <http://www.lulu.com/content/2234565>

[BibTeX Entry](#)

Book description

Seaside is a Web development framework implemented in Smalltalk. It allows the easy creation of powerful Web applications using high level abstractions on the application components and on the underlying hypertext transfer protocol. In doing so, it builds upon the strengths of the Smalltalk object-oriented programming language and transcends many of the common practices needed in other, less dynamic languages.

This book explains the major concepts of Seaside in a clear and intuitive style. A working example of a ToDo List application is developed to illustrate the framework's important concepts that build upon each other in an orderly progression. Besides the notions of users, tasks, components, forms and deployment, additional topics such as persistence, Ajax and Magritte are also discussed.

Acknowledgments

First and foremost, we wish to thank the developer team of the Seaside and script.aculo.us frameworks, with Avi Bryant, Julian Fitzell, and Lukas Renggli leading the way.

Furthermore, we are grateful to all the people who have helped and still help us in improving the quality and contents of our tutorial. The following people have given us extensive feedback:

- Jens Lincke
- Richard Munn
- Klaus D. Witzel
- Keith Hodges
- Kevin Lacobie
- Gilad Bracha

We also thank the many HPI students and everybody else who has given us hints on how to improve the tutorial.

We would like to thank three people especially: Norman Holz, for writing the section about session handling in the fine print chapter; Tobias Pape, for his constant support with LaTeX (book only); and Robert Krahn, for a thorough code review and refactoring.

Table of Contents

■ Overview

- Welcome
- An Introduction to Seaside
- Acknowledgments
- Table of Contents

■ 1 - Introduction

- Web Application Development
- Seaside
- Tutorial Contents
- Requirements
- ToDo Application
- Tutorial Source Code
- Summary

■ 2 - First Steps

- Using the Right Image
- Installation
- Creating the First Component
- How to Configure Your Application
- How to Use Halos
- How to Debug
- Summary

■ 3 - ToDo Application

- Model Overview
- User Model
- Task Model
- Summary

■ 4 - Components

- Think in Components
- Creating Our ToDo Components
- Reuse
- Gluing Components Together
- Summary

■ 5 - Forms

- Form Elements
- Calling Components
- Summary

■ 6 - Tasks and Sessions

- Create Your Own Session

- User Login as a Task
- Login Component
- Register Component
- Message Component
- Summary

■ 7 - External Resources

- Introduction
- The Simple Way of Life
- WAFFileLibrary
- StToDoLibrary
- External Directory
- Summary

■ 8 - Persistence

- Introduction
- Saving All Data in the Image
- Saving All Data in a Relational Database: Glorp
- Saving All Data in an Object-Oriented Database: GOODS
- Saving All Data in an Object-Oriented Database: Magma
- One Database Connection per Session
- Summary

■ 9 - Ajax

- Future of the Web and Ajax
- script.aculo.us
- Installation
- Updater
- InPlaceEditor
- Lightbox
- Summary

■ 10 - Magritte

- What is Magritte?
- Describe StTask
- Create a Magritte Task
- Dynamic Descriptions
- Create a Report
- Summary

■ 11 - The Fine Print

- Introduction
- The URL with _s and _k
- Request/Response Processing
- Session Handling
- Rendering Tree
- Continuations
- Canvas
- Summary

■ 12 - Last But Not Least

- Advanced Seaside
- Configurations
- Decorations
- Static URLs

- Seaside Unit Tests
- Coding Conventions
- Links
- Conclusion

■ **About Us**

- Authors
- Feedback
- Software Architecture Group
- Copyright
- Imprint

[Back](#)

1 - Introduction

What you are going to learn

- [Web Application Development](#)
- [Seaside](#)
- [Tutorial Contents](#)
- [Requirements](#)
- [ToDo Application](#)
- [Tutorial Source Code](#)
- [Summary](#)

Web Application Development

Today it is impossible to imagine the Internet not to exist. More and more businesses rely on the Internet to make their products and services available to their customers. One can even say the Internet is now the most important business platform today. The World Wide Web (WWW) is the vehicle of this transformation. Invented in 1989, the main purpose at the time was to connect static documents to each other. As the Web has become more dynamic this is no longer the case and the demands on developers who write applications for the Web are now higher than at any time in history. One of the first ideas to change the Web landscape was the Common Gateway Interface (CGI) which was invented in 1993. It served to exchange data between the Web server and external software which could then produce pages tailored to individual customer interactions. With the introduction of CGI the first Web frameworks also appeared on the market, so one can say this was the point where true Web applications were born. Today, CGI is no longer a complete solution for Web development problems. Software is getting more and more complex and therefore there is the need to create new layers of abstraction to overcome the obstacles which unfortunately still occur when using the WWW.

With time and advancing techniques Web applications have become more and more popular with users and developers and they are unlikely to go away. The following definition narrows down the concept of **Web application** and as a result identifies characteristics, features and problems:

“An application running on a Web server which interacts with Web browsers over the Internet.”

Doubtless the most important benefit in this style of application is their ease of use. Users do not usually need to install additional software and everything runs in the same browser where all the other Web applications run. This saves disk space on the user's local PC, the control system of the browser is well-known and rich (history, back and forward button) and the user needs only minor support installing and maintaining the software. Developers also see a big advantage since they only need to deploy and maintain the software on their server. There is no need to navigate the customer through sometimes complicated install routines and the cost is thus less for both parties. Since all data and most processing occurs on the server; however, this can lead to a higher burden for performance and to a potential security risk. Also some customers like to store their data locally and the typical "feel" of desktop applications is difficult to achieve with Web applications. This criticism is currently being addressed with new technologies, like Ajax.

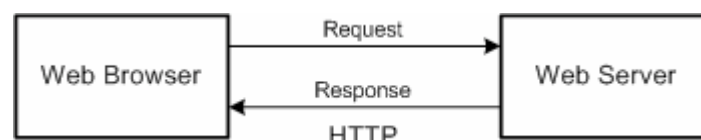


Figure 1.1: Typical Web Scenario

In figure 1.1 you can see a typical scenario. A client sends a request to a server; the server processes this request and sends a response back to the client. The Hyper Text Transfer Protocol (HTTP) serves as a transmission protocol with its advantages and disadvantages. As mentioned above the original purpose of the WWW was to link up static documents. The HTTP also originates from this time.

The HTTP is stateless and therefore the server cannot evaluate each request in a consistent control flow. This

means that control state needs to be constantly transported to and from the client so that the server can use it. In addition, the server cannot send any updates to the client, but rather has to wait for the next incoming request. HTTP is limited to the request/response cycle as you can see in figure 1.1. Other problems are the inability of the server to react to the interactions of the user with its browser. The server does not know if back or forward buttons were pressed and synchronization and flow control problems often occur, for example when a customer sends a form for a second time. Many frameworks still do not address these issues and thus it is the developer's job to develop workaround solutions. For a detailed introduction to the problems of Web development see [Stephan Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications.](#)

The need for complicated workarounds and non-obvious logic interferes with developer productivity and the maintainability of the code. It would be much nicer to reach a Web development level which approaches that of desktop applications. A Web application framework that supported reusable and recomposing components with their own control flows, standard objects for forms and sortable reports, automatic creation of valid XHTML (Extensible Hypertext Markup Language) and easy debug and compile cycles would improve the current situation a lot.

Seaside



Figure 1.2: Seaside Logo

Seaside is different! The framework provides a uniform, pure object-oriented view of Web applications and combines a component-based approach with a continuation-based one. With this, every component has its own control flow which leads to high reusability, maintainability and a high level of abstraction. Additionally it is written in Smalltalk which leads to the possibility to debug and update applications on the fly. It provides a layer over HTTP and HTML (Hypertext Markup Language) that let you build highly interactive Web applications.

Seaside is a freely available open source Web framework implemented in Smalltalk ([Squeak](#), [Pharo](#), [VisualWorks](#), [Dolphin](#) and [GemStone](#)) which was developed in 2002 by **Avi Bryant** and **Julian Fitzell**.

By means of continuations, multiple independent control flows between different components get modeled and these stateful components bind the Web application together. Initially it sounds a little complicated but it is not really. A continuation represents the control state at a given point in the computation. Seaside creates continuations at runtime and uses them to continue at the same point in the program execution later on. With this concept developers and users can come very close to the **feel** of a real desktop application (see chapter "The Fine Print"). This behavior is very difficult to implement in other Web application frameworks.

Seaside supports the development of a Web site through the use of different reusable components. It hides the complicated and costly problems like refresh or back buttons, stateless HTTP or generation of valid XHTML and connects the components using the strengths of Smalltalk (integrated development environment, access to development tools and rich debugging support, for instance).

A complete Seaside Web application results from the interaction of many components among themselves, as is typical in other object-orientated and component-based programs. Developers do not think in terms of pages but in terms of individual, stateful components which compose the user interface out of reusable smaller parts. Additionally, the use of callbacks (block closures) can specify actions which get triggered through a click (for example on a link by the user). All this makes it easier to reuse parts of applications as they are very loosely connected, login components or a table with sortable entries, for example.

Workflows can easily be modeled as a continuous piece of code in Seaside. Components will get embedded with a call/answer mechanism into any procedures and the back button of the browser still works correctly. Also Seaside supports CSS (Cascading Style Sheets), Ajax through [script.aculo.us](#) and Comet style server-push technology.

Unhandled errors and failures can be reported to the Web page instantly and developers can debug the program without a restart in the same session on the fly. But developers must take care to separate the data model and the user interface explicitly. No Web site templates will get supported and HTML markup is generated programmatically. But this ensures, by use of different patterns and the support for the CSS, a good design and the far-reaching possibilities of object orientation are not limited.

Coming to the end of this little introduction, the following is a part of the session notes of the OSCON 2006 from Avi Bryant, one of the inventors of Seaside.

Web Heresies: The Seaside Framework

“Over the last few years, some best practices have come to be widely accepted in the Web development world. Share as little state as possible. Use clean, carefully chosen, and meaningful URLs. Use templates to separate your model from your presentation.”

“Seaside is a Web application framework for Smalltalk that breaks all of these rules and then some. Think of it as an experiment in tradeoffs: if you reject the conventional wisdoms of Web development, what benefits can you get in return? Quite a lot, it turns out, and this experiment has gained a large open source following, seen years of production use, and been heralded by some as the future of Web applications.”

Avi Bryant, Dabble DB

Tutorial Contents

This tutorial provides an introduction to the Seaside Web framework. In twelve chapters, its fundamental elements are explained. The tutorial aims to provide the user an early start for quickly writing their own applications and achieving good results. We assume prior knowledge in some areas; they are listed below in Requirements.

The tutorial character based on one working example. It is a small task management application for multiple users called **ToDo Application**. We recommend that you do not try to do the whole tutorial at once, but repeat the examples in several of the chapters to allow the concepts to sink in. The example source code is accessible as well (see below), so that it will be easy for you to experiment with it and to gain your own experience.

Here is a brief description of the twelve chapters of the tutorial. The first chapter, which you are reading right now, deals with the conditions and looks at the technologies used in detail. In the second chapter, we start to build the foundation for the ToDo Application, introducing how to work with Seaside and how to configure an entry point, before concluding the chapter with the first static text output on a Web site.

Chapter three builds the model for the application, which means what a **user** or a **task** look like. This is done based on the MVC pattern. Chapter four is about one of the most important Seaside philosophies: it explains the Component concept. Seaside Web sites are generated by user-written reusable components. Web forms are the main parts in chapter five and it is shown how easy it can be to realize those with callbacks. Chapter six then goes one step further by showing the possibility of using tasks, which can be understood like workflows making it very easy for the developer to program Web application behavior. This works on the classic example of user administration, which here casts a light on the own session.

External resources like images, CSS or JavaScript may belong to a Web site as well. Chapter seven describes the possibilities to manage such external resources you have with Seaside and their pros and cons. Up to that point, the whole application is only practicable for one time; afterwards, all user data are forgotten. Persistence is treated in the eighth chapter which presents four different possibilities in detail: with Glorp, we take a look at an O/R mapper for PostgreSQL, and with GOODS and Magma, at the luxury of object-oriented databases. We also discuss the approach of saving all data in the Squeak image.

In the ninth chapter, the focus is on an additional library which makes it possible to use Ajax functionality in Seaside Web sites. The script.aculo.us framework, with the Seaside integration provided by Lukas Renggli, offers an easy way to create your own Web site in the style of interactive Web sites. The tenth chapter introduces the meta-description framework Magritte which allows you to easily create different views on and input dialogs for domain objects without much coding. Chapter eleven describes some internal details of Seaside with which you can easily understand how it does work. Finally, in chapter twelve, we treat some topics that did not fit into any other chapters - for instance, coding conventions, sources for more information and links to interesting blogs on the topic of Seaside.

Requirements

The Seaside Tutorial has mainly been made for people with basic knowledge of object-oriented programming especially with Smalltalk, but it is suitable for others as well. We try to reduce the fundamental knowledge on the elements demanded here. If you have suggestions for better descriptions of any part, please send us your

feedback.

Recommended Knowledge

- **Object-oriented programming** should be known well enough that words like class, object, inheritance or reuse do not put question marks in your eyes. Here, no links are given, just because there is a wealth of standard literature and we do not want to prefer any of it.
- **Smalltalk** (as a programming language) has to be known well, because Seaside itself only exists in this language. Should you have any problems with it, have a look at the following link:
 - [Smalltalk Free Books](#)
- **Squeak** is used as the Smalltalk environment of choice in this tutorial, but Seaside has also been implemented on other common Smalltalk platforms (see the corresponding documentation). For more help with the treatment of Squeak, the following links are quite helpful. However, we try to use particularities of the Squeak Smalltalk dialect as seldom as possible
 - [Squeak by Example \(ebook\)](#)
 - [Squeak Web site](#)
 - [Mailinglists](#)
 - [Squeak Wiki](#)
 - [Morphic Tutorial](#)
- Knowledge in **HTML** and **CSS** for the easier training of this topic are quite desirable. A complete reference to this topic can be found here:
 - [W3 Schools](#)

ToDo Application

Basically, the ToDo Application is a simple task management system for multiple users. Every user can create, edit, and categorize tasks for their own use. The system shows the next job and gives the user the possibility to manage their tasks. Also, there are forms to create new users or to log in to the private sections of the Web site. The simple scenario covers all the main features of a Web application, like forms, persistence and Ajax.

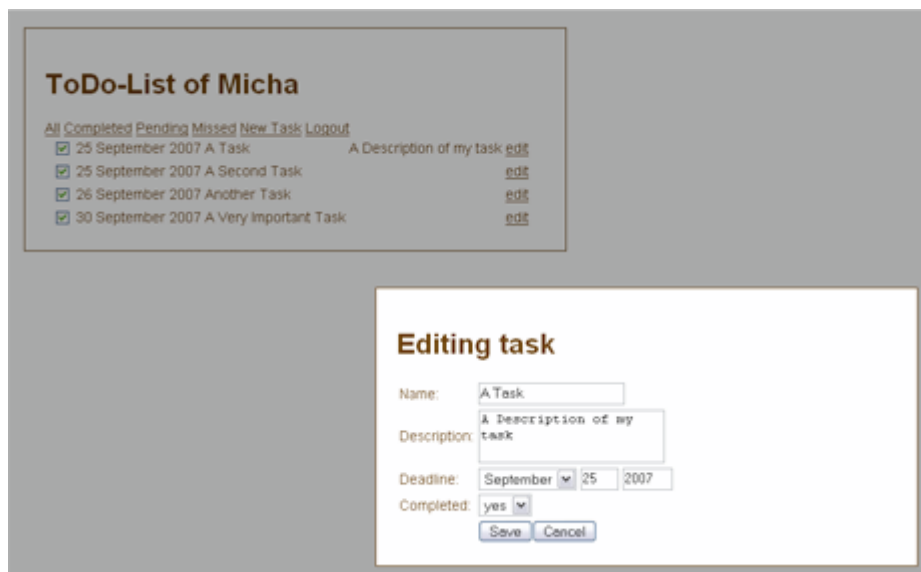


Figure 1.3: ToDo Application Demo Screenshot

In figure 1.3 you can see a screenshot of the application (final version). This should present you a little preview of the ToDo Application and shows what the application performs and what it looks like.

Tutorial Source Code

Here you can find two change sets. The first one contains only the source code with an image database and without a dependency to Glorp. The second one is the same except that it includes the changes made by using Magritte. For first experiments with the Web application, Glorp is optional. For details on how to install GOODS or Glorp, please read chapter "Persistence".

[Change Set: ToDo Application](#)

[Change Set: ToDo Application with Magritte](#)

Install the Tutorial Change Set

First save the change sets in your Squeak image directory (see chapter "First Steps - Using the Right Image"). After that, you can open a file list browser in Squeak (world menu -> open... -> file list) and select the change set. Look at the upper right corner in this window and click the **fileIn** button. Another simple way is to drag and drop the change set file into your open Squeak image. A menu will open and you can click on the entry **install into new change set** or **fileIn entire file**.

Now you can, in your favorite code browser, see the package *STTutToDoApp* and all classes we have written in this tutorial. You can also use the web-dev and any other Squeak image in which Seaside is already installed and you can open a Web browser with the URL (Uniform Resource Locator) **http://localhost:8080/todo/** and enjoy the ToDo Application. In your Web browser, use HTTP port 8080 or the one you specified in Seaside.

Summary

Chapter 1 begins with the problem of Web development and offers as possible solution the Web framework Seaside. This tutorial shall be an introduction into this interesting topic, the most important concepts are explained in one working example. The requirements to the reader and the ToDo Application are a topic in the following. Finally, the source code is presented in short. If you want to know how to create your first component with Seaside, read the next chapter.

[Back](#)

2 - First Steps

What you are going to learn

- [Using the Right Image](#)
- [Installation](#)
- [Creating the First Component](#)
- [How to Configure Your Application](#)
- [How to Use Halos](#)
- [How to Debug](#)
- [Summary](#)

Using the Right Image

When you decide to create an application with Seaside and Squeak, you also have to decide which versions to use. This tutorial is made for Squeak 4.2 and Seaside 3.0. Please note that Squeak and Seaside are supported by the community and the current version may be much younger than the suggested ones. That said, we do not guarantee any of the code given here to function in more recent or older versions of Seaside or Squeak.

Current Squeak images with Seaside already integrated can be downloaded, for example, from the Squeak web site <http://squeak.org/Documentation/Installation/#h-5>.

If you are new to Squeak and Seaside you should have a look at the [Seaside All-In-One](#). It contains a full installation of Seaside 3.0 and is ready to use for Windows, Linux and Max OS X. Only extract the archive and run the executable of your platform.

Installation

If you use a prepared image, like the one you can download as suggested above, you do not need to install any packages at this moment. Those images should already have Seaside and script.aculo.us integrated.

Otherwise you can install Seaside manually (see figure 2.1). Open a SqueakMap Package Loader (world menu -> *open...* -> *SqueakMap Package Loader*) and find the package *Seaside*, right click on it and choose install. Wait a moment and follow the instructions below to check and start the server.

Another way to install and to use for later updates is to open a Monticello Browser (world menu -> *open...* -> *monticello browser*) and add the following repository (*Repository type* -> *HTTP*) or you can use the entry at the *Seaside2* package if it is already installed.

```
MCHttpRepository
  location: 'http://www.squeaksource.com/Seaside'
  user: ''
  password: ''
```

Click on open and choose the packages you want to install (load icon) or update (merge icon) (see figure 2.2). Here you can find many other useful add-ons and the latest version of Seaside.

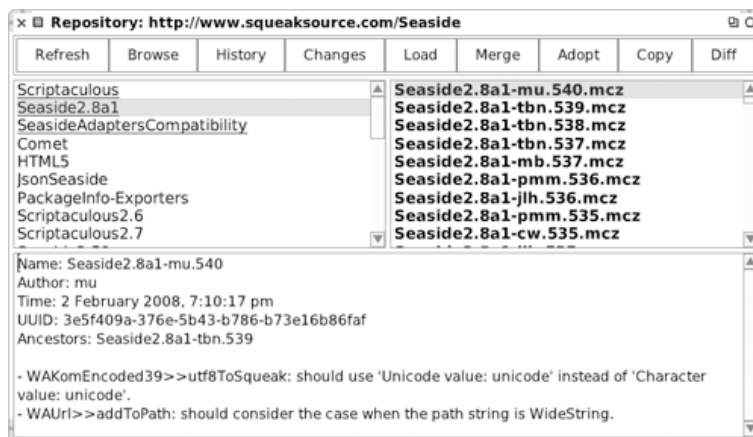


Figure 2.2: Seaside repository

As Seaside is hosted by a Comanche Web server, you need to have that server running for going on with this tutorial.

So, make sure that Seaside is running correctly by opening a workspace and doing the following:

```
HttpService allInstances explore.
```



Figure 2.3: HttpService allInstances explore.

In figure 2.3 you can see an explorer window with an array. If the first element is similar to

```
1: seaside [running] port: 8080
```

then Seaside is already started and running. Notice that the dedicated port is 8080. If the suggested array is empty, you have to start the server for hosting Seaside manually. That means to open a workspace and do

```
WAKom startOn: 8080.
```

We let the server listen on port 8080. Note that you can choose the port freely under consideration of security policies of your operating system.

Stopping the server is as simple as starting it. Just do the following in a workspace:

```
WAKom stop.
```

If you need another encoding (maybe you want to publish a German Web site), stop the current Web server and start a new one with the following code:

```
WAKomEncoded39 startOn: 8080.
```

Now make sure your server is started and running, and be ready to check what you have done so far. Just open the URL <http://localhost:8080/> in your local Web browser.

You should now see the Dispatcher Viewer of Seaside (figure 2.4).



Figure 2.4: Dispatcher View

This page shows all currently registered Web applications. By clicking on the name you simply open the designated Web site. Feel free to explore the tests to discover the capabilities Seaside will offer and get a feeling of what you are going to learn during the next chapters.

Creating the First Component

You should now be well prepared for making the first steps on creating Web-based applications with Seaside. Now you can try building your first component.

But first let us discover what is necessary:

- Every newly-created component is inherited from **WAComponent**.
- Every component needs a method called **#renderContentOn:** accepting a renderer.

- Components that shall be root components need a class method called **#canBeRoot**.

So what does that mean?

WAComponent is the basic component provided by Seaside. That means every visible component inherits from **WAComponent**. There are many more components you can use, like **WATask** or **WABatchList**, but more on that later. For now, just keep **WAComponent** in mind.

While inheritance is one requirement, the rendering part is another. To generate HTML code from within your Smalltalk code, you need a renderer. **WAComponent** ensures that the method **#renderContentOn:** is called. By giving the method a renderer you have access to all **WACanvas** methods provided by Seaside. So let us keep that theoretical stuff in mind and create the first component of our ToDo Application

First you should add a new class category, *STTutToDoApp*.

After that we create the first class. Since we ultimately want to have a ToDo Application, we should think about a root component, a component that will be the parent for every other component. So let us say it should look like this:

```
WAComponent subclass: #StRootComponent
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'STTutToDoApp'

StRootComponent
```

Now you have a new blank class ready to serve rendered content. As mentioned before, we need the **#renderContentOn:** method. So just create it and give it the parameter for the renderer.

```
renderContentOn: html
  "I render everything by calling html"
```

You may realize that the parameter is called **html**. That is our renderer. We call it **html** because we only render HTML content. Now you are ready for some action! Imagine you want your Web application to display its name. If so, you can implement the renderer this way:

```
html text: 'ToDo Application'.
```

Now you ordered the renderer to print out some text with the given string. Whilst seeing the site show the name of the application is enjoyable, you could also imagine it showing the current date as well.

To achieve this, the complete method should look like this:

```
renderContentOn: html
  "I render everything by calling html"

  html text: 'ToDo Application'.
  html text: Date today.

StRootComponent>>#renderContentOn:
```

Remember we want that component to be the root component, so let us also declare that. Change to the class side and add the method **#canBeRoot**.

The only task of the method is to return the Boolean value **true**. It is needed to register your application later on.

```
canBeRoot

^true

StRootComponent class>>#canBeRoot
```

Notice that you overwrite the existing **#canBeRoot** method of **WAComponent** that returns false by default.

How to Configure Your Application

Right now, you have created your first class of the ToDo Application and you surely want to see what has been done so far. Therefore, we will continue registering the application with the Seaside configuration. Open the URL: **<http://localhost:8080/config/>**.

You should now see a configuration page more or less like in figure 2.5, figure 2.6, and figure 2.7.

Remember to use your defined port if it is not 8080. You should now be asked for a login. Here you use the credentials you defined when you installed Seaside manually, or you use *User: admin - Password: seaside* (which are default credentials).



Seaside2.8a1-lr.522.mcz 28 October 2007 2:56:18 pm

/seaside/

| | | | | |
|--------------------------|-------------------|---------------------------|----------------------|------------------------|
| browse | Dispatcher Viewer | Configure | Copy | Remove |
| config | Dispatcher Editor | Configure | Copy | Remove |
| examples | Directory | Configure | Copy | Remove |
| files | Files | Configure | Copy | Remove |
| pier | Pier CMS | Configure | Copy | Remove |
| tests | Directory | Configure | Copy | Remove |
| tools | Directory | Configure | Copy | Remove |

Figure 2.5: Seaside Configuration - Registered Applications

Settings

Default entry point

browse

Add entry point

Name: Type: Application

Figure 2.6: Seaside Configuration - Settings

Again, you can see an overview of all registered applications (Figure 2.5) but now, you can also change configurations or remove entries. Take a look at the input field under *Add entry point* (figure 2.6). An entry point defines the URL of your application. Give it a try. Fill in the name **todo** and choose application as a type. Click add when you are done. Additionally, in figure 2.7 you see some statistics about the running Seaside server, like uptime, memory usage or garbage collection. Sometimes it is necessary to clear all the server caches manually for it sees the link *clear caches* at the bottom of the page.

Statistics

```

uptime                25h2m1s
memory                56,789,740 bytes
  old                  53,580,732 bytes (94.3%)
  young               184,332 bytes (0.3%)
  used                53,765,064 bytes (94.7%)
  free                3,024,676 bytes (5.3%)
GCs                  184,490 (488ms between GCs)
  full                82 totalling 15,610ms (0.0% uptime), avg 190.0ms
  incr               184408 totalling 111,306ms (0.0% uptime), avg 1.0ms
  tenures             552 (avg 334 GCs/tenure)
Since last view 31 (663ms between GCs)
  uptime              20.5s
  full                0 totalling 0ms (0.0% uptime)
  incr                31 totalling 19ms (0.0% uptime), avg 1.0ms
  tenures             1 (avg 31 GCs/tenure)

```

[Clear Caches](#)

Figure 2.7: Seaside Configuration - Statistics

Figure 2.8 shows you the preferences you can set for your applications. Currently, you can just ignore the options for libraries and ancestry. Look at configuration and notice the field *Root Component*. Here you can see every component that returns true in the **#canBeRoot** method. Look at the drop down list and choose **StRootComponent**. Just ignore the rest of the configuration view. — it will be described later on.

Libraries

Add Library:

- WStandardFiles [\(remove\)](#)

Ancestry

Add ancestor:

- WRenderLoopConfiguration [\(remove\)](#)
 - WSessionConfiguration
 - WAGlobalConfiguration

Configuration

General

| | | | |
|-----------------------------|-----------------------------------|--------------------------|---|
| Deployment Mode | false | override | inherited from WAGlobalConfiguration |
| Error Handler | WAWalkbackErrorHandler | override | inherited from WRenderLoopConfiguration |
| Main Class | WRenderLoopMain | override | inherited from WRenderLoopConfiguration |
| Redirect Continuation Class | WRedirectContinuation | override | inherited from WSessionConfiguration |
| Redirect Handler | WRedirectHandler | override | inherited from WSessionConfiguration |
| Render Continuation Class | WRenderContinuation | override | inherited from WSessionConfiguration |
| Root Component | <input type="text" value="None"/> | clear | |
| Session Class | None | override | inherited from WSessionConfiguration |
| Session Expiry Seconds | PRPierFrame | override | inherited from WSessionConfiguration |
| Use Session Cookie | SUAllTests | override | inherited from WSessionConfiguration |
| | StRootComponent | | |
| | WAAllTests | | |

Figure 2.8: Web application configuration

Congratulations! You have built your first component of your new ToDo Application. Click on save, and on close after that. Take a closer look at the list of registered applications. Your todo entry point should appear there now. Just click on the name and see the result (figure 2.9) of your rendered component. The URL should be **http://localhost:8080/todo/**.

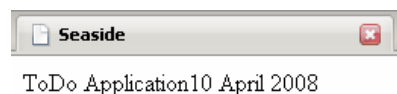


Figure 2.9: ToDo Application - First Component

How to Use Halos

Maybe you did not enjoy your first application that much and were put off by the menu at the bottom of your browser window. This menu will only be available when your application is not in deployment mode and gives you some additional ways to inspect your application. Let us concentrate on the link called *Toggle Halos*. Just click it and see what happens.

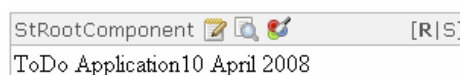


Figure 2.10: Seaside Halos

You should immediately notice the changed view (figure 2.10). A frame, called a halo, has appeared around your component. There is a top line with some icons and the name of your component (**StRootComponent**). The icons are clickable. There is one for opening a class browser, one to inspect objects and a link for the CSS style editor. While we leave out the inspection halo let us have a closer look on the class browser link.

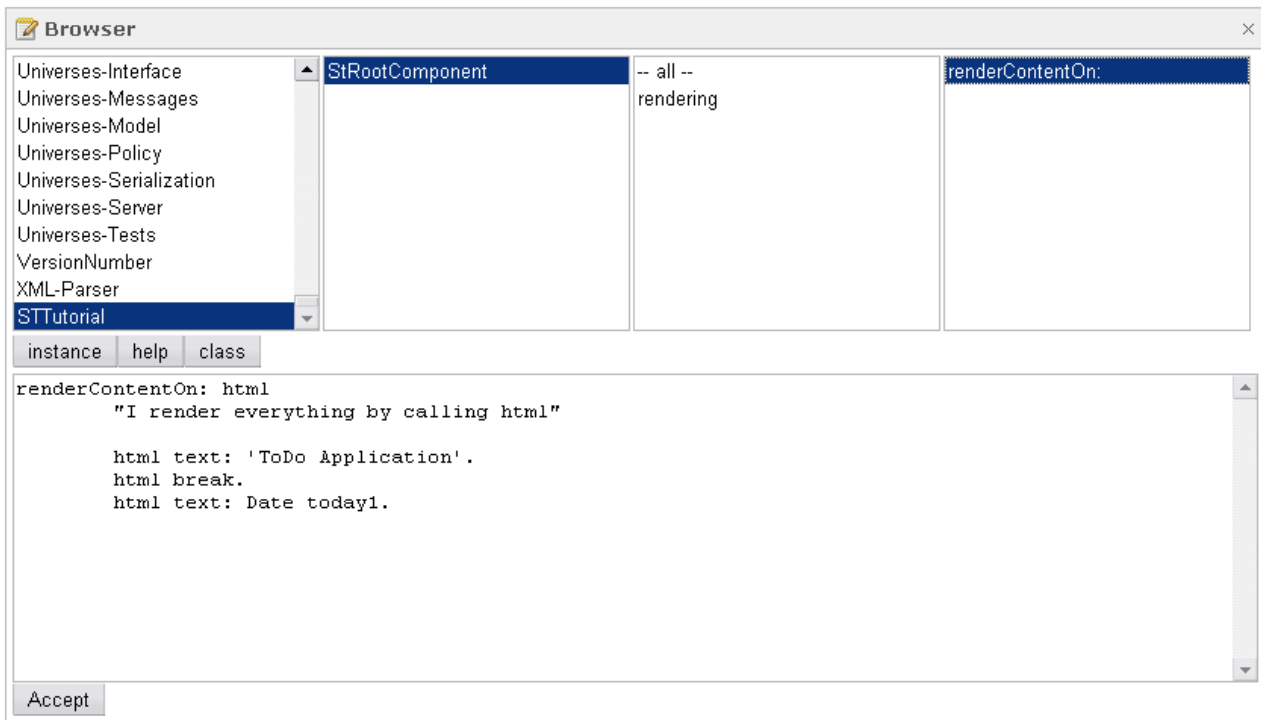


Figure 2.11: Web class browser

After clicking on the class browser halo you can see a class browser style window (figure 2.11). Indeed, this browser contains every available Smalltalk class and you are able to change those classes with it. Remember the importance of having your application in deployment mode when making it public.

Your Seaside component class should be automatically selected. What about changing the appearance of your application? Having the name and the date on the same line and without any whitespace looks strange. We can now change that **on the fly**. Click on the **#renderContentOn:** method in the right pane of the class browser. You should now see the method you created earlier. Just put in

```
html break.
```

between both text messages and click on accept. After clicking on the *x* on the right side of the top line of the class browser you can now see your application again. Notice the line break between the name and the date. In summary, the method should now look like this:

```
renderContentOn: html
    "I render everything by calling html"

    html text: 'ToDo Application'.
    html break.
    html text: Date today.
```

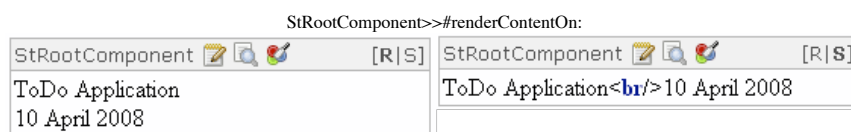


Figure 2.12: Render vs. Source View

Before you can explore the halos by yourself you should point your view to the right side of the halo menu. There you can see *[R/S]* where the *R* means rendered view and the *S* stands for source view (figure 2.12). Try switching both views and notice the generated HTML source code when clicking on the *S*. This HTML code is generated by your renderer used in the **#renderContentOn:** method.

Just for your information: there is the CSS style editor where you can write pure CSS code regarding your classes and IDs. When adding a CSS command in this window, a **#style** method is added to your class. It only returns the CSS code as a Smalltalk string. This looks nice and simple at first, but keep in mind that this feature is deprecated. It is better to create an external CSS file and link it to the application (explained later on), especially when having big applications. Never underestimate the advantages of having the style commands in one resource over having them in thousands of classes with thousands of commands.

How to Debug

When talking about advantages of Seaside, you have to talk about debugging. While Seaside is integrated into Squeak or VisualWorks, it also works with the infrastructure of the images. This means that debugging with Seaside is debugging with the image-integrated debugger on the fly.

Let us test it. Just force an error and the raising of the debugger. Imagine you want to change your application to show the date of yesterday instead of today. So we could subtract 1 from the date. But now we make a little mistake: instead of subtracting, we divide the date by 1. Change the code to be similar to this:

```
renderContentOn: html
    "I render everything by calling html"

html text: 'ToDo Application'.
html break.
html text: Date today / 1.
```

StRootComponent>>renderContentOn:

When you refresh your application's Web page, you should see a Seaside-generated error page (figure 2.13). Have a good look at it. There you see (from top to bottom):



Figure 2.13: Seaside Error Page

- Error message directly below the Seaside logo.
- Links for handling the error.
- Suggestions as to what could have caused the problem.
- Finally, the stack trace of the last sent messages.

Now that you can see the error, you decide to solve the problem on the fly. Take a look at the links below the error message. Just click on the **debug** link. Notice: Nothing happens! Really? Have a look at your Squeak desktop window.

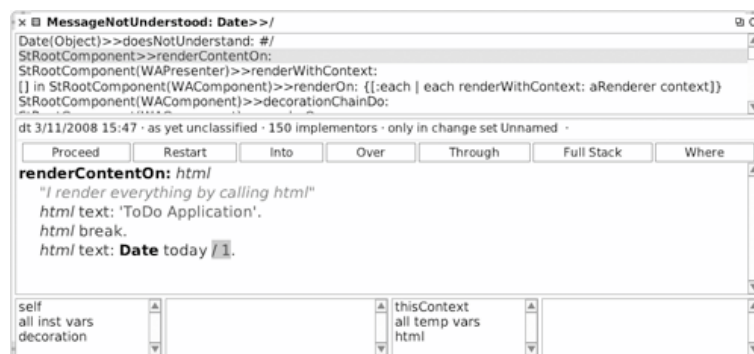


Figure 2.14: Squeak Debug Window

There should be a debug window (figure 2.14) waiting for your attention. As you already know how to work with Squeak, click on the entry where **#renderContentOn:** is sent to **StRootComponent**. The code causing the problem should be highlighted.

Now just remove the division and let the method look like this:

```
renderContentOn: html
    "I render everything by calling html"
```



```
html text: 'ToDo Application'.  
html break.  
html text: Date today.
```

StRootComponent>>#renderContentOn:

Save your changes, click on Restart and then on Proceed. The debug window disappears. Switch back to your Web browser and see the page rendered correctly.

Summary

In this chapter you have learned how to create your first Seaside Web application. Congratulations, the most difficult part is done. Simple, was it not? You started with the installation of a Seaside server, worked through the creating and configuring of your first Web component and ended up using the tools you found in Seaside and Squeak... The next chapters are going to get deeper and deeper into the Web framework and build, piece by piece, the entire 'ToDo Application'.

[Back](#)

3 - ToDo Application

What you are going to learn

- [Model Overview](#)
- [User Model](#)
- [Task Model](#)
- [Summary](#)

Model Overview

We have discussed the basic features of Seaside and demonstrated their use. Now, we will start to develop our ToDo Application. First of all, Seaside implements a precise separation between the model and the view of data. So before we can start to build the Web site (view), we need to think about the model. This chapter describes the two main classes **StUser** and **StTask**, which we derive from our description (see chapter "Introduction"). Figure 3.1 shows a UML (Unified Modeling Language) chart with the dependency between the two classes.

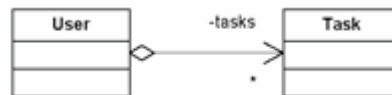


Figure 3.1: ToDo Application (UML Diagram)

The user can administer tasks in our application. This short specification suggests to us the UML chart in figure 3.1. The system is used by different users, each of whom has a set of individual tasks (aggregation in figure 3.1). The data model is deliberately kept very simple because we do not want to distract the reader's attention from Seaside. It is easily possible to integrate your own (possibly more sophisticated) data models into a Seaside application.

In the following, we explain the attributes and methods of the classes **StUser** and **StTask** in detail. The topic **Persistence** is the content of chapter 8

User Model

The user class is the most important class in our data model. An arbitrary number of users will be created in our application and saved in the database later on. Users will be identified in distinct sessions after the log in process. Based on the user's session, all further data, their tasks for instance, will be found, so that only their owner can manage them. The following class definition shows the **StUser** class with its instance variables required for identification, authentication, registration and application-specific properties.

```

Object subclass: #StUser
  instanceVariableNames: 'id userName email tasks password'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'STTutToDoApp'

```

StUser

After the creation of the class **StUser**, it is highly recommended that you perform automatic creation of instance variable accessors. One possible way to do this in Squeak is to use the context menu on a class among the entry *more....* There, you can find *create inst var accessors*. Select this command to have all accessors created. In different browsers or environments, this feature may be accessed differently, so consult your system documentation in case you are not using Squeak and its standard browser. When all else fails, you will have to use hand-written accessor methods.

If you look at the source code, you can see the super class of `StUser` is **Object**. That is, the entire model of the application is independent from Seaside. This implies that reusing existing models is also possible. Later on, we will use Seaside to link the model and view to a dynamic Web page.

The class **StUser** provides the following instance variables:

id

A unique key for identifying persistent objects primarily used in databases as primary key.

userName

A **String** object, which is used for addressing the user.

email

For uniqueness of registration and usernames we require their email address (**String**). Additionally, we can contact the user personally.

tasks

This attribute reflects the aggregation of the UML chart. An **OrderedCollection** consists of task objects and is assigned to every user individually.

password

Based on standard security mechanisms, the password is encrypted with a hash function (for example **MD5**) and stored in this attribute as a **String**. This way, we guarantee that no password string is saved directly.

In addition to the accessor methods, this class has two extra methods:

```
initialize

self tasks: OrderedCollection new.

StUser>>#initialize
```

The initialize method creates a new **OrderedCollection** for tasks of every new user object.

```
addTask: aTask

^ self tasks add: aTask

StUser>>#addTask:
```

The method just adds a task object to the collection. It may be surprising that there is no **#removeTask:** method. The reason for this is in our specification: we want to show completed tasks, so we do not require a delete function. This may seem like an example of bad practice for application development, but the simplification has a few advantages. If you develop your own applications, remember to delete your obsolete objects.

Task Model

The class **StTask** shows our task's data model. A task always belongs to an ordered collection of tasks maintained by its owning user. However, it does not need to know about the user object and can be seen as a unit of its own. Its properties give the task the ability to describe and file itself and to deposit its state in time and process. Therefore, it is important to mention not to mix it up with the **WATask** of Seaside. Both are completely different classes and have nothing in common.

```
Object subclass: #StTask
  instanceVariableNames: 'completed deadline taskDescription id
  taskName'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'STTutToDoApp'

StTask
```

The **StTask** inherits from **Object** and so it does not have a direct connection to Seaside. Again, the accessor methods can be generated automatically. There is nothing mysterious about the class, the instance variables are

the following:

completed

The state whether a task is finished or not. Only the Boolean value is saved.

deadline

The date by which the task should be completed. This value is used in subsequent computations by the two test methods (see below).

taskDescription

A text describing the task in detail.

id

Analogous to the **StUser**, a primary key is required for database storage.

taskName

A name that briefly describes the task.

*You should not use the variable names **name** and **description** instead of **taskName** and **taskDescription** because both names are used internally in Squeak.*

Analogous to **StUser**, there are again two special methods that, this time, test the completion state of a task.

```
isPending
```

```
^ self completed not and: [self deadline >= Date today]
StTask>>#isPending
```

This method tests whether the state of the task has been completed or not. If the deadline is still ahead and if the task has not yet been completed, it will be pending.

```
isMissed
```

```
^ self completed not and: [self deadline < Date today]
StTask>>#isMissed
```

This method describes a logical value that shows whether the deadline has been crossed without the task having been finished by the user.

Remember to initialize the task with a default state for completed and for the deadline:

```
initialize
```

```
self
  deadline: Date tomorrow;
  completed: false.
StTask>>#initialize
```

Summary

This chapter, relatively independent from Seaside, should be the basis for the application thus the main model of a simple ToDo list for multiple users is introduced here. It is easy to understand and stores only the most necessary objects (tasks and users). The important component concept is the content of the next chapter.

[Back](#)

4 - Components

What you are going to learn

- [Think in Components](#)
- [Creating Our ToDo Components](#)
- [Reuse](#)
- [Gluing Components Together](#)
- [Summary](#)

Think in Components

As we have already said, Seaside's way of creating Web applications is different from most other Web frameworks. Seaside adopts a component-based approach to tie different objects and their contents together and generate a single Web page from them. That way, web interfaces can be constructed as a hierarchical tree of stateful objects. To clarify this, imagine a simple standard Web page: usually, there is some kind of navigation means, a menu, and another part of the page contains the real contents corresponding to the particular menu items. This resembles a very simple component structure. The root component represents the visible page as a whole. If you put all your rendering code into this single component you would flinch at the immense complexity: there would be a massive amount of code pertaining to all kinds of different aspects, but no clear structure.

Components are a way to semantically divide your page into parts. Obviously, the navigation menu is one such part. Everything that belongs to it goes into our new menu component. Furthermore, the menu component instance can contain information about its current state. That is, you can safely store the currently activated menu item in it. The reason for this is that each user's session data can be stored in dedicated component instances this way. Sessions and the pertaining component instances are entirely managed by Seaside. Your root component only has to inform Seaside that menu component is its child and render it in the right place.

The currently shown page, corresponding to the last selected menu item, is another component. The component paradigm does not stop here: a page can contain forms, lists and other standard elements. These can be components, too. For example, a subscription form or login field can be written as components and therefore be reused later on in other applications. Seaside even provides many standard components, for example a simple time selector or a mini calendar (date picker).

As you can see, it is easy to construct a system from different collaborating components. In fact, this approach is similar to approaches adopted in assembling usual desktop applications, where reuse also is an important goal. With Seaside you are able to easily create reusable components whose appearance can be controlled via CSS.

Creating Our ToDo Components

Now let us go back to our evolving ToDo Application. So far, we have created model classes for representing users and tasks, and a simple component, which we have called **StRootComponent**. It will serve as the root of our component structure. As a first step, we would want to have two other components: a menu where we can choose whether we want to see pending or completed tasks, and a component that presents a corresponding list. So, create two empty subclasses of **WComponent** named **StMenuComponent** and **StListComponent**. To get some visible results, we start with changing the rendering in our root component to something more valuable:

```
renderContentOn: html

    html heading: 'ToDo-List'.
    html div
        class: 'menu';
        with: self menuComponent.
```

```
html div
  class: 'list';
  with: self listComponent.
```

```
StRootComponent>>#renderContentOn:
```

But where do the **menuComponent** and **listComponent** come from? They do not exist yet. So, add them to the instance variables of the class **StRootComponent** and create the accessor methods for them. These variables also have to be initialized:

```
initialize

super initialize.
self
  menuComponent: StMenuComponent new;
  listComponent: StListComponent new.

StRootComponent>>#initialize
```

Here, it is very important to call the **#initialize** method of our super class **WACComponent** because it is required for various data initialization purposes. If you render our Web page in your browser now, you get an almost empty page: only the headline is displayed. But you can see the sub components with enabled halos (figure 4.1). The reason is, of course, that we do not render anything in our subcomponents yet. Therefore, let us fill them with some content:

```
renderContentOn: html

html anchor
  callback: [];
  with: 'Menu entry'.

StMenuComponent>>#renderContentOn:
```



Figure 4.1: Root Component with Halos

This little piece of code does not do anything really useful, it just prepares us for the next level. It creates a simple anchor with the title **Menu entry** on our page. The **#callback:** message applies a block to our anchor, which will be evaluated as the user clicks on the link. We leave this empty for now.

```
renderContentOn: html

html table: [
  html
    tableRow: [
      html tableData: [html text: 'Table entry'];
    ]
    tableRow: [
      html tableData: [html text: 'Table entry'];
    ]
].

StListComponent>>#renderContentOn:
```

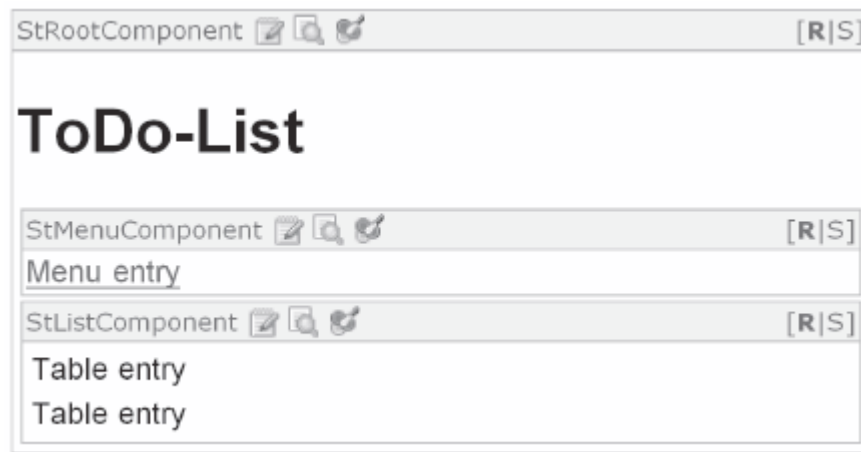


Figure 4.2: Root Component with Link and Table

Again, this is just a simple snippet showing you how to create a basic table. You can see that you do not need to write HTML code to create a table structure. You will see how to beautify it via CSS in one of the next chapters. If you look at your Web page now (figure 4.2 shows it with some style sheets added), you will see that our components have rendered the content as intended. But if you click on the link, an error shows up, although nothing should happen! The reason is that our root component knows its subcomponents, but we did not tell Seaside about them. For this, you need to override the **#children** method and return an array containing all our child components.

```
children
    ^ Array with: self menuComponent with: self listComponent
    StRootComponent>>#children
```

Before we get our real business data into our components, we have to dig into another lesson...

Reuse

Component reuse is a worthwhile goal, though sometimes hard to achieve. It can consume much time better spent on your application rather than perfectionism. However, if your applications are getting larger, the time will come when you have to refactor the code and merge similar components into one. This will result in more useful components, perhaps reusable in more than one or two of your Seaside applications.

For now, let us assume our tutorial menu component has to be reused in other places. This means that we cannot simply enter our menu entries into the source code of the component but instead make it configurable. So we need something like **StMenuComponent>>#addMenuEntry:**. As the action to execute upon clicking the link is unknown to the menu component, too, it has to be passed as a block side by side with the name of the link. This may sound like a dictionary where we map a title to its action, but that is unfortunately not fully correct, since a dictionary is unordered, so you will never know in which order the inserted data are stored. Instead, we will use an **OrderedCollection** which contains **Associations**. Please create an instance variable named **entries** and some accessor methods in our **StMenuComponent** and initialize it with an empty **OrderedCollection**. Remember to call **#super initialize** if you implement that in the **#initialize** method. Now we can continue with the method for adding entries:

```
addEntry: aString withAction: aBlock
    ^ self entries add: aString -> aBlock
    StMenuComponent>>#addEntry:withAction:
```

That was pretty easy. As these entries will not be rendered yet; we have to change our simplistic **#renderContentOn:** implementation. An entry is an association between a string and a block, with the message **#key** at **#with:** we add the string as output text and with **#value** we return (not execute) the block and connect it to the **#callback:** method.

```
renderContentOn: html

self entries
  do: [:entry |
    html anchor
      callback: entry value;
      with: entry key]
  separatedBy: [html space].

StMenuComponent>>#renderContentOn:
```

Now we have a reusable menu component. We could add some CSS classes to be able to configure the design via style sheets, but this will be covered in one of the next chapters. Of course, you want to see some results now, therefore we refactor **StRootComponent>>#initialize** to call **self initializeMenuComponent** instead of simply initializing the **menuComponent** instance variable. **#initializeMenuComponent** then will look like this, although we are not filling in actions because our list component is not ready yet:

```
initializeMenuComponent

self menuComponent: (StMenuComponent new
  addEntry: 'All' withAction: [];
  addEntry: 'Completed' withAction: [];
  addEntry: 'Pending' withAction: [];
  addEntry: 'Missed' withAction: [];
  yourself).

StRootComponent>>#initializeMenuComponent
```



Figure 4.3: Reusable Menu Component (HTML)

In figure 4.3 you can see a simple list of menu entries (HTML Code) beneath the headline. Additionally you can see that Seaside have generated all links automatically and the developer doesn't need to know which link triggers which piece of code (callback). So, let us move on to our list component. Assume there is a need to make this one reusable, too. How can we do this? It shall show us a list of to-do entries, therefore we have to supply it with this list. However, not every list item should always be displayed and furthermore they have to be sorted in different ways. This means that a filter and a sort block are needed. Additionally, the list components should consist of arbitrary types of items. The problem with this is that the component cannot decide how an item is rendered. As Seaside supports the separation of model and view, the **StTask** must also not know its appearance. Therefore, we need a **renderItem** block, which specifies how to render the given item. Please create the four instance variables **items** **filterBlock** **sortBlock** **renderItemBlock** and their accessors in the **StListComponent** class. Afterwards we can alter the **#renderContentOn:** method:

```
renderContentOn: html

html table: [(self sortBlock
  value: (self filterBlock value: self items))
```



```
do: [:item | html tableRow: [self renderItemBlock
    value: item
    value: html]]].
```

```
StListComponent>>#renderContentOn:
```

As you can see, this method first sends the items through the set filter and afterwards through a sort block. For each remaining item a table row is created and the `renderItemBlock` is evaluated with the item and the current renderer as parameters. The block should use the delivered html object to render the item data into the table rows.

Now the first version of our list component is ready to render a filtered and sorted list of items. However, be aware that the two components presented here are designed for reusability. You most certainly would not implement them like this for an application as simple as ours if you did not want to support reuse.

Gluing Components Together

To finish this chapter, we want to get our little component structure to work as expected. First, the correct initialization of our list component is needed. So, as done before, take out the assignment to the **listComponent** instance variable from the **StRootComponent>>#initialize** and instead call this new method from there:

```
initializeListComponent
```

```
self listComponent: StListComponent new.
self listComponent
    sortBlock: [:items |
        items sortBy: [:a :b | a deadline < b deadline]];
    renderItemBlock: [:task :html |
        self renderTask: task asRowOn: html].
self showPendingTasks.
```

```
StRootComponent>>#initializeListComponent
```

This code snippet sets a simple sort block after the list component is instantiated. In the next line the `renderItemBlock` is set, which calls a **#render** method on self. So the **StRootComponent** knows what a task looks like. Afterwards it sets the currently shown filter to display only the pending tasks. The method **StRootComponent>>#showPendingTasks** is one of four methods which just set the corresponding filter in the list component:

```
showPendingTasks
```

```
self listComponent
    filterBlock: [:items | items select: [:item | item isPending]].
```

```
StRootComponent>>#showPendingTasks
```

```
showAllTasks
```

```
self listComponent
    filterBlock: [:items | items].
```

```
StRootComponent>>#showAllTasks
```

```
showCompletedTasks
```

```
self listComponent
    filterBlock: [:items | items select: [:item | item completed]].
```

```
StRootComponent>>#showCompletedTasks
```

```
showMissedTasks
```

```

self listComponent
  filterBlock: [:items | items select: [:item | item isMissed]].

StRootComponent>>#showMissedTasks

```

We are almost done. Of course it is necessary to call the methods defined above. We already were at that point when we finished the menu component. Go into the **StRootComponent>>#initializeMenuComponent** and adapt the actions for the menu entries, which were previously empty, to the following:

```

initializeMenuComponent

self menuComponent:(StMenuComponent new
  addEntry: 'All' withAction: [self showAllTasks];
  addEntry: 'Completed' withAction: [self showCompletedTasks];
  addEntry: 'Pending' withAction: [self showPendingTasks];
  addEntry: 'Missed' withAction: [self showMissedTasks];
  yourself).

StRootComponent>>#inititalizeMenuComponent

```

If you cannot resist to try the Web page now, you will surely get an error. You might already know the last missing method: **StRootComponent>>#renderTask:asRowOn:**. At the moment, we will try to keep it very simple and therefore just put date, name and description of our task into the table row:

```

renderTask: aTask asRowOn: html

html tableData: [html
  tableData: aTask deadline asString;
  tableData: aTask taskName;
  tableData: aTask taskDescription;
  tableData: aTask completed asString].

StRootComponent>>#renderTask:asRowOn:

```

But wait! Again, something is missing. Although everything works now, we do not have any data to present. Because input forms and database are covered in later chapters, by now we will only create some test data. Append the line **self listComponent items: self testTasks** to the method **StRootComponent>>#initializeListComponent**. Afterwards create the just used method:

```

testTasks

^ OrderedCollection
  with: (StTask new
    deadline: Date yesterday;
    completed: false;
    taskName: 'Missed task')
  with: (StTask new
    deadline: Date tomorrow;
    completed: false;
    taskName: 'Pending task')
  with: (StTask new
    deadline: Date tomorrow;
    completed: true;
    taskName: 'Already completed task')

StRootComponent>>#testTasks

```

ToDo-List

All Completed Pending Missed

| | |
|--------------------------------------|-------|
| 10 March 2008 Missed task | false |
| 12 March 2008 Already completed task | true |
| 12 March 2008 Pending task | false |

ToDo-List

All Completed Pending Missed

| |
|--------------------------------|
| 12 March 2008 Already complete |
|--------------------------------|

Figure 4.4: Final Root Component

In figure 4.4 you can see the final root component. In the left part all tasks are shown and in the right part the user has clicked on the completed link to show only the corresponding tasks.

Summary

Now you have a very simple working application using components and you know the most important concept in Seaside very well. In the following tutorial chapters you will create further components and beautify the ones created here. So keep on reading!

[Back](#)

5 - Forms

What you are going to learn

- [Form Elements](#)
- [Calling Components](#)
- [Summary](#)

Form Elements

Up to now, our ToDo Application is not capable of creating or removing tasks. For this we surely need HTML forms and, as you might expect, Seaside has very powerful tools for this and provides a wide range of available tags, objects and methods. Usually you open a form tag via `html form: []` and then insert as many input fields, labels, etc. as you like. Seaside allows you to bind input fields to the instance variables of your component, so hidden fields and special mapping are not needed - Seaside does this automatically.

The image shows a web form titled "Editing task". It contains the following elements:

- Name:** A text input field containing "ATask".
- Description:** A text area containing "My Description".
- Deadline:** A date picker showing "March", "11", and "2008".
- Completed:** A dropdown menu showing "no".
- Buttons:** "Save" and "Cancel" buttons at the bottom.

Figure 5.1: Task Editor

Let us get on with creating a simple component which will be responsible for collecting the input from the user. It can be used for editing existing tasks and adding new tasks. Figure 5.1 shows the final appearance of our task editor. Create a subclass of **WComponent** named **StTaskEditor** which has one instance variable **task**. Create accessors for the variable and initialize it with **#StTask new**. Remember to call **#super initialize** if you choose to put the initialization into the **#initialize** method. Now we can render our form (the defines CSS classes will be used later):

```
renderContentOn: html

html div
  class: 'generic lightbox';
  with: [
    html heading: 'Editing task'.
    html form: [
      html table: [
        html
```

```

        tableRow: [self renderNameInputOn: html];
        tableRow: [self renderDescriptionInputOn: html];
        tableRow: [self renderDateInputOn: html];
        tableRow: [self renderCompletedSelectionOn:
html];

        tableRow: [self renderButtonsOn: html]]]].

StTaskEditor>>#renderContentOn:

```

As you can see, it is very easy to create a form and render some content into it. We structure it in a table here to align the different labels, input fields and buttons. The latter are rendered in sub-methods. The first two, responsible for the name, the description, and the deadline of a task, are very similar:

```

renderNameInputOn: html

    html
        tableData: [html text: 'Name: '];
        tableData: [html textInput on: #taskName of: self task].

StTaskEditor>>#renderNameInputOn:

renderDescriptionInputOn: html

    html
        tableData: [html text: 'Description: '];
        tableData: [html textArea on: #taskDescription of: self task].

StTaskEditor>>#renderDescriptionInputOn:

renderDateInputOn: html

    html
        tableData: [html text: 'Deadline: '];
        tableData: [html dateInput on: #deadline of: self task].

StTaskEditor>>#renderDateInputOn:

```

After rendering a label, methods create some kind of text input field. Afterwards they bind the fields to an instance variable of the task. In fact, for every form input tag, a value can be set and a callback created, just like with a normal anchor. The callback is evaluated when the form is submitted. The **#on:of:** message just wraps this behavior for your convenience. It sends the provided selector to the given object to set the value of the input field and in the automatically created callback it uses the same selector plus **:** to set the new value. This way, the used instance variable (or its accessor methods) always contains the correct value from the input field.

```

renderCompletedSelectionOn: html

    html
        tableData: [html text: 'Completed: '];
        tableData: [
            html select
                add: true;
                add: false;
                on: #completed of: self task;
                labels: [:value |
                    value
                    ifTrue: ['yes']
                    ifFalse: ['no']]].

StTaskEditor>>#renderCompletedSelectionOn:

```

This method creates an html selection box. It does this by adding two values, which should be selectable. But as true and false cannot be rendered directly, a block can be set which converts every possible value to its corresponding label, like **yes** for true and **no** for false. The **#on:of:** message you already saw above connects the select box to the completed field of the task.

```
renderButtonsOn: html

html
  tableData;
  tableData: [
    html submitButton
      callback: [self answer: true];
      value: 'Save'.
    html submitButton
      callback: [self answer: false];
      value: 'Cancel'].

StTaskEditor>>#renderButtonsOn:
```

As the name of this method already states, it renders two buttons into our form. You can see that button instantiation is very easy. They also understand the **#callback:** message. The delivered block will be evaluated if the user clicks on the button and the form is submitted. But we do not know the message **#answer:** in those blocks. You will learn about its meaning in a few seconds, when we connect our new task editor component to the rest of our application.

Calling Components

To connect the listed tasks to our newly created task editor we have to add an instance variable **taskEditor** to our **StRootComponent** and create the accessors for it. In addition, it should be lazy initialized with an empty instance of **StTaskEditor**. Of course, we could create a new instance for every edit request and thus get rid of the variable, but this would be an unnecessary overhead of object creation.

Back in our **StRootComponent>>#renderTask:asRowOn:** method, we now can add another table cell with a simple anchor for editing a task:

```
renderTask: aTask asRowOn: html

"...
  tableData: [html anchor
    callback: [self editTask: aTask];
    with: 'edit']].

StRootComponent>>#taskEditor

editTask: aTask

self taskEditor task: aTask copy.
(self call: self taskEditor)
  ifTrue: [aTask copyFrom: self taskEditor task].

StRootComponent>>#editTask:
```

As you can see, **#editTask:** does some interesting things. It contains one of the most important methods of Seaside and demonstrates its beauty: **#call:**. Figure 5.2 illustrate the following description and the correlation between the source code and a view in the Web browser. Every component can *call* other components with its callbacks of anchors, form buttons, etc. This means that if the callback is evaluated, the current component known under **self** is replaced with the content of the called component on the Web page. In our example it looks like this: if you click on the edit link of any task, the whole **StRootComponent** is replaced with our task editor. This way you can easily switch to the editor form in a simple callback.

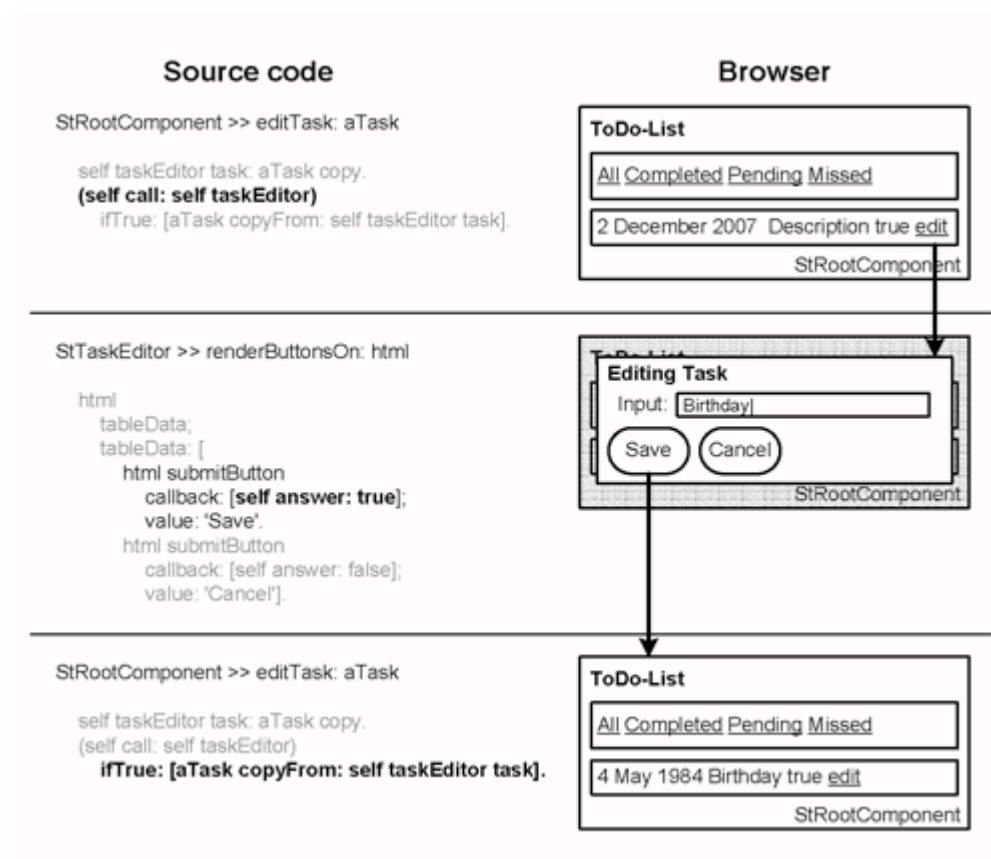


Figure 5.2: Call and Answer

Even better, Seaside remembers the position in your code and runs it from there again, after the called component finished. This means, **#call:** blocks your current component, that is your callback evaluation, waits for the called component to be finished, and afterwards goes on with your callback. As you may have already considered, the message **#answer:** which we used above in the task editor does exactly this finishing of a component. You can even send an answer object with it. In our case that would be `true` and `false` for *Save* and *Cancel* respectively. This answer is returned by **#call:** and you can evaluate it, just as we did. So, if the task editor was closed with save, `true` is returned and we can copy the data of the temporary task edited to the original one.

Let us use this example once again for creating a new task with the same editor. For this, we create a simple new method which we will call from our menu later:

```

createNewTask

| answer |
answer := self call: self taskEditor newTask.
answer ifTrue: [self listComponent items add: self taskEditor task].

StRootComponent>>#createNewTask
  
```

Here we assign the return value of our task editor explicitly to a temporary variable and test it afterwards for `true`. If so, we simply add the new task to our list component. Of course, at the moment these new items will not be saved after leaving the page, for example when starting a new session. You will learn how to do that later.

The **#newTask** method of our task editor is quite simple:

```

newTask

self task: StTask new.

StTaskEditor>>#newTask
  
```

Finally, we add a new link to our menu that calls the **#createNewTask** method. That is done within the method **StRootComponent>>#initializeMenuComponent**:

```
initializeMenuComponent

self menuComponent: (StMenuComponent new
  addEntry: 'All' withAction: [self showAllTasks];
  addEntry: 'Completed' withAction: [self showCompletedTasks];
  addEntry: 'Pending' withAction: [self showPendingTasks];
  addEntry: 'Missed' withAction: [self showMissedTasks];
  addEntry: 'New Task' withAction: [self createNewTask];
  yourself).

StRootComponent>>#initializeMenuComponent
```

Summary

Here two fundamental concepts of Seaside have been explained. On the one hand what do forms look like and how are they rendered, on the other hand the more important concept of **call and answer**. Components call other components and wait until these have finished their execution to return an answer. Using this concept it is easy to connect components with each other, to reuse them and to describe workflows (the topic of the next chapter). With the user management of our ToDo Application the session and task concept of Seaside is shown there as well.

[Back](#)

6 - Tasks And Sessions

What you are going to learn

- [Create Your Own Session](#)
- [User Login as a Task](#)
- [Login Component](#)
- [Register Component](#)
- [Message Component](#)
- [Summary](#)

Create Your Own Session

Every time a user accesses the application, a new instance of a session class is created. It exists as long as the user is interacting with the application, and the session has not yet expired. The application can store user-related data in the session, which then can be used by other components. The default session class is **WASession**.

The ToDo Application could store the user object in the session, for instance to determine if a user is logged in, or to access user data from other components. So, create a subclass of **WASession** called **StSession** with the instance variable **user** and corresponding accessors. You can then define some methods related to the login process:

```
login: aUser

    self user: aUser.

StSession>>#login:

logout

    self user: nil.

StSession>>#logout

isLoggedIn

    ^ self user isNil not

StSession>>#isLoggedIn
```

To be able to access your own session you have to modify the entry point to your application in the Seaside configuration (see chapter "First Steps"), overriding the **Session Class** (figure 6.1).

Configuration

General

| | | |
|-----------------------------|------------------------|--------------------------------------|
| Deployment Mode | false | override |
| Error Handler | WAWalkbackErrorHandler | override |
| Main Class | WARenderLoopMain | override |
| Redirect Continuation Class | WARedirectContinuation | override |
| Redirect Handler | WARedirectHandler | override |
| Render Continuation Class | WARenderContinuation | override |
| Root Component | StRootComponent | clear |
| Session Class | StSession | revert to: WASession |
| Session Expiry Seconds | 600 | override |
| Use Session Cookie | false | override |

Figure 6.1: Config Session

You can now access your session object by just calling `self session`, like our `#go` method does (see below for more details):

Other components can also use the session data. Additionally the method `StRootComponent>>#renderContentOn:` was extended with a div and some CSS classes for later use.

```
renderContentOn: html

html div
  class: 'generic';
  with:[
    html heading: 'ToDo-List of ', self session user userName.
    html div
      class: 'menu';
      with: self menuComponent.
    html div
      class: 'list';
      id: 'list';
      with: self listComponent].
```

`StRootComponent>>#renderContentOn:`

User Login as a Task

For our ToDo Application it is necessary that a user can be identified by an email address and a password. To simplify the login process we can now use a task.

A task is a specialized component that specifies a workflow but it is not able to render HTML content. Instead, a task is able to call a component and process the component's final state when it is finished. For example, we could create a task as subclass of **WATask** which first calls a login component (**StLoginComponent**). If this component answers that the login was successful we can call a component which displays personal content of the logged-in user.

So let us create a class **StRootTask** as subclass of **WATask** (**WATask** is a subclass of **WAComponent**) and change the **root component** of our ToDo Application to this class (see chapter "First Steps"); do not forget to add the **#canBeRoot** class method.

*At this point we refactored a bit and changed the name of **StRootComponent** to*

StLoggedInComponent. *We thus prevent possible confusion between **StRootTask** and **StRootComponent**. Furthermore, **StRootComponent** is no longer a root of our application because a login mechanism will be integrated in front of the task management. For this reason, the **#canBeRoot** method can be removed from the class side.*

Because a task does not render HTML directly but instead calls other components, you can think of a task as a process. The key method for a **WATask** is **#go**, which is invoked as soon as a task is displayed. So we need to create such a method and declare the main behavior of our application. At first, we want to display a login screen, which has a form containing input fields for email and password, and a link for registering new users. So we could create a login component and call it from within the task. While this sounds easy, we still have to consider what the possible answers of the login component are and how the task should behave.

The first scenario would be that no user is registered and someone clicks on the registration link. So the login component would just answer **#registerUser**. The task could then call another component which displays a registration form. Let us call this component **StRegisterComponent**. It could answer the newly created and registered user, or just **nil** if a user canceled the registration process.

The second scenario would be that someone tries to login with user credentials, so the login process could succeed or fail. If it fails, the login component would just not answer anything, but display an error message instead. These are details of the login component we will discuss later. If the login was successful, the login component just answers the user object found in the database corresponding to the user credentials.

So what should our task finally do? If no user is answered from either the login component or the registration component the task would just repeat the whole process, by processing its **#go** method again. Instead, if a user is answered by one of the called components, the task should do two things: store the user object in the current session and call a logged-in component, which displays the user's private tasks.

After you created a subclass of **WATask** called **StRootTask** and changed the **root component** of our ToDo Application you have to create a **#go** method which specifies the process above. The method could look like this:

```
go

| loginAnswer user |
loginAnswer := self call: StLoginComponent new.
loginAnswer = #registerUser
    ifTrue: [user := self call: StRegisterComponent new]
    ifFalse: [user := loginAnswer].
user ifNotNil: [
    self session login: user.
    self call: StLoggedInComponent new].

StRootTask>>#go
```

We will now discuss the several components which are called by our task.

Login Component

The **StLoginComponent** has to display input fields for user credentials, a link for registering new users and maybe some error messages if the login failed. It also has to verify entered login data. Figure 6.2 shows a final version of the **StLoginComponent**.

The screenshot shows a web form within a brown border. At the top, the title 'ToDo Application' is displayed in a large, bold, brown font. Below the title, the text 'Please login with email and password:' is shown in a smaller brown font. Underneath this text are two adjacent white input fields with thin grey borders. To the right of the second input field is a grey button with the word 'Login' in white text. Below the input fields and button is a brown underlined link that reads 'Sign up for the ToDo Application'.

Figure 6.2: Login Form

What we need now are a new component (**StLoginComponent**) and two instance variables **email** and **password** for the input fields, and corresponding accessors. The next step is to create the view, so create a **#renderContentOn:** method, add a form, input fields, a button and a link. The method looks like this:

```
renderContentOn: html

html div
class: 'generic';
with: [
    html
        heading: self applicationName;
        render: self messageComponent;
        text: 'Please login with e-mail and password:';
        form: [
            self
                renderTextInputOn: html;
                renderPasswordInputOn: html;
                renderLoginButtonOn: html;
                renderSignUpAnchorOn: html]]].
```

StLoginComponent>>#renderContentOn:

We want to show error messages using an encapsulating **StMessageComponent**. Create an instance variable **messageComponent** with accessors and make a lazy initialize with such a component. **StMessageComponent** will be explained below.

The **#applicationName** method simply returns a string with the name of the application.

```
applicationName

^ 'ToDo Application'

StLoginComponent>>#applicationName
```

The rendering methods for the input fields are the following:

```
renderTextInputOn: html

html textInput
    on: #email of: self;
    value: ''.
html space.

StLoginComponent>>#renderTextInputOn:
```

```
renderPasswordInputOn: html

    html passwordInput
        callback: [:value | self password: (self hashPassword: value)];
        value: ''.

        StLoginComponent>>#renderPasswordInputOn:

renderLoginButtonOn: html

    html submitButton
        callback: [self validateLogin];
        text: 'Login'.

        StLoginComponent>>#renderLoginButtonOn:

renderSignUpAnchorOn: html

    html paragraph with: [html anchor
        callback: [self registerUser];
        with: [html text: 'Sign up for the ', self applicationName]].

        StLoginComponent>>#renderSignUpAnchorOn:
```

Again, we can see that the input fields use the **#on:of:** method to set values and use callbacks if major processing is required.

Because of security reasons we do not want to store the user's password in plain text and decide to use an **Secure Hash Algorithm** (SHA) encryption algorithm. The value of the input field is hashed (a **LargePositiveInteger**) and saved in encrypted form in the instance variable **password**. This is done when processing the callbacks of **#renderPasswordInputOn:**. It hashes the current value using **#hashPassword:** and stores in the user object.

```
hashPassword: aString

    aString
        ifEmpty: [^ 0]
        ifNotEmpty: [^ SecureHashAlgorithm new hashMessage: aString].

        StLoginComponent>>#hashPassword:
```

The cryptographic capabilities are provided by the standard class **SecureHashAlgorithm**. If you need other algorithms than SHA have a look at the **Cryptography** package available via the SqueakMap Package Loader. Besides SHA, other encryption algorithms like DES, RSA128 or MD5 are implemented. Hashing using other algorithms works quite similar; simply change **SecureHashAlgorithm** to **MD5** for instance.

When the user inserts his credentials and clicks the *Login* button, the **#validateLogin** method is processed via the callback.

```
validateLogin

    | user |
    user := self session findUserByEmail: self email.
    (user notNil and: [user password = self password])
        ifTrue: [self answer: user]
        ifFalse: [self loginFailed].

        StLoginComponent>>#validateLogin
```

The method has to validate the credentials given by the user. It queries the current session to determine whether or not there is a user with the given email address. Therefore the session requires access to the database as explained in chapter "Persistence". If a user with the given email address can be found and its password can be

validated, the user object is answered to the task. The task can continue processing its **#go** method. If no user was found or the login could not be validated, an error message is shown. That is done by **#loginFailed** which sets an error message within the **StMessageComponent**.

```
loginFailed
```

```
self messageComponent errorMessage: 'Login failed.'.
```

```
StLoginComponent>>#loginFailed
```

As shown above, the registration link calls **#registerUser** in its callback. The method just answers the symbol **#registerUser** which terminates the processing of the login component and continues with the task's **#go** method which can now call the registration component.

```
registerUser
```

```
self answer: #registerUser.
```

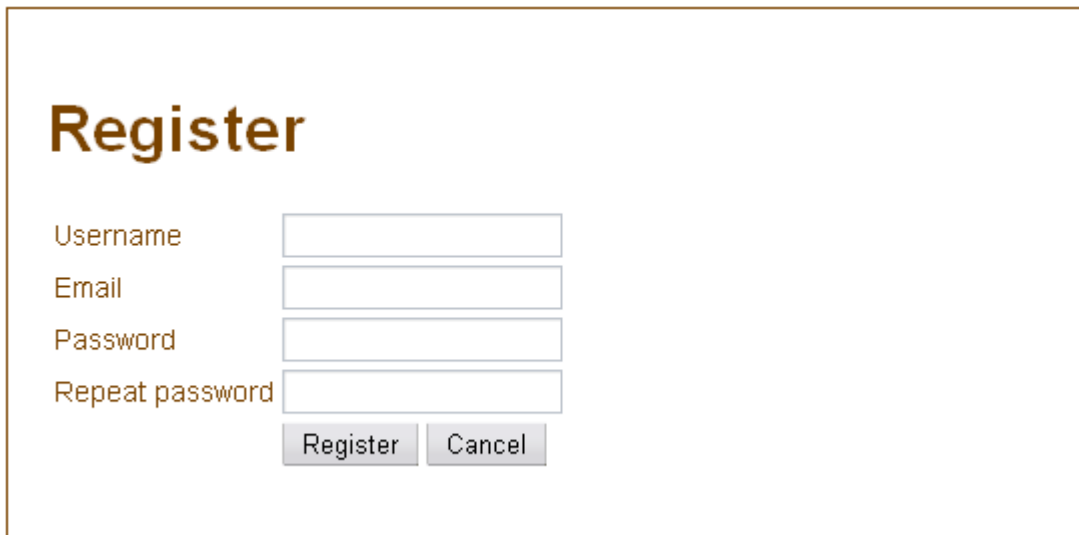
```
StLoginComponent>>#registerUser
```

Having finished editing their tasks, users should then be able to log out. Therefore, we add a link *Logout* to the menu component by adding `addEntry: 'Logout' withAction: [self session logout. self answer: true];` at the end of **StLoggedInComponent>>#initializeMenuComponent**.

If you try to execute the application you can see that many features are not working. We have insert many new methods which are needed for the next parts especially the persistence chapter. So keep on reading.

Register Component

The registration component displays a form field (figure 6.3), which contains input fields for user name, email, password and password repetition. If all data are valid and the two passwords are the same, a new user will be created, stored in the database and answered to our task, so the user can automatically be logged in.



The image shows a web form titled "Register" in a large, bold, brown font. Below the title, there are four input fields, each with a label to its left: "Username", "Email", "Password", and "Repeat password". The labels are in a smaller, brown font. The input fields are white with a thin blue border. Below the input fields, there are two buttons: "Register" and "Cancel". The buttons are light gray with a slight shadow and rounded corners. The entire form is enclosed in a thin brown border.

Figure 6.3: Register Form

Initially, you need the a new component **StRegisterComponent** and some instance variables and their accessors, namely: **user** and **repeatedPassword**. We decided to store the data entered in the input fields in a user object, so just the password repetition requires a dedicated instance variable. Please initialize **user** with an empty user object (**StUser**). The next step is to build the view. That is simple: we just need four input fields, a submit button and a cancel link.

```
renderContentOn: html
```

```

html div
  class: 'generic';
  with: [
    html
      heading: 'Register';
      render: self messageComponent;
      form: [
        html table: [
          self
            renderUsernameTextInputOn: html;
            renderEmailTextInputOn: html;
            renderPasswordTextInputOn: html;
            renderRepeatedPasswordTextInputOn: html.
        html tableRow: [
          html
            tableData;
            tableData: [
              self renderSubmitButtonOn: html.
              html space.
              self renderCancelButtonOn:
html]]]]].

```

StRegisterComponent>>#renderContentOn:

As you can see we are using the **StMessageComponent** to display errors again, for example if the user forgot to fill in a field, so do not forget to create the instance variable **messageComponent** with its accessors and a lazy initialization with an **StMessageComponent**.

The rendering methods for the register and cancel link are quite simple:

```
renderSubmitButtonOn: html
```

```

html submitButton
  callback: [self registerUser];
  text: 'Register'.

```

StRegisterComponent>>#renderSubmitButtonOn:

```
renderCancelButtonOn: html
```

```

html submitButton
  callback: [self answer: nil];
  text: 'Cancel'.

```

StRegisterComponent>>#renderCancelButtonOn:

The rendering methods all create a table row with the caption in the one column and the input field in the other one.

```
renderUsernameTextInputOn: html
```

```

html tableRow: [
  html
    tableData: 'Username';
    tableData: [
      html textInput
        callback: [:value | self user userName: value];

```

```

        value: self user userName]].

        StRegisterComponent>>#renderUsernameTextInputOn:

renderEmailTextInputOn: html

    html tableRow: [
        html
            tableData: 'E-mail';
            tableData: [
                html textInput
                    callback: [:value | self user email: value];
                    value: self user email]].

        StRegisterComponent>>#renderEmailTextInputOn:

renderPasswordTextInputOn: html

    html tableRow: [
        html
            tableData: 'Password';
            tableData: [
                html passwordInput
                    callback: [:value | self user password: (self
hashPassword: value)]]].

        StRegisterComponent>>#renderPasswordTextInputOn:

renderRepeatedPasswordTextInputOn: html

    html tableRow: [
        html
            tableData: 'Repeat password';
            tableData: [
                html passwordInput
                    callback: [:value | self repeatedPassword: (self
hashPassword: value)]]].

        StRegisterComponent>>#renderRepeatedPasswordTextInputOn:

```

Since password must be hashed again, the **StRegisterComponent** also needs the **#hashPassword:**:

```

hashPassword: aString

    aString
        isEmpty: [^ 0]
        ifNotEmpty: [^ SecureHashAlgorithm new hashMessage: aString].

        StRegisterComponent>>#hashPassword:

```

When a user clicks the *Register* button, **#registerUser** is invoked, which first checks whether all input fields are filled correctly and the two password fields match. If the input of some field is wrong, we show an error message; but if all fields are valid, the value of the instance variable **#user** is stored in the database. The same user object is answered to our task. So then it can automatically log in the new user.

```

registerUser

    self user userName isEmptyOrNil
        ifTrue: [^ self messageComponent infoMessage: 'Please choose a

```



```

username!'].
    self user email isEmptyOrNil
        ifTrue: [^ self messageComponent infoMessage: 'Please enter your
e-mail address!'].
    (self session findUserByEmail: self user email)
        ifNotNil: [^ self messageComponent errorMessage: 'The e-mail
address is already registered!'].
    self user password = 0
        ifTrue: [^ self messageComponent infoMessage: 'Please choose a
password!'].
    self user password = self repeatedPassword
        ifFalse: [^ self messageComponent infoMessage: 'Your repeated
password does not match!'].
    self session database addUser: self user.
    self answer: self user.

```

StRegisterComponent>>#registerUser

When clicking the cancel link, **nil** is answered to our task, so the task can display the login component again.

Message Component

The **StMessageComponent** is a helper component that displays different types of messages. It should encapsulate this functionality from all other components. As it is a component, other components can simply call `html render: self messageComponent`.

The **StMessageComponent** needs three instance variables with accessors for the message (**messageString**), the type of the message (**messageType**), and a boolean indicating weather the message has been shown or not (**wasShown**, initially set to **true**).

Later on, we want to use different styles for the messages. Therefore, we add CSS classes depending on the message's type. The style sheets will be added in chapter "Resources"

The messages **#infoMessage:** and **#errorMessage:** set the message string and the correct style. They also set **wasShown** to **false** so that the message gets rendered.

```

errorMessage: aString

self
    messageString: aString;
    messageType: 'error';
    wasShown: false.

StMessageComponent>>#errorMessage:

infoMessage: aString

self
    messageString: aString;
    messageType: 'info';
    wasShown: false.

StMessageComponent>>#infoMessage:

```

Now we define the rendering method. It should only generate output if the message was not yet shown. Depending on the current **messageType**, the corresponding CSS class should be used.

```

renderContentOn: html

```

```
self wasShown ifTrue: [^ self].  
html div  
  class: self messageType, 'Message';  
  with: self messageString.  
  
StMessageComponent>>#renderContentOn:
```

Summary

You have seen how to access session data in the whole Web application. This we have used afterwards in our user login and with its help we could then explain the task concept. It is easy to use and describes workflows in a single method. The next chapter deals with external resources, like images, text files etc; what features does Seaside offer and which of them should be used when?

[Back](#)

7 - External Resources

What you are going to learn

- [Introduction](#)
- [The Simple Way of Life](#)
- [WAFileLibrary](#)
- [StToDoLibrary](#)
- [External Directory](#)
- [Summary](#)

Introduction

In this chapter, you will learn about different methods to access external resources like images, scripts or other media. We start with a very simple and Squeak-like variant. Afterwards we directly dig into providing plain files and directories as this is a much more efficient way. This approach will be integrated into our ToDo Application. Finally, a combination of standard Web servers like Apache and Seaside will be presented.

The Simple Way of Life

The main external resources you will use are images. Seaside offers you a very simple way to incorporate them into your Web site:

```
renderContentOn: html

    html image
        form: (Form fromFileName: 'image.png').
```

However, this code has several flaws. The first one is that the image will be loaded time and again with each incoming request. While this can be circumvented by implementing a simple image pool or cache, another problem is that images cannot be cached for different sessions by the browser.

The simple way for incorporating CSS and JavaScript code has already been discussed in chapter "First Steps". You can simply alter the **#style** and the **#script** method of your component:

```
style

    ^ 'h1 { color: yellow; }'

script

    ^ 'alert("Hello World");'
```

But as already stated previously this is not the recommended way, because it spreads the CSS and JavaScript code over all of your different components. Additionally, every **#script** and **#style** method results in one more HTML header line requesting that special part of the CSS or JavaScript. This means that the browser has to send many different URL requests to our server, thus increasing load. Furthermore, URLs generated for the script parts are different for each page load and therefore cannot be cached by the server or the browser. Another reason for not using these two methods is convenience for the designers. They are used to working with plain files and surely do not want to stare at Smalltalk code.

WFileLibrary

A **WFileLibrary** allows us to serve static files directly from Seaside without the need for a stand-alone server like Apache. These files can even reference each other - for example CSS using images - and can be distributed the same way as normal Smalltalk code via Monticello and friends. To use a **WFileLibrary**, you first have to subclass it. For convenience, we will name this subclass **MyFileLibrary**. Your real files get into Seaside via two ways: the first one is programmatically by using the **MyFileLibrary** class methods **#addAllFilesIn:** or **#addFileNamed:**. For example:

```
MyFileLibrary addAllFilesIn: '/path/to/directory/with/files'.
MyFileLibrary addFileNamed: '/path/to/background.png'.
```

The other way is via the Web interface. First, you need a **files** entry point in your Seaside configuration. This can be created by selecting *Files* in the type field of the *add entry point* dialog in the Seaside configuration. Usually, **files** is chosen as the name for the entry point. This entry point is always needed if you want to use a file library, even if you do not want to use the Web interface. To upload files now, you have to browse to your Seaside configuration application and click on *configure* for your files entry point. Then choose *configure* behind **MyFileLibrary** and afterwards you can upload files to your server (figure 7.1).

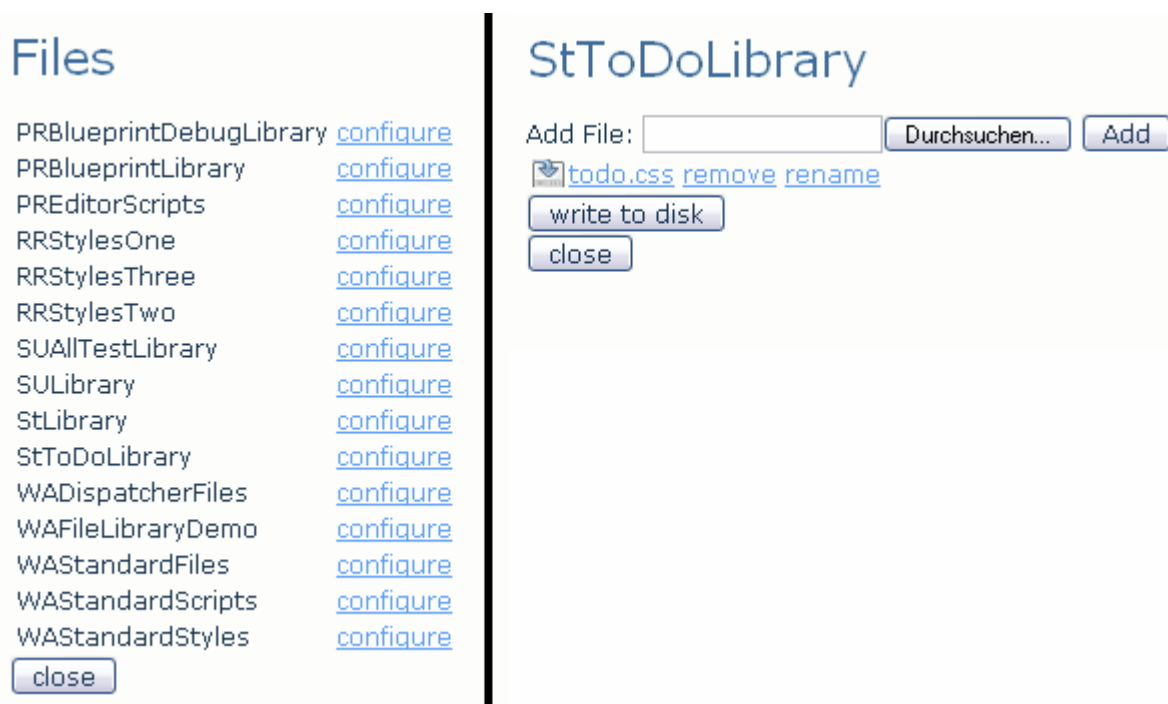


Figure 7.1: Seaside File Library Configuration

Each file in the file library is represented by a method. The method name is created from the file name, the dot is removed and the first letter of the suffix is capitalized; for instance **main.css** becomes **mainCss**. Each file gets its own human-readable URL, so they can be integrated into your application the same way static files would be normally integrated into a Web page. They have a constant path that is **/seaside/<Static File Library>/<FileLibrary class name>/<filename>** for example, **/hirschfeld/seaside/files/MyFileLibrary/background.png**. These can be conveniently generated by

```
MyFileLibrary / #aSelector.
```

where **#aSelector** is the name of the method representing that file. Images can therefore be integrated very easily:

```
renderContentOn: html

html image
  url: MyFileLibrary / #backgroundPng.
```

But can we use our style sheets and JavaScripts? For this, you can hook into the creation of the HTML header of your seaside Web pages. While rendering your Web page, the method **#updateRoot:** is called for every visible component. If you override this method in your own component, you have the ability to add scripts, style sheets and other things. An example:

```
updateRoot: anHtmlRoot

    super updateRoot: anHtmlRoot.
    anHtmlRoot stylesheet
        url: MyFileLibrary / #mainCss.
    anHtmlRoot javascript
        url: MyFileLibrary / #mainJs.

MyFileLibrary>>#updateRoot:
```

You can also override the method **#selectorsToInclude**. It returns an array with selectors, that should be included automatically. Alternatively to the listing above you can write:

```
selectorsToInclude

    ^ Array with: 'mainCss' with: 'mainJs'

MyFileLibrary>>#selectorsToInclude
```

Despite the obvious benefits, using file libraries also has its disadvantages. As you may have noticed, the methods in the **WFileLibrary** representing static files on disk cannot be automatically updated when one of the original files changes. This means that you have to re-read changed files manually. Another problem is the size of binary (and some times even big text data) in the image. Binary data are converted to the programmatic definition of a **ByteArray** containing all byte values. This means that the source code for such a binary file is much larger than the original file, and additionally, that the compiled method is large, too. This blows up your image size and wastes memory on your machine.

StToDoLibrary

In chapter "Tasks and Sessions - Message Component" we have defined CSS classes for the different types of messages, the **StMessageComponent** should display. There are some other global styles that should be kept in a central library. So we create a new class **StToDoLibrary** which inherits from **WFileLibrary** and add some style definitions:

```
todoCss

    ^ ,
body {
    font: normal 100% Arial, Helvetica, Verdana, sans-serif;
}

.generic {
    font-family: Geneva, Arial, Helvetica, sans-serif;
    font-style: normal;
    font-weight: normal;
    font-size: 14px;
    color : #663300;
    padding: 20px;
    margin-top: 20px;
    margin-left: 20px;
    border: 1px solid #663300;
    width: 500px;
```

```

}

.lightbox {
    background-color: #FFFFFF;
}

.errorMessage {
    background-color: #ffb0b0;
    border: 1px solid #ee0000;
    padding: 5px;
    margin-bottom: 10px;
}

.infoMessage {
    background-color: #fbffcc;
    border: 1px solid #eda33a;
    padding: 5px;
    margin-bottom: 10px;
}

.copyright {
    font-family: Geneva, Arial, Helvetica, sans-serif;
    font-size: 10px;
    font-style: normal;
    font-weight: normal;
    color: #663300;
    text-align: left;
    margin-left: 20px;
}

a {
    color : #663300;
    text-decoration: underline;
}

```

StToDoLibrary>>#todoCss

The **StToDoLibrary** might also be a good place to keep the application's name as it is a globally available resource store. So we add a class side method

```
applicationName
```

```
^ 'ToDo Application'
```

StToDoLibrary class>>#applicationName

Remember to rewrite **StLoginComponent>>#applicationName** to

```
applicationName
```

```
^ StToDoLibrary applicationName
```

StLoginComponent>>#applicationName

Finally include the style sheet by overriding **#selectorsToInclude**:

```
selectorsToInclude
```

```
^ # ( todoCss )
```

```
StToDoLibrary>>#selectorsToInclude
```

Remember to add the **StToDoLibrary** to the libraries of your application. In chapter "First Steps" we have described how to configure. To add a library go to the preference page and have a look at the top of the Web site (**Libraries**). Here you can select the **StToDoLibrary** and add it to the web application.

It would be also nice if the title of our ToDo Application appears in the page title. Since a task is not able to change it, that must be done within the session's **#updateRoot::**:

```
updateRoot: anHtmlRoot
```

```
super updateRoot: anHtmlRoot.
```

```
anHtmlRoot title: StToDoLibrary applicationName.
```

```
StSession>>#updateRoot:
```

External Directory

The favorite method for storing static content like images, CSS, and JavaScript files or other media files is the external directory. That includes setting up a **normal** Web server along with your Seaside installation. This could be on another port but usually the standard Web server is used as a proxy for Seaside. This means that Web site requests are forwarded to Seaside, while others are handled by Apache. The result is the painless usage of relative URLs like **/images/arrow.png** in your Seaside code. The reason for preferring this method is the ability to simply alter static CSS and JavaScript in files instead of modifying Smalltalk code.

As an example, consider an Apache Web server running on standard port 80. Now, you want to run a Seaside application, the ToDo Application, too. To combine both servers, start Seaside on another port, say, 8080, and proxy special URL requests to it. The code needed in the Apache configuration file would look similar to this:

```
ProxyPass /directory/todo http://localhost:8080/todo/
<Location /directory/todo/>
    ProxyPassReverse /todo/
</Location>
```

The Apache must have loaded the module `mod_proxy` to enable its proxy facilities.

As most Seaside applications generate some absolute links, you also have to change your applications configuration. Open the configurations page for your application and go to the server configuration. By default, no *Resource Base URL*, *Server Hostname*, and *Server Path* are specified. But if you are proxying your image, you have to specify this information. This means that you have to fill in a server name that is reachable from the entire network, **www.example.org** for instance. As *Server Path* you have to put the absolute path to your application. In the most cases it is the path you used in the ProxyPass directive, **/directoy/squeaksource** in the example above.

In some cases your application may generate absolute URLs that point to paths of your seaside server and that are not accessible from the entire Internet (for example when using a File Library). If you want to rewrite them within a generated document, refer to an Apache module like [mod_line_edit](#).

Another possibility is to make use of the Resource Base URL. Specify the URL to a directory that is served from your Web server, **http://www.example.org/some/** for example. Then use the **#resourceUrl:** message:

```
html anchor
    resourceUrl: 'directory/documentation.pdf';
    with: 'Read the documentation'.
```

```
#resourceUrl:-Example
```

This will generate a link to **http://www.example.org/some/directory/documentation.pdf**.

Unfortunately, proxying the Seaside Web server (**KomHttp**) through another Web server like Apache costs some performance. Although there are some experiments with FastCGI and Seaside 2.9 will eventually contain a FastCGI module, the proxy solution allows for load balancing. One recommended tactic is to start multiple Squeak images behind an Apache server to accomplish successful load balancing and to take advantage of the full abilities of multiple processors and cores.

Summary

As you have seen, every method for integrating external resources into your Seaside Web application has its own advantages and disadvantages. There are easy ways, which sadly do not scale as the application grows. There are sophisticated methods, like proxying with Apache, which are hard to set up and unnecessary for small applications. Also, be aware of the necessity to install another version control system outside Squeak, if you are working in a team and using external resources. Monticello is not capable of this.

We think it is very useful to start with the smaller and easier solutions and slowly move towards the larger and more complicated solutions when you really need to (Apache proxy for instance). The simple way may be a good starting point and allow for fast development. However, for images you need some other way to transfer them to your working team than Monticello, and style sheets and scripts can end up scattered over your components very quickly. If you find this problem in your code, go on to the next stage and use a file library. If this solution starts to give you and your team members headaches and sleepless nights, move on to the sophisticated Apache way.

In the next chapter you will see which alternatives you have to store Smalltalk objects persistently.

[Back](#)

8 - Persistence

What you are going to learn

- [Introduction](#)
- [Saving All Data in the Image](#)
- [Saving All Data in a Relational Database: Glorp](#)
- [Saving All Data in an Object-Oriented Database: GOODS](#)
- [Saving All Data in an Object-Oriented Database: Magma](#)
- [One Database Connection per Session](#)
- [Summary](#)

Introduction

This chapter is all about persistence. Whenever you want your application to save data for later use, you have to deal with persistence.

In the following, the four main options to save your data persistently are described.

Saving All Data in the Image

Saving all data in the image is an option that works for a single instance of your application. If you have more than one instance running, maybe for load sharing, you have to think about sharing your data between the running applications.

So let us assume you have one running image. What you could do now is create a kind of database class and store all your data in class variables (for example all your registered users). So create a subclass of **Object** named **StImageDatabase** and add a class variable **Users** with class side accessors. The getter should do a lazy initialize with an empty **OrderedCollection**:

```
users

^ Users ifNil: [Users := OrderedCollection new]

StImageDatabase class>>#users
```

But that is not all! If your image crashes for whatever reason, your data would be lost. So, every time you save data in your image you have to save your whole image on disk. Surely you could as well save the image in fixed time intervals, but then you have to deal with possible data loss. To save your image, you could execute a method like:

```
saveImageWithoutMonitor

SmalltalkImage current saveSession.

StImageDatabase>>#saveImageWithoutMonitor
```

Because a Web application might have several users accessing the application simultaneously who may, moreover, be causing the image to be saved at the same time, you need to introduce a mutex as another class variable. The lazy initialize is the following:

```
writeMutex
```

```
^ WriteMutex ifNil: [WriteMutex := Monitor new]
```

```
StImageDatabase class>>#writeMutex
```

With this mutex you can modify the image saving method like below to be sure that the image saving process is only running once at a specific time:

```
saveImage
```

```
self class writeMutex critical: [self saveImageWithoutMonitor].
```

```
StImageDatabase>>#saveImage
```

All you have to do is call that image saving method whenever you are going to add or delete data (a user for instance).

```
addUser: aUser
```

```
self class users add: aUser.
```

```
self saveImage.
```

```
^ aUser
```

```
StImageDatabase>>#addUser:
```

When a user adds a new task to its list, the method **#addTask:toUser:** is called. For the image database, it is quite simple:

```
addTask: aTask toUser: aUser
```

```
aUser addTask: aTask.
```

```
self saveImage.
```

```
^ aTask
```

```
StImageDatabase>>#addTask:toUser:
```

We have to implement a **#findUserByEmail:** that returns the user with the supplied email address or nil if none was found:

```
findUserByEmail: anEmail
```

```
^ self class users
```

```
detect: [:each | each email = anEmail]
```

```
ifNone: [nil]
```

```
StImageDatabase>>#findUserByEmail:
```

For compatibility reasons with the other presented persistence solutions, we have to add the two methods **#connectionFailed** (indicating whether the connection to the database succeeded), **#disconnect** (it is called to close the connection to the database), and **#save:** (saving the given object in the database).

```
connectionFailed
```

```
^ false
```

```
StImageDatabase>>#connectionFailed
```

```
disconnect
```

```
StImageDatabase>>#disconnect
```

```
save: anObject
```

```
StImageDatabase>>#save:
```

This works pretty well, but saving an image causes a massive amount of disk writes and your application slows down during that writing process. Also, you are not able to share your saved data with other applications or another instance of your application for load balancing. So, let us look at better options.

Saving All Data in a Relational Database: Glorp

One such option is to use an **O/R mapper** and a relational database as its back-end. With this option, it is possible to share your data between other applications. You also no longer have to worry about conflicts, which could occur if more than one dataset is trying to be saved at the same time. All the work is done by the relational database management system of your database server.

That is nearly almost true; you also have to invest some thorough work in your data models. Since you are programming in an object-oriented environment, you have to map your data to the relational view of the database. This is done using **GLORP** (Generic Lightweight Object-Relational Persistence).

In the following, we will use **PostgreSQL** as the relational database management system of choice.

Installing a PostgreSQL Server

If you do not have access to an existing PostgreSQL server, you have to install one on your machine. Installation files for Windows and Linux, you can find [here](#) and a detailed description about installing the server on a Mac OS X machine can be found on [Marc Liyanage's Web site](#).

After the database server has been installed, make sure the authentication mechanism of the database is using password and not md5! You can find the setting in **data/pg_hba.conf** in your PostgreSQL directory.

Now you can create a new database user called **postgres** with password **postgres** and a database **StDatabase**. Make sure that the encoding of your database is UTF-8.

Installing the GLORP Framework

To get GLORP working, two new packages, **GLORP Port** and **PostgreSQL Client for Squeak**, are required. Open the SqueakMap Package Loader via the World Menu and *open....* First install the package **PostgreSQL Client for Squeak** and then **GLORP port**. Installing GLORP takes some time, so do not worry.

The Error Message

If you get an error message during the installation asking you whether you would like to open a debugger, just click on **Yes**. Now the installation is paused. Please open the System Browser and go to the class method **#basicIsSqueak** of the **Dialect** class.

Edit this method, so it looks like this:

```
basicIsSqueak
    ^ true
    "^ (Smalltalk respondsTo: #vmVersion) and: [(Smalltalk vmVersion
copyFrom: 1 to: 6) = 'Squeak'.]"

Dialect>>#basicIsSqueak
```

Using GLORP

Using GLORP means mapping your model's attributes to data columns of the relational database. To achieve this, you first need to create a subclass of **DescriptorSystem** called **StGlorpDatabase**. Every model has its own database table containing all attributes. The connection between a user and its tasks is given through foreign key

relations, that means each object, **StUser** or **StTask** has its own id for identification and every task has also a foreign key, which is the id of its owner (user).

Let us look at the models you have created. For each model, you have to create a method like this:

```
tableForSTUSER: aTable

aTable
  createFieldNamed: 'id' type: platform sequence;
  createFieldNamed: 'userName' type: platform text;
  createFieldNamed: 'email' type: platform text;
  createFieldNamed: 'password' type: platform text.

(aTable fieldNamed: 'id') bePrimaryKey.

StGlorpDatabase>>#tableForSTUSER:
```

The naming convention is **#tableForTABLENAMEINUPPERCASE**. With these methods, you are specifying which tables and columns should be created in the relational database and which data types these columns should have. All available data types can be found in the method category *types* of the class **PostgreSQLPlatform**. Each column name and type should match an attribute of your model. The object itself is identified by its id. Since each row in a relational database has to be somehow unique, we choose to specify the id as primary key, so this id is unique for every table row.

Next, you have to specify the mapping between the model's attributes and the database tables' columns. For each model, create a descriptor that looks like this:

```
descriptorForStUser: description

| table |
table := self tableNamed: 'stuser'.
description table: table.
(description newMapping: DirectMapping)
  from: #id to: (table fieldNamed: 'id').
(description newMapping: DirectMapping)
  from: #userName to: (table fieldNamed: 'userName').
(description newMapping: DirectMapping)
  from: #email to: (table fieldNamed: 'email').
(description newMapping: DirectMapping)
  from: #password to: (table fieldNamed: 'password').
(description newMapping: OneToManyMapping)
  attributeName: #tasks;
  referenceClass: StTask;
  collectionType: OrderedCollection;
  orderBy: #id.

StGlorpDatabase>>#descriptorForStUser:
```

Above, you can see that most attributes are directly mapped to columns of the database table **stuser**. All attributes of the object are accessed through symbols, for example **#userName** accesses the instance variable `userName` of the user object. Be sure to create accessors for your instance variables. What you can also see is that we need a **OneToManyMapping** instead of a **DirectMapping** for the user's tasks. Every user has several tasks which are collected in the user object's **#tasks** instance variable. So, we need to define a **OneToManyMapping** of the attribute **#tasks**, which is an **OrderedCollection** containing instances of the referenceClass, here **StTask**. Additionally, we want to order the collection by task ids.

You now need a class model enumerating all attributes. Do not forget to tell the model that the attribute **#tasks** is a collection of **StTask** instances.

```
classModelStUser: model

    model
        newAttributeNamed: #id;
        newAttributeNamed: #userName;
        newAttributeNamed: #email;
        newAttributeNamed: #password;
        newAttributeNamed: #tasks collectionOf: StTask.

StGlorpDatabase>>#classModelStUser:
```

After having created the methods for **StUser**, we need to do the same for **StTask**. Let us start with the class model and database table layout:

```
classModelStTask: model

    model
        newAttributeNamed: #completed;
        newAttributeNamed: #deadline;
        newAttributeNamed: #taskDescription;
        newAttributeNamed: #id;
        newAttributeNamed: #taskName.

StGlorpDatabase>>#classModelStTask:

tableForSTTASK: aTable

    aTable
        createFieldNamed: 'id' type: platform sequence;
        createFieldNamed: 'completed' type: platform boolean;
        createFieldNamed: 'deadline' type: platform date;
        createFieldNamed: 'taskDescription' type: platform text;
        createFieldNamed: 'taskName' type: (platform varchar: 100);
        createFieldNamed: 'stuser_id' type: platform int4.

    (aTable fieldNamed: 'id') bePrimaryKey.
    aTable
        addForeignKeyFrom: (aTable fieldNamed: 'stuser_id')
        to: ((self tableNamed: 'stuser') fieldNamed: 'id').

StGlorpDatabase>>#tableForSTTASK:
```

The class model is quite straight-forward. The table layout is similar to the **StUser**'s one except the additional foreign key. We define that the value of **stuser_id** must reference a valid **id** within the users table.

The attribute mapping is very simple again:

```
descriptorForStTask: description

| table |
table := self tableNamed: 'sttask'.
description table: table.
(description newMapping: DirectMapping)
    from: #completed to: (table fieldNamed: 'completed').
(description newMapping: DirectMapping)
    from: #deadline to: (table fieldNamed: 'deadline').
(description newMapping: DirectMapping)
```

```

        from: #taskDescription to: (table fieldNamed:
'taskDescription').
        (description newMapping: DirectMapping)
        from: #id to: (table fieldNamed: 'id').
        (description newMapping: DirectMapping)
        from: #taskName to: (table fieldNamed: 'taskName').

```

StGlorpDatabase>>#descriptorForStTask:

Create the method **#allTableNames**, which should return a collection of all used tables:

```

allTableNames

^ #('stuser' 'sttask')

StGlorpDatabase>>#allTableNames

```

Create the method **#constructAllClasses**, which should return a collection of all used classes:

```

constructAllClasses

^ super constructAllClasses
  add: StUser;
  add: StTask;
  yourself

StGlorpDatabase>>#constructAllClasses

```

Now just add the instance variables **glorpSession**, **connectionFailed**, add their accessors. The **glorpSession** variable will contain the database connection as an instance of **GlorpSession**, and **connectionFailed** is a **Boolean**, which is true if anything is wrong with the database connection.

The next step is to overwrite **#initialize**. The platform we use is PostgreSQL, which we need to describe the datatypes of the database tables. We also want every new instance of our database class to be connected to the database, so call **#connect**.

```

initialize

super initialize.
self platform: PostgreSQLPlatform new.
self connect.

StGlorpDatabase>>#initialize

```

What does **#connect** do? This method creates at first a login object, containing user name, password, host, database name and our platform. The next step is to define an accessor for the database for the newly created login. **#glorpSession** is then set to an instance of **GlorpSession** which needs a **DescriptorSystem**, which in turn is our instance of **StGlorpDatabase** containing all necessary descriptions about our models. We also need some exception handling in case anything goes wrong. In case of an error while connecting to the database our **#connectionFailed** variable is true, otherwise, false.

```

connect

| accessor |
accessor := DatabaseAccessor forLogin: self createLogin.
self glorpSession: (GlorpSession forSystem: self).
self glorpSession accessor: accessor.
self connectionFailed: false.
[accessor login] ifError: [:err | self connectionFailed: true].

```

```
StGlorpDatabase>>#connect
```

```
createLogin
```

```
^ Login new
  database: platform;
  username: 'postgres';
  password: 'postgres';
  connectionString: 'localhost_STDatabase';
  yourself
```

```
StGlorpDatabase>>#createLogin
```

When we can connect we also need to be able to disconnect. If the current database connection has not timed out, we can simply log out.

```
disconnect
```

```
self glorpSession
  ifNotNil: [self glorpSession accessor logout].
```

```
StGlorpDatabase>>#disconnect
```

The next method does just one thing: it creates **PGSequence** objects representing the id sequence of each model. If anything is wrong, the error will be shown in the Transcript.

```
createAllSequences
```

```
self glorpSession system platform areSequencesExplicitlyCreated
  ifFalse: [^ self].
self glorpSession system allSequences do: [:each |
  self glorpSession accessor
    createSequence: each
    ifError: [:error | Transcript show: error messageText]].
```

```
StGlorpDatabase>>#createAllSequences
```

In the next step, we can finally create our tables. In the first loop, we create the tables with their names, columns and data types. Additionally, we create all the indexes we need, which allows the database to reorder the table rows, so that a row can be found faster if we search about an attribute whose column has been indexed. The second loop is to create foreign key relations between the tables. For example, the **sttask** table will have a foreign key, which is the id of the owner of the specific task, because one user can have several tasks.

```
createAllTables
```

```
| accessor errorBlock allTables |
accessor := self glorpSession accessor.
errorBlock := [:errorx | Transcript show: errorx messageText].
allTables := self glorpSession system allTables.
allTables
  do: [:each | accessor
    createTable: each ifError: errorBlock;
    createTableIndexes: each ifError: errorBlock];
  do: [:each | accessor createTableFKConstraints: each ifError:
errorBlock].
```

```
StGlorpDatabase>>#createAllTables
```

Finally, **#dropAllThenCreateSchema** will recreate all tables by deleting all existing tables and creating them

again.

```
dropAllThenCreateSchema

self glorpSession accessor dropTables: self allTables.
self
  createAllSequences;
  createAllTables.

StGlorpDatabase>>#dropAllThenCreateSchema
```

You can now open a workspace and execute

```
StGlorpDatabase new dropAllThenCreateSchema.
```

If your descriptions are fine all tables should be added. You can check this by using some of the PostgreSQL Tools, like PGAdmin

So you learned to create your schema for saving your objects. But how do you save objects? That is easy, all you have to do is register your objects in an unit of work:

```
addUser: aUser

self glorpSession
  inUnitOfWorkDo: [self glorpSession register: aUser].
^ aUser

StGlorpDatabase>>#addUser:
```

The **unit of work**-block commits all changes made during the block execution. **#Register** checks whether the object was read from the database. If it was read, only the changes are saved. If it was not read, the object is saved as a new object.

Adding a task to a user is simple now:

```
addTask: aTask toUser: aUser

self glorpSession
  inUnitOfWorkDo: [aUser addTask: aTask.
                  self glorpSession register: aUser].
^ aTask

StGlorpDatabase>>#addTask:toUser:
```

Now you can save objects. Reading objects is easy, too.

```
findUserByEmail: anEmailAddress

^ self glorpSession
  readOneOf: StUser where: [:each | each email = anEmailAddress]

StGlorpDatabase>>#findUserByEmail:
```

#readOneOf:where: reads the first object which is an instance of aClass and matches the where-block. More reading operations can be found in the *api/queries*-category of the **GlorpSession** class.

More Information about GLORP can be found on the [GLORP Web site](#).

Saving All Data in an Object-Oriented Database: GOODS

Another option to make your data persistent and share them with other applications is to use an object-oriented database like **GOODS** (Generic Object Oriented Database System).

Installing the GOODS Database

There are two ways to install GOODS: the first is to download a binary of the GOODS server, the second, to download the sources and compile the server on your own.

A binary for Windows platforms can be found on the [Squeak Wiki](#).

If you would like to compile the server yourself, download the sources from <http://www.garret.ru/~knizhnik/goods.html>, extract the sources and start compiling. The documentation can be found at the same page.

To setup and start the server, you have to create a configuration file named **#sttd.cfg** in the directory where the server binary resides. Edit the file to make it look like this:

```
1
0:localhost:6100
```

This tells GOODS to start a single instance of the server listening on port 6100 of localhost. Open a command prompt, go to the GOODS directory and start the server with the command:

```
goodsrv sttd
```

To terminate the server later on, just type **logout** in the execution window.

Installing the GOODS Framework

The next step is to install the GOODS Client Framework in Squeak. So just open the **SqueakMap Package Loader** (WorldMenu -> open...) and install the package **GOODS**.

Using GOODS

First we need a new database class, e.g. **StGOODSDatabase**, which has an instance variable **db**. We also need some methods for connecting, disconnecting and saving data. Let us create them:

```
connect

    self db: (KKDatabase
        onHost: self localhost
        port: self defaultPort).

StGOODSDatabase>>#connect

disconnect

    self db logout.

StGOODSDatabase>>#disconnect

initialize

    self connect.

StGOODSDatabase>>#initialize
```

We created and edited the **#initialize** method in the way shown above to allow us to be connected to the database every time we instantiate a new database object. Furthermore, we need to create a root object, from which all other objects to be made persistent in the database are referenced. Let us take a dictionary for that example:

```
createRoot

| users root |
users := OrderedCollection new.
root := Dictionary with: ('users' -> users).
self db
    root: root;
    commit.
```

```
StGOODSDatabase>>#createRoot
```

What you see above is that we create a new root object (which is a **Dictionary** containing all users using the key **users**) and commit the changes to the database. There are several other database-related methods like **#refresh** or **#rollback** which can also be used. They can be found in the class **KKDatabase**, but for now the above code should be sufficient.

Now we need a method to get all our users,

```
users

^ self db root
    at: 'users' ifAbsent: [self error: 'Database root not
initialized!']
```

```
StGOODSDatabase>>#users
```

and a method to add users, do not forget to commit the changes!

```
addUser: aUser

self users add: aUser.
self db commit.
^ aUser
```

```
StGOODSDatabase>>#addUser:
```

This works pretty well, but be aware of critical situations when two or more different database connections want to write their data. GOODS does not have a very good database management system. And all object structures are stored as they are, not in other database-internal structures. This means that, if, for example, you want to store a **BTree** and edit this tree at the same time with several connections, the database could crash. Please keep this in mind.

Saving All Data in an Object-Oriented Database: Magma

Another object-oriented database is [Magma](#). Other than GOODS, it provides you with a full-blown Smalltalk-only implementation, which is capable of persisting your objects either locally or on a remote Magma server. To install Magma, you should open the Package Universe and go to the *Persistence* category. Here you will find (among other packages) three versions of Magma: client, server and tester. They should be at least version 1.0. Client is the smallest and contains only the amount of Magma code required to connect to a remote server. The server package contains the client package and, additionally, code to establish your own server. The tester package contains the complete Magma distribution and includes a large test code base. As we will set up our own server here, the server package is recommended. However, be aware that Magma and Glorp have a naming collision with the class `Cache`. Although not verified, we believe that it is not possible use both of them seamlessly at the same time. Please see [Installing Magma](#) for the full Magma installation guide.

The next thing you have to do is to decide whether you want your database running in the same Squeak image your client will run in, or in another image. The former is a little bit easier but limits the scalability of your Web application: If you want to do things like load balancing you will need more than one image connecting to the server and thus one extra image running the Magma server. The only difference between these two solutions relevant in the scope of this tutorial is the way connections to Magma are established. Switching between the

two alternatives is easy.

Now its time to set up our database. That is, we first create a Magma repository on our hard disk (the following example path is Windows specific). In your workspace, simply execute:

```
MagmaRepositoryController
  create: 'C:\MagmaDBs\SeasideTutorial'
  root: Dictionary new.
```

This will take some seconds to create the required file structure in the specified directory. If you chose to run Magma in a second image, you now have to start the Magma server in that image on some port and inspect it for later operations:

```
MagmaServerConsole new
  open: 'C:\MagmaDBs\SeasideTutorial';
  processOn: 51001;
  inspect.
```

To shut down the server at any time, just send it a **#shutdown** message, for example via the inspector.

Now we are able to build a class similiar to those we had with GOODS above. Name it **StMagmaDatabase**, give it an instance variable **session** and create the accessors for it. Afterwards we can write the **#connect** method:

```
connect

self session: (MagmaSession
  hostAddress: self localhost
  port: self defaultPort).
self session connectAs: 'tutorial'.

StMagmaDatabase>>#connect

localhost

^ #(127 0 0 1 ) asByteArray

StMagmaDatabase>>#localhost

defaultPort

^ 51001

StMagmaDatabase>>#defaultPort
```

As you can see, we simply connect to **localhost:51001** under the name **tutorial**. Of course, the host address should be the computer on which your Magma database runs. If you want to use Magma locally only, you can use the following code:

```
connect

self session: (MagmaSession
  hostAddress: self localhost
  port: self defaultPort).
self session connectAs: 'tutorial'.

StMagmaDatabase>>#connect
```

For the **#disconnect**, there are two different implementations regarding whether you work locally or remotely --

with another image running Magma. As said above, we are using remote access here:

```
disconnect

self session disconnect.
self session: nil.

StMagmaDatabase>>#disconnect
```

For the sake of completeness, here is the implementation of **#disconnect** for the local variant, which additionally has to close the repository used by the local session:

```
disconnect

self session
  disconnect;
  closeRepository.
self session: nil.

StMagmaDatabase>>#disconnect for local connections
```

From now on, usage is very similar to GOODS. As you might have seen, we created the repository with a Dictionary as the root. Via this root, we can now navigate to every object in the database via references. Magma can automatically detect changes to this structure and save them, if we surround those changes with a **#commit**. Thus, the following methods should be very clear for you:

```
createUsers

self session
  commit: [self session root
    at: 'users' put: OrderedCollection new].

StMagmaDatabase>>#createUsers

users

^ self session root
  at: 'users' ifAbsent: [self error: 'Database root not
  initialized!']

StMagmaDatabase>>#users

addUser: aUser

self session commit: [self users add: aUser].
^ aUser

StMagmaDatabase>>#addUser:

addTask: aTask toUser: aUser

self session commit: [aUser tasks add: aTask].
^ aTask

StMagmaDatabase>>#addTask:toUser:
```

Keith Hodges has written another [Magma Tutorial](#). There you can learn more details about the [Magma seasideHelper](#) and how to integrate him in our ToDo Application.

One Database Connection per Session

It is advisable to have only one database connection per user session. Thus, every time a new user connects to your Web application, a connection to the database is established. If the user's session expires or the user logs out you can disconnect from the database.

To achieve this, create an instance variable in your session class called **database** with corresponding accessors. For illustration purposes, we will use the Image Database approach. Add the following:

```
initialize

super initialize.
self database: StImageDatabase new.

StSession>>#initialize
```

We want to disconnect from the database whenever the session is terminated. In such a case, Seaside calls the session's **#unregistered** method.

```
unregistered

self database disconnect.
super unregistered.

StSession>>#unregistered
```

Finally we have to connect the database with our application. Up to now, users get a new list of todos whenever they start a new session. The first step is to display the user's list instead of the default **testTasks** list. We get them from the current session's user object. Of course, we can also remove the **#testTask** method right now.

```
initializeListComponent

self listComponent: StListComponent new.
self listComponent
  sortBlock: [:items |
    items sortBy: [:a :b | a deadline < b deadline]];
  renderItemBlock: [:task :html |
    self renderTask: task asRowOn: html];
  items: self session user tasks.
self showPendingTasks.

StLoggedInComponent>>#initializeListComponent
```

If you are working with the **StImageDatabase**, all might work fine for you. But in other databases, you have to save the modified object explicitly. So we must modify the **StLoggedInComponent**'s **#createTask** and **#editTask** methods:

```
createNewTask

(self call: self taskEditor newTask) ifFalse: [^ self].
self session database
  addTask: self taskEditor task
  toUser: self session user.

STLoggedInComponent>>#createNewTask

editTask: aTask

self taskEditor task: aTask copy.
(self call: self taskEditor) ifFalse: [^ self].
aTask copyFrom: self taskEditor task.
```

```
self session database save: aTask.
```

```
STLoggedInComponent>>#editTask:
```

The last step before having a working ToDo Application is to implement the **StSession>>#findUserByEmail:** method. It dispatches the request to the used database:

```
findUserByEmail: anEmail
```

```
^ self database findUserByEmail: anEmail
```

```
StSession>>#findUserByEmail:
```

The database solutions work pretty well but you may ask what happens if the server is down, and how to display a specified error message.

Achieving this is pretty easy: all you have to do is add the following to your session class:

```
responseForRequest: aRequest
```

```
self database connectionFailed
```

```
  ifTrue: [^ WResponse new nextPutAll: 'No Database Connection'].
```

```
^ super responseForRequest: aRequest
```

```
StSession>>#responseForRequest:
```

Summary

Persistence is a huge topic in Seaside and Squeak, different alternatives have been illustrated on how to store data of a Web application. In addition to these, there are many other interesting approaches, especially [GemStone](#). A complete list specially for Squeak can be found on the [Squeak Wiki](#). Ajax and the comfortable integration in Seaside will be the topic of the following chapter.

[Back](#)

9 - Ajax

What you are going to learn

- [Future of the Web and Ajax](#)
- [script.aculo.us](#)
- [Installation](#)
- [Updater](#)
- [InPlaceEditor](#)
- [Lightbox](#)
- [Summary](#)

Future of the Web and Ajax

The intent of the future Web is to create Web pages that feel like common desktop applications. By the use of Ajax, some parts can be renewed and replaced individually, instead of reloading the whole page. The page seems to be more responsive, its interactivity and usability increases.

The name Ajax stands for Asynchronous JavaScript And XML. JavaScript is the client side programming language, in which the function calls are made. Using the XMLHttpRequest object allows to perform asynchronous HTTP requests in the background, that means the normal HTML and JavaScript processing is not affected. In such a call, typically an XML encapsulated HTML snippet is transferred and, upon arrival, exchanged within the page.

script.aculo.us

The most popular Ajax framework is [script.aculo.us](#). It provides many visual effects like fading elements in and out or dragging elements. For example, one could create an online shop where users have to drag products and to drop them into a shopping cart. Information about dragged and dropped elements can be passed back to the application so that it is able to react to these events. The framework also provides InPlaceEditors or HTML updates that can be used for dynamic auto completing input forms, for example. These components will be discussed later on.

Script.aculo.us was written by Thomas Fuchs and first released in June 2005. It is based on the Prototype framework, written in JavaScript. As all data are exchanged by XML messages, no special programming language is required on the server side. So there are many bindings for different languages like PHP or Ruby.

Lukas Renggli has integrated the framework in Seaside. As usual in Smalltalk, all code is encapsulated very well. For most use cases, it is not even necessary to know a single line of JavaScript code. The link <http://localhost:8080/javascript/scriptaculous> refers to the integrated script.aculo.us (Seaside) demo application and shows in a specific manner the functions and capabilities of the library. You can find the source code of all demo applications within the Web site and immediately adopt the desired effects in your own applications. In every Seaside image, you can find this demo application when the script.aculo.us library works correctly.

In the next three subchapters, we show, by means of a few simple examples, how we extended our ToDo Application with Ajax features and thereby improved usability. Before that, we will say a few words about the required configuration to use the capabilities of script.aculo.us.

Installation

In our proposed web-dev image, the script.aculo.us integration is already installed. In many other Seaside installations, you can find the framework directly. The simplest check is if you can find a class like **SULibrary**. This main library contains all the JavaScript code for the framework and is indispensable. Another way is to

open the internal script.aculo.us Web site under the path (**/seaside/tests/scriptaculous**) which works fine provided the installation is correct.

If all tests are negative you must install script.aculo.us separately. Seaside 2.7 – better: 2.8 – should be installed already. To achieve this, you should open the Monticello Browser in the usual way if you want to update the framework as described in chapter "First Steps". Next, load the most recent Scriptaculous packages into your image.

To use the new functions with your own application it is necessary to bind the **SULibrary**. To do this, you should start the Seaside config Web page in your Web browser. Find the entry point of your application and follow the link *configure*. Hereafter, you see, at the top of the config Web page, all of your bound libraries. With the drop-down list you can select and add the **SULibrary**. If everything works, you can now see the name in the list of the loaded libraries, and, at this point, you can use all functions of the script.aculo.us framework in Seaside and your application.

Updater

As the name Asynchronous JavaScript And XML implies, the strength of this concept lies in the asynchronous background communication with the Web server. The most significant advantage is to reduce data traffic, because the server needs to update only those parts of the Web page whose content has actually changed. The user gets a major usability improvement and the feeling of a flexible (desktop) application since the blocking loading of a Web site is hidden. The most commonly used script will be **SUUpdater**, which enables us to render just parts of a Web page without loading the entire content. These parts are identified with HTML ID tags mostly used in DIVs. In this example, we insert an updater in every checkbox of the task list. Here, we want to use the checkbox without a form and a send button, which means that we need a script that sends the changes to our Web server and Seaside in the background and then renders the list again. The method **#renderTask:asRowOn:** describes the look of a task row and it calls the following method to render a checkbox at the first position of a row (see below).

```
renderCheckboxWith: aTask on: html

| updater |
updater := html updater
    id: 'list';
    callback: [:renderer |
        aTask completed: aTask completed not.
        renderer render: self listComponent];
    yourself.
html checkbox
    onClick: updater;
    value: aTask completed.

StLoggedInComponent>>#renderCheckboxWith:on:
```

The message **#checkbox** creates a normal checkbox element, whose state is set with **#value**. The interesting part is the **onClick** handler to which the **SUUpdater** script will be attached. It is triggered every time somebody clicks on the checkbox. Its construction looks like a usual HTML element in Seaside. The properties of a script are facultative or necessary. The methods **#id:** and **#callback:** are both required: the first one sets the target which should be updated (the list component is embedded in a DIV tag with the `id='list'`), and the second one registers a callback block which is called if the script goes active. It is not useful to leave out one of the properties - indeed, the script will be rendered but it will have no real effect. The callback has one parameter in its block, the current **renderer** (here called **:renderer** to differentiate it from **html**) to update the changed parts. It renders all content in the target division. The **html** renderer is, at the time the script is triggered, not available any more, so we need a new one and get it with the parameter **:renderer**. Both are objects of the same class so you can use them similarly. In this case we render the list component new because only here we have potentially changed data (for example a task switched from pending to completed state).

InPlaceEditor

InPlaceEditors (Figure 9.1) give the user the ability to edit text without an explicit static text field. The user

moves the mouse over the text and with a further click can edit the highlighted region with new text input. This function is shown as an example in the task list. With this, you can edit the name directly without the form in the task editor. The following source code shows the simple use of the script.



Figure 9.1: InPlaceEditor

```
renderInPlaceEditorWith: aTask on: html
```

```
| editor |
editor := html inplaceEditor
    cancelControl: false;
    triggerInPlaceEditor: [:value | aTask taskName: value];
    callback: [:renderer | renderer render: aTask taskName];
    yourself.
html div
    script: editor;
    with: aTask taskName.
```

StLoggedInComponent>>#renderInPlaceEditorWith:on:

Like the updater script (see the previous subchapter), this method is called from **#renderTask:asRowOn:** (see below), in which the appearance of a task is defined. But instead of rendering `tableData: aTask name`, we now call the message above. You can easily see the separation between the DIV as view (**#with:** message) and the InPlaceEditor at the **#script:** message. The script consists of three properties. First, we do not want a cancel button. Second, when the modification is submitted, the triggered block will be activated and the data model in our application will change. Third, the HTML element which has the specified id will be updated. Here, we do not need an id or target because the renderer makes an update in the same DIV. The first message is optional; without it, the result would be an InPlaceEditor with a predefined cancel button.

```
renderTask: aTask asRowOn: html
```

```
html tableData: [html
    tableData: [self renderCheckboxWith: aTask on: html];
    tableData: aTask deadline asString;
    tableData: [self renderInPlaceEditorWith: aTask on: html];
    tableData: aTask taskDescription;
    tableData: [html anchor
```

```
callback: [self editTask: aTask];
with: 'edit']].
```

```
StLoggedInComponent>>#renderTask:asRowOn:
```

Lightbox

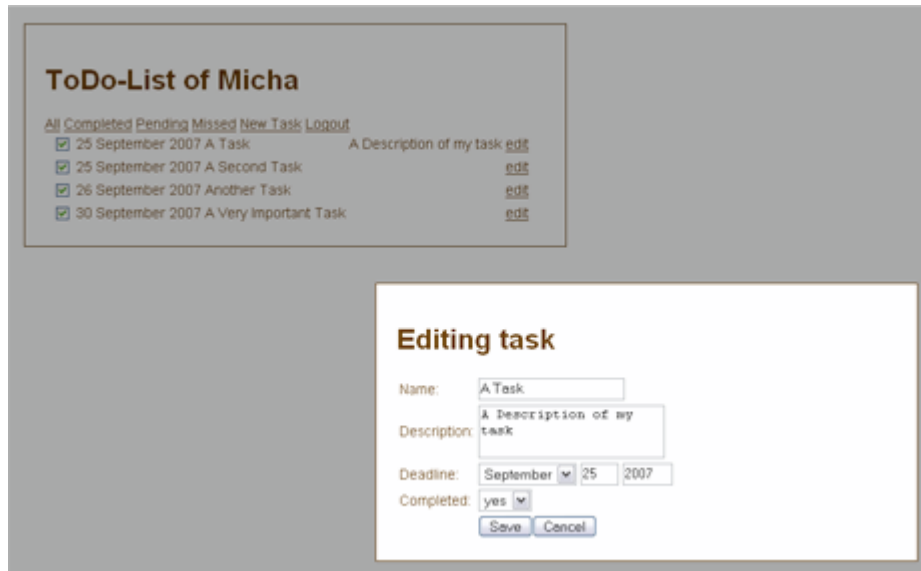


Figure 9.2: Lightbox with Task Editor

It is not easy to get the feeling of a real desktop application in a Web browser. A little trick with a lot of power is to use a lightbox. Here, the renderer paints a separate window as an overlay on the complete Web site and dims the background (Figure 9.2), you can see your dimmed Web site in the background and a form mask in the front of your page. In our ToDo Application we use this feature to render the task editor in the foreground. At this point, we do not need a new script but we simply change the **#call:** message into **#lightbox:**. Seaside and script.aculo.us do the remaining work and render the component in the overlay. The result is amazing and needs just a bit of CSS post processing (we have already added the styles via the **StToDoLibrary**). The modified **#createNewTask** method shows the call of a lightbox; the **#editTask:** is changed in a similar way.

```
createNewTask
```

```
(self lightbox: self taskEditor newTask) ifFalse: [^ self].
self session database
  addTask: self taskEditor task
  toUser: self session user.
```

```
StLoggedInComponent>>#createNewTask
```

Summary

Seaside and script.aculo.us have many other scripts and effects, but it goes beyond the scope of this tutorial to describe each single one of them. We have discussed three short examples which we think provide a good basis for making your own experiences with the technology. The next chapter tells of the meta-description framework Magritte with which you can in a simple way generate parts of your Seaside application automatically.

[Back](#)

10 - Magritte

What you are going to learn

- [What is Magritte?](#)
- [Describe StTask](#)
- [Create a Magritte Task](#)
- [Dynamic Descriptions](#)
- [Create a Report](#)
- [Summary](#)

What is Magritte?

In this chapter, we will introduce Magritte, a meta-description framework. You should have read and understand the other chapters before you continue reading.

As you remember, we have created a special dialog (**StTaskEditor**) for creating and editing tasks. What if we do not only have tasks but also many other different objects that should be editable via such a dialog? And what if we have multiple lists - do we always need to create a list view like **StListComponent**? The answer is: no, we use Magritte!

Magritte provides a set of so-called descriptions. You use them to declaratively describe all attributes of a domain object. Such a description defines the type of each attribute, for example, a **Date**, **Boolean**, or any other object. By default, all descriptions are collected into a description container. In addition, you can also use dynamic descriptions, that means you define new description containers with a subset of the object's descriptions.

These containers are used when creating a component for an object. This is done by sending **#asComponent** to that object. For every description stored in the container, an input field will be displayed and filled with the current value of the described attribute. You see: not only can Magritte create a task editor for us, it is even more powerful than ours!

Magritte was invented by [Lukas Renggli in his master's thesis](#). As a proof of concept, he implemented the content management and wiki system Pier. Both are still improved and used by many Smalltalkers around the world.

So let us start rewriting our ToDo Application.

Describe StTask

In our ToDo Application, we have two domain objects: the user and the task. We want to use Magritte for the creation of input dialogs for the tasks. Hence, we have to add descriptions for every instance variable that should be editable. They are as follows:

completed
for Boolean values
deadline
for a Date
taskDescription
for multi-lined text
taskName
for a single-lined text

Let us start with the description for **completed**:

```
descriptionCompleted
```

```
^ MABooleanDescription new
  selectorAccessor: #completed;
  label: 'Completed?';
  priority: 30;
  default: false;
  yourself
```

```
StTask class>>#descriptionCompleted
```

What can we learn from this code? The method name starts with `description`. As said above, Magritte queries all these methods. The second part of the method name is, by convention, the name of the described instance variable. The method returns an instance of **MABooleanDescription**. For many data types there are predefined description classes as we will see in the following.

Every description has the methods

- **#selectorAccessor:**,
- **#label:**,
- **#priority:**,
- and **#default:**

(there are many others; see **MADescription**). **#SelectorAccessor:** gets the name of the accessor method as a symbol. In our case, the **StTask** has the getter **#completed:** and the setter **#completed**. **#Label** defines a string that is shown in front of the input element to identify the field. The value of **#priority:** determines the order of the elements in the generated view. The first element is the one with the lowest value, the last the one with the highest. The **#default** message sets the default value for that element.

Now, we can create the descriptions for **deadline** and **taskDescription**:

```
descriptionDeadline
```

```
^ MAMemoDescription new
  selectorAccessor: #deadline;
  label: 'Deadline';
  priority: 20;
  default: Date today;
  yourself
```

```
StTask class>>#descriptionDeadline
```

```
descriptionTaskDescription
```

```
^ MAMemoDescription new
  selectorAccessor: #taskDescription;
  label: 'Description';
  priority: 40;
  default: '';
  yourself
```

```
StTask class>>#descriptionTaskDescription
```

We see that the description class for date input is **MAMemoDescription**, the one for multilined texts **MAMemoDescription**. Now, we will create the description for the task name including the conditions above. Conditions can be added using the **#addCondition: labelled:** method which accepts a block as argument. The block can have one argument that represents the current value of the instance variable. If the block evaluates to false, then the given message is displayed.

```
descriptionTaskName
```

```

^ MAStringDescription new
  selectorAccessor: #taskName;
  label: 'Task Name';
  priority: 10;
  default: 'New Task';
  addCondition: [:value | (value ~= 'New Task')
    and: [value size <= 50]]
  labelled: 'Task name is invalid or too long';
  yourself

```

```
StTask class>>#descriptionTaskName
```

Create a Magritte Task

After having created the Magritte descriptions for a **StTask**, we want to see it in action in the next step. New tasks are created using the **StTaskEditor** which was called in message **StLoggedInComponent>>#createNewTask**. So let us rewrite its source code:

```

createNewTask

| task |
task := self call: StTask new asComponent addValidatedForm.
task ifNotNil: [self session database
  addTask: task toUser: self session user].

```

```
StLoggedInComponent>>#createNewTask
```

Now, log in to the ToDo Application and create a new task. The form will look as shown in figure 10.1.

The image shows a web form titled 'Task Editor'. It has four main input areas: 'Task Name' with the text 'New Task', 'Deadline' with the date '12 March 2008' and a 'Choose' button to its right, 'Completed?' with an unchecked checkbox, and 'Description' with a large text area. At the bottom of the form are two buttons: 'Save' and 'Cancel'.

Figure 10.1: Task Editor generated by Magritte

How does this work? Let us have a look on the code again. We have created a new **StTask** and called **#asComponent**. This method is provided by Magritte and implemented in **Object**. It calls the **#description*** methods and creates a component with input fields for all descriptions. By default, the created dialog does not have any submit buttons. Only when sending the **#addForm** or **#addValidatedForm** message, buttons are added. When the cancel button is pressed, the component answers nil. If the save button was pressed and all conditions from the descriptions evaluate to true, the created (or edited) object is returned. In case of unfulfilled conditions, the form is displayed again and the error message is shown. The latter is only done if the component received the **#addValidatedForm** message.

If the component returned a task, then we store it in the database.

Dynamic Descriptions

In the last step, we used Magritte for creating input forms for new tasks. In this step, we want to use it for editing existing ones. It is quite similar. But when editing a task, we also want to display the id of the task without it being editable. Therefore, we create a dynamic description. On the instance side of **StTask**, we add **#descriptionEdit**. The name of the description starts with **description** and is followed by some identifying words

by convention. It is not necessary like at class side, but it is a good idea to stick to the naming schema.

```
descriptionEdit

^ self description copy
  add: (MANumberDescription new
    selectorAccessor: #id;
    label: 'Id';
    priority: 5;
    beReadOnly;
    yourself);
  yourself

StTask>>#descriptionEdit
```

We copy the default description container and add a new description for a number as the id is a numerical value if set. By sending the **#beReadOnly** message, the created input element will not be editable by the user.

The container must be copied as it is cached by Magritte. Unless we copy it, we would always add another **MANumberDescription** every time the method is called.

You cannot only add descriptions to existing containers, you may also want to omit descriptions. Therefore, the descriptions container implements the whole **Collection** protocol, that means methods like **#select:** or **#collect:**.

Back to our ToDo Application. Now, we have a dynamic description that should be used for an edit dialog. So, **StLoggedInComponent>>#editTask:** has to be modified.

```
editTask: aTask

| task |
task := self call: (aTask descriptionEdit asComponentOn: aTask)
addValidatedForm.
task ifNotNil: [self session database save: aTask].

StLoggedInComponent>>#editTask:
```

The method looks quite similar to **#createTask**. They differ a bit in saving the modified task and a bit more in the way how Magritte is involved. For an empty object, for object creation so to speak, we called **StTask new addValidatedForm**. But for existing objects, we request the description container which should be shown (**aTask descriptionEdit** in our case) and call the message **#asComponentOn:** followed by the object to edit. This will create the component to be shown.

Replace one **aTask** by **StTask new** in the first line and see what is happening. If you replace the first one, nothing changes. That is obvious because **#descriptionEdit** returns the same description container for both an empty and an existing task. But as the second **aTask** is the object to modify, you will see an empty dialog when replacing it by **StTask new**. If you call **#description** instead of **#descriptionEdit**, you will get the dialog you already know from creating tasks. But it is also filled with the values of the edited task.

Do you remember how much code was required to create the **StTaskEditor** component? It does the same things we just did with Magritte. Imagine you do not only have one or two views on an object but a dozen. Using Magritte can save you many lines of code.

Create a Report

Magritte is not only able to create input forms for objects. It can also display reports, a table of a collection of objects. In this section, we want to create such a report and use it instead of **StListComponent**. The list should also simply display the name and the deadline of the task and whether it is completed. A report is created by calling **MAReport class>>#rows:description:**. The first argument is the collection of items to display and the second a description container that describes each item. We just want to show the name and the deadline of the task, so we have to create a second dynamic description:

```
descriptionReport

  ^ super description
    select: [:each | #(taskName deadline) includes: each accessor
selector]

StTask>>#descriptionReport
```

Now we can create the report. As it is a component, too, we need an instance variable for it to be stored in which must also be returned by **StLoggedInComponent>>#children**. So, create an instance variable called **report** with corresponding accessors, and modify **#children** to look like this:

```
children

  ^ Array with: self menuComponent with: self report

StLoggedInComponent>>#children
```

The report is built once and updated every time items are added, modified or the selection is changed. Let us create a method **StLoggedInComponent>>#initializeReport** that builds the initial report.

```
initializeReport

self report: (MAREport
  rows: self session user tasks
  description: StTask new descriptionReport).
self report
  addColumn: (MAToggleColumn new
    selector: #completed;
    title: 'Completed';
    setDescription: StTask descriptionCompleted;
    sorter: nil;
    yourself);
  addColumn: (MACommandColumn new
    addCommandOn: self selector: #editTask: text: 'edit';
    yourself).

StLoggedInComponent>>#buildReport
```

As described above, the report is created by calling **MAREport>>#row:description:**. Initially, we add all our tasks to the report. The description is the newly created dynamic one, queried from an empty task.

In the second part of the message, we add two columns to the report. The first one contains a link to toggle the completion state of the task, the second holds a link to edit the task. Let us have a closer look at both. The **MAToggleColumn** needs the selector of the instance variable to toggle. The column needs a description to know what type of value should be displayed, so we use the **StTask class>>#descriptionCompleted** message we have already defined. As we do not want this column to be sortable (every column of a report is sortable by default), we set the sorter to nil. The edit link is even easier to create. We add a **MACommandColumn** to the report and call **addCommandOn: anObject selector: aSymbol text: aString**. This creates a link with the caption **aString**. When clicking on the link, **aSymbol** is called on **anObject** and the item displayed in the current row is supplied as argument. In our case, we call the **#editTask:** message of **self** (**StLoggedInComponent**). Pay attention on the colon at the end of **#editTask:**, it is part of the selector's name.

Since we want to filter tasks out of the list (for example the pending ones when displaying the completed), we need another instance variable with accessors, namely **filterBlock**, that keeps a block. It will be evaluated for every task and only those tasks are displayed, that evaluate to **true**.

The **#initializeReport** message must be called when initializing the component. So we change **StLoggedInComponent>>#initialize** to look like this:

```
initialize

super initialize.
self
    initializeMenuComponent;
    initializeReport;
    filterBlock: [:item | true].

StLoggedInComponent>>#initialize
```

The next step is to render the report instead of the list component. To achieve this, we have to modify **StLoggedInComponent>>#renderContentOn:**. Instead of calling `self listComponent` in the last line we call `self report`. That is it. Start a new session of your ToDo Application and have a look at the report Magritte rendered for us.

You may have seen very quickly that we still have a little work to do. The links to limit the tasks do not work yet and the report will not be updated when adding or editing some tasks. So let us fix this right now.

Similar to the list component, we use a filter block to filter all unneeded items. Create an instance variable **filterBlock** and corresponding accessor methods. The refresh of the report is done in a method **#refreshReport**.

```
refreshReport

self report rows: (self session user tasks select: self
filterBlock).

StLoggedInComponent>>#refreshReport
```

As you see, we set the rows of the report to all the user's tasks that match the filter block. Now we change all the show methods:

```
showAllTasks

self
    filterBlock: [:item | true];
    refreshReport.

StLoggedInComponent>>#showAllTasks

showCompletedTasks

self
    filterBlock: [:item | item completed];
    refreshReport.

StLoggedInComponent>>#showCompletedTasks

showMissedTasks

self
    filterBlock: [:item | item isMissed];
    refreshReport.

StLoggedInComponent>>#showMissedTasks

showPendingTasks

self
    filterBlock: [:item | item isPending];
    refreshReport.
```



```

StLoggedInComponent>>#showPendingTasks

createNewTask

| task |
task := self call: StTask new asComponent addValidatedForm.
task ifNotNil: [self session database
    addTask: task toUser: self session user.
    self refreshReport].

StLoggedInComponent>>#createNewTask

editTask: aTask

| task |
task := self call: (aTask descriptionEdit asComponentOn: aTask)
addValidatedForm.
task ifNotNil: [self session database save: aTask.
    self refreshReport].

StLoggedInComponent>>#editTask:

```

| ToDo-List of David | | | |
|---------------------------|---------------------------|---------------------------|------------------------|
| All | Completed | Pending | Missed |
| New Task | Logout | | |
| Task Name | Deadline | Completed | |
| Already completed task | 12 March 2008 | true | edit |
| Missed task | 12 March 2008 | false | edit |
| Pending task | 10 March 2008 | false | edit |

Figure 10.2: The report created by Magritte

If a task was added or edited, the report must also be updated by calling **#refreshReport**.

That is all we need to do. We still have the same application (see figure 10.2, it looks like the list component), but used less code to implement it. All dialogs and reports we have had to create by ourselves are now generated by Magritte. All we had to do was to describe the attributes of our domain objects, **StTask** in this example.

Summary

In this chapter, you have learned how to use Magritte to create powerful input dialogs and reports. The only thing you had to do was to describe your domain objects properly and then to invoke Magritte. Magritte generates the dialogs and validates your given conditions. With Magritte, you can develop faster and save a huge amount of code.

Magritte also prevents your objects to become invalid during editing by the use of the Memento pattern: the input dialog is working on a copy of the original object. Only when you successfully submit the form and no condition fails its data are copied back. If another person changed the same object concurrently, Magritte tries to merge your changes automatically.

You cannot only describe basic data types like numbers or strings. Complex structures, that are classes or even collections of classes, can also be described using relation descriptions. Of course, you can extend the description system by your own descriptions, too.

Magritte is not limited to Web applications. There is a Morphic binding, too, that works quite similarly. Instead

of sending **#asComponent**, you call **#asMorph** and a new morph is shown. If you have described your domain objects once, you can use these descriptions in both environments.

The layout of the generated forms is not finite. You can use CSS to style it, but in some cases you may want to have a very special layout. Therefore, you can create your own Magritte renderer or component. Within domain objects, you can then define that your renderer or component should be used instead of the default one.

This chapter was only a very small introduction to the meta-description framework Magritte. We hope that you have seen that it allows you to fast and easily create different views to your domain objects, faithfully obeying Magritte's motto: **Describe once, get everywhere**.

[Back](#)

11 - The Fine Print

What you are going to learn

- [Introduction](#)
- [The URL with _s and _k](#)
- [Request/Response Processing](#)
- [Session Handling](#)
- [Rendering Tree](#)
- [Continuations](#)
- [Canvas](#)
- [Summary](#)

Introduction

This chapter will introduce a few important technical details of Seaside. So you can learn more about the concepts and it will be an entry point to the backstage area of this sophisticated Web framework, as well as an answer to the most frequently asked questions about the internals, like 'what do _s and _k mean?', 'how will my be Web site rendered?' or 'why does the back button work?' Seaside is permanently under development and things will be changed, thus some of these details will be different in the near future. This should still be enough for you to understand the main concepts. The following chapter is based on the Seaside Version 2.8 (Monticello Package: **Seaside2.8a1-lr.522**).

The URL with _s and _k

If you look in the address bar of you browser after running a Seaside application you will see in most cases a URL like **http://localhost:8080/anEntryPoint?_s=runEAdkJInTOQeop&_k=dcmGcWIy**. This seems very cryptic and not really RESTful ([Representational State Transfer](#)). So what do these parameters mean? First of all: **http** is the Hypertext Transfer Protocol and it describes how to transfer information through the network (especially the Web). This is not in the scope of Seaside and we do not discuss it here (For more information: http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol). The second part **localhost:8080** describes your Web server and the port used. You can find them in your Squeak image. Have a look at the adapter between Seaside and the Comanche Web server **WAKom** and the class **HttpService** (see chapter "First Steps - Installation"). **Seaside** is the path identifier for the **WADispatcher** to decide whether to handle the incoming request or not. The leftover part of the URL is the exciting one and this part is generated by Seaside automatically. So let us have a closer look at it:

anEntryPoint

identifies a **starting point** for Seaside to handle the incoming request that can be a **WADispatcher** like the first for the path **seaside**, a **WFileHandler** to allow the access to files or in most cases a **WAApplication**. When you register your own component as a root for your entry point (see chapter "First Steps - How to Configure Your Application"), a new **WAApplication** object is added to a **WADispatcher**. These will forward all requests to the application object. The super class of **WAApplication** is a **WRegistry** storing all sessions, so the application can look for the active session to handle the request. In addition, the application object has two instance variables. The **configuration** contains a list of **WAConfigurations** that define attributes like the **rootComponent** or custom configurations and the variable **libraries** containing a collection of **WALibrary** which are used to serve CSS, JavaScript and images.

_s=runEAdkJInTOQeop

describes the **handler field** of the application. This is a unique key used to identify the active session for the user. The value is constant during the whole application for each user. In the **WRegistry** (see above) all **WASession** objects are stored for one application. The dictionaries **handlersByKey** and **keysByHandler** save this information. This concept allows a unique session for each user which handles the request and stores the current state. By the way, the process also checks whether the session is expired. Should the value for a handler field be missing, the application will create a new one.

_k=dcmGcWIy

is a unique key named **action field**. All handlers and **ResponseContinuations** are identified with _k. These handlers (up to 20 objects) are stored in the session 'least recent used cache' (**WALRUCache**). The current session forwards an incoming request to one of the stored handlers and the selected one process callbacks, renders the Web site or makes an internal redirect. A request without a _k will create a new render handler. Further information to this topic can be read in the next section.

When you move over a link with your mouse you can see another URL in the status bar with an additional parameter. These numbers describe the key for the callback dictionary in Seaside. Every link has a unique key to identify the corresponding callback block in the **WARenderingContext**. Also you can see that the action field is different from the address bar. The reason for that will be explained in the next section, as well.

Request/Response Processing

Once you start the Web server in your squeak image with the command **WAKom startOn: 8080**, an instance of **HttpService** (or **ComancheNetService**) will be created. This object has as plugged-in the current **WAKom** instance and this again is an entry point to the default dispatcher. Now the Web server accepts requests and forwards them to the **WAKom** plug-in.

```

process: komRequest
  | request response komResponse |
  request := self convertRequest: komRequest.
  response := self handleRequest: request.
  komResponse := self convertResponse: response.
  response release.
  ^ komResponse

```

WAKom>>#process:

The method above gets a native request and consists of four parts: First, the native request is converted to a Seaside request (**WARequest**); second, this request is transferred to the dispatcher which returns a **WAResponse**; third, the response is converted back to native HTTP; and last but not least, the final response is returned to the Web server and the user (browser). We will now follow the handle request path to the dispatcher. There the **starting point** of the application is identified by the URL. With this information the dispatcher can choose a registered entry point from his dictionary or use the default (in general **browse**) if he cannot find the required one. The selected entry point receives the message **#handleRequest: aRequest** again and he can decide what to do. A dispatcher takes the same path again, a file handler will look for the required file and an application (more precisely the **WARegistry**) searches for the `_s` value in the URL.

```

handleRequest: aRequest
  ^ (aRequest fields includesKey: self handlerField)
    ifTrue: [ self handleKeyRequest: aRequest ]
    ifFalse: [ self handleDefaultRequest: aRequest ]

```

WARegistry>>#handleRequest:

Here we distinguish two paths depending on the `_s` value. In the first case a key exists and the following method is called.

```

handleKeyRequest: aRequest
  | key handler keyString |
  "Under some circumstances, HTTP fields are collections of values"
  key := [keyString := aRequest at: self handlerField.
    (keyString isKindOfClass: OrderedCollection)
    ifTrue: [keyString := keyString first].
    WAExternalID fromString: keyString] on: Error do: [:e | nil].
  handler := handlersByKey at: key ifAbsent: [nil].
  ^ (handler notNil and: [handler isActive])
    ifTrue: [handler handleRequest: aRequest]
    ifFalse: [self handleExpiredRequest: aRequest]

```

WARegistry>>#handleKeyRequest:

The method reads the `_s` value and converts it to a **WAExternalID**. Afterwards the suitable handler (a **WASession**) will be chosen from the dictionary **handlersByKey**. A expired session or a nil value will trigger a redirect to the starting point of the Web application, else the handler performs the request.

The other path (see above **WARegistry>>#handleRequest:**) cannot find a `_s` value and so the application executes **#handleDefaultRequest:**.

```

handleDefaultRequest: aRequest
  ^ (self sessionClass new setParent: self) handleRequest: aRequest

```

WAApplication>>#handleDefaultRequest:

This method creates a new session with the application as parent and sends **#handleRequest:** again. Please note: At this point the session does not get its own `_s` value and will not be registered in the **WARegistry**. This happens a few steps further when the renderer creates a new rendering context and binds itself to the action field (see below). For that reason the automatically generated URL is valid and the current session can be found.

The following UML class diagram (Figure 11.1) symbolizes the previous description. The diagram is not complete and it is just showing the most important classes in this context. The starting point for an application is **WAApplication**, which looks for the suitable session that handles the incoming request. Furthermore you can see the abstract super class **WARequestHandler** for entry points and sessions which adds the possibility to handle requests and lets a dispatcher become a starting point, too.

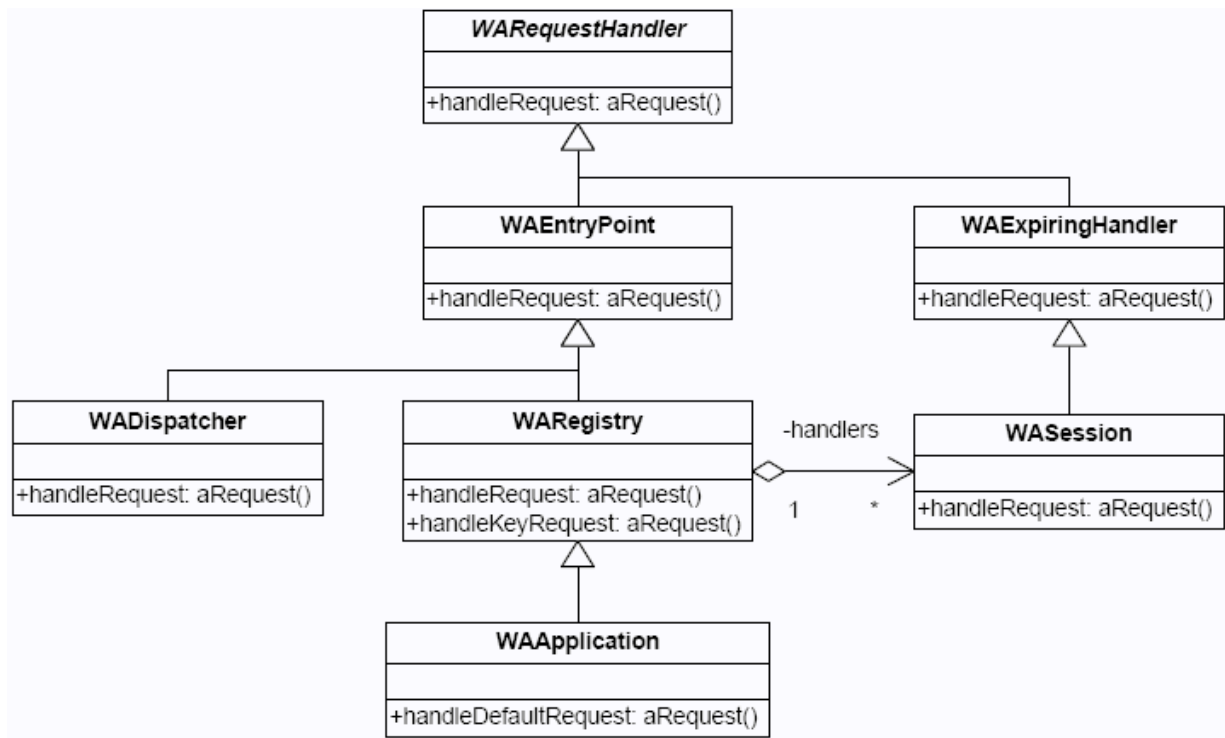


Figure 11.1: Request Handler

Up to this point the request/response processing is easy in contrast to what will come. Now the session has a request and the task to process. At first, the session sets the last access on 'now', stores the current request and starts the execution in a new process. After that the execution path comes to the key method **#performRequest: aRequest**. Please take a hint and ignore the method **#withEscapeContinuation: aBlock** at this point. We will consider the details in "Continuations" and you should only know that with the help of this method, the executing process can abort at any point in the calculation and return a response to the Web server by sending **#self session returnRespond: aResponse**. The concept of **continuations** is sometimes difficult to understand and we will discuss it later.

```

performRequest: aRequest
| key continuation |
key := aRequest fields
at: self actionField
ifAbsent: [ "self start: aRequest ].
((aRequest fields includesKey: self checkCookiesField)
and: [ aRequest cookies isEmpty ])
ifTrue: [ cookiesEnabled := false ].
(key isKindOfClass: OrderedCollection)
ifTrue: [ key := key first ].
continuation := continuations
at: key
ifAbsent: [ "self unknownRequest: aRequest ].
^ continuation value: aRequest

```

WASession>>#performRequest:

Let us have a look at the method **#performRequest:**. First, do not be afraid when seeing the variables **continuation** or **continuations**. These are a heritage of previous Seaside versions where real continuations have played a very important role but in Seaside 2.8 the implementation has been refactored. The temporary variable **continuation** describes in most cases a handler which handles an incoming request to render the Web site or perform callbacks. The other variable is an instance variable of **WASession** of the type **WALRUCache**. There the last, about 20 handlers are stored in a 'least recent used cache' and ready to handle a request again. Figure 11.2 shows the internal structure. Each session has a cache storing redirect handlers (**WARedirectContinuation**), render handler (**WARenderContinuation**) or response continuation (**ResponseContinuation**). The last one is a real continuation and will be described in section "Continuations". The others are handlers with the name of continuations thus a historical artifact and these are then topic of this chapter.

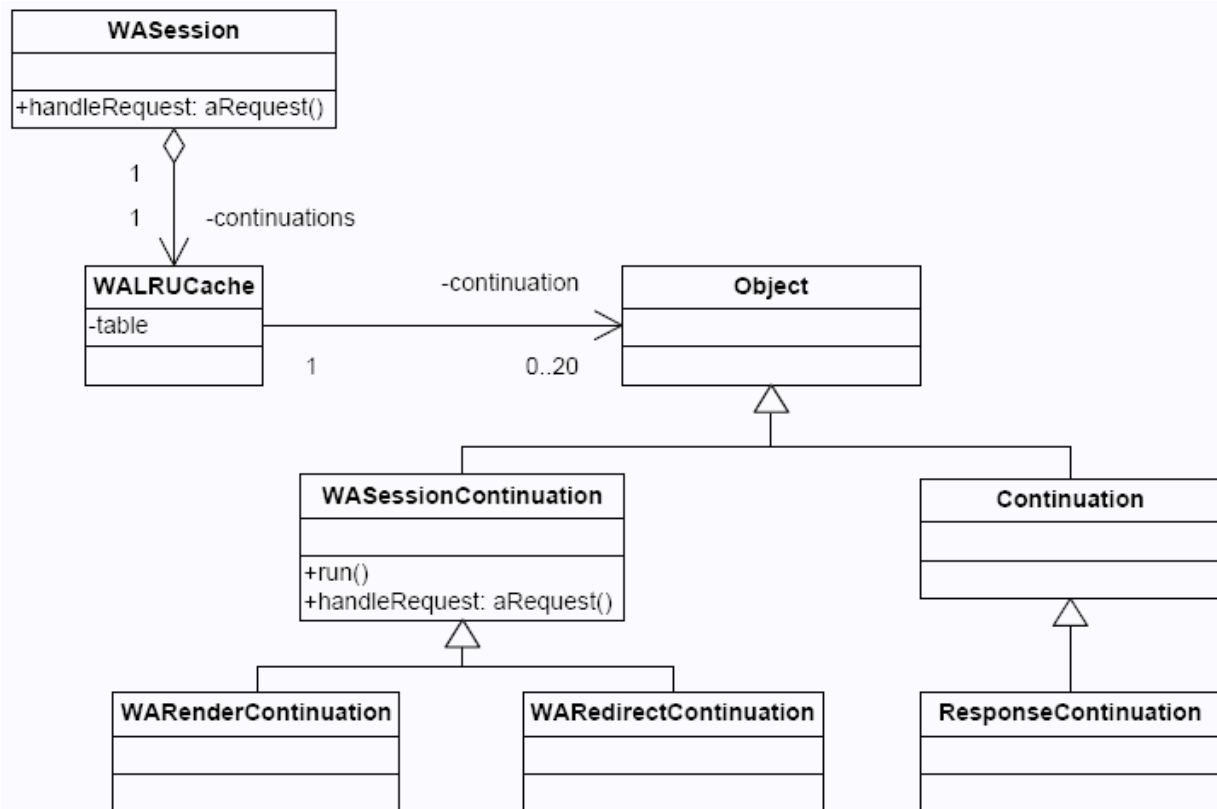


Figure 11.2: Session, Handlers and a Continuation

Without the middle part of the method **#performRequest:** (irrelevant in this discussion), you can see two different return statements and you could think there are two paths the request can go along, but in reality there are four. Think about it! You know the answer, because the first path is the default one without an action field (**_k**) and all the others are depending on the type of the stored **continuation**. There are three different types that can be saved in the session cache (**continuations**) **WRedirectContinuation**, **WRenderContinuation** and **ResponseContinuation**. A response continuation is rarely used as a real continuation and so we consider this one in section "Continuations". Consequently three paths are left over. The following code is not really helpful for a description and so we will visualize the following parts with UML activity diagrams. At important points we will link to relevant lines of code and classes. Each handler (redirect or render) has two separate paths (**#run** and **#handleRequest:**) where you can begin to explore the internal execution. Starting with **#performRequest:**, we follow the paths depending on the action field and the user interaction. For a better understanding we consider as example our ToDo Application (as a component, see the tutorial up to chapter 6) and we will sometimes show the current URL, a snapshot of the LRUCache and the reference URL of a link from the Web site to the Web server.

1. **Request without an action field (default)** - The first path is performed every time when an incoming request has no **_k** parameter, for instance **http://localhost:8080/todo**. From this it follows that the session is also new and the LRUCache itself is empty. Figure 11.3 shows the first path starting from **WSession>>#performRequest:** and calling the method **self start: aRequest.**

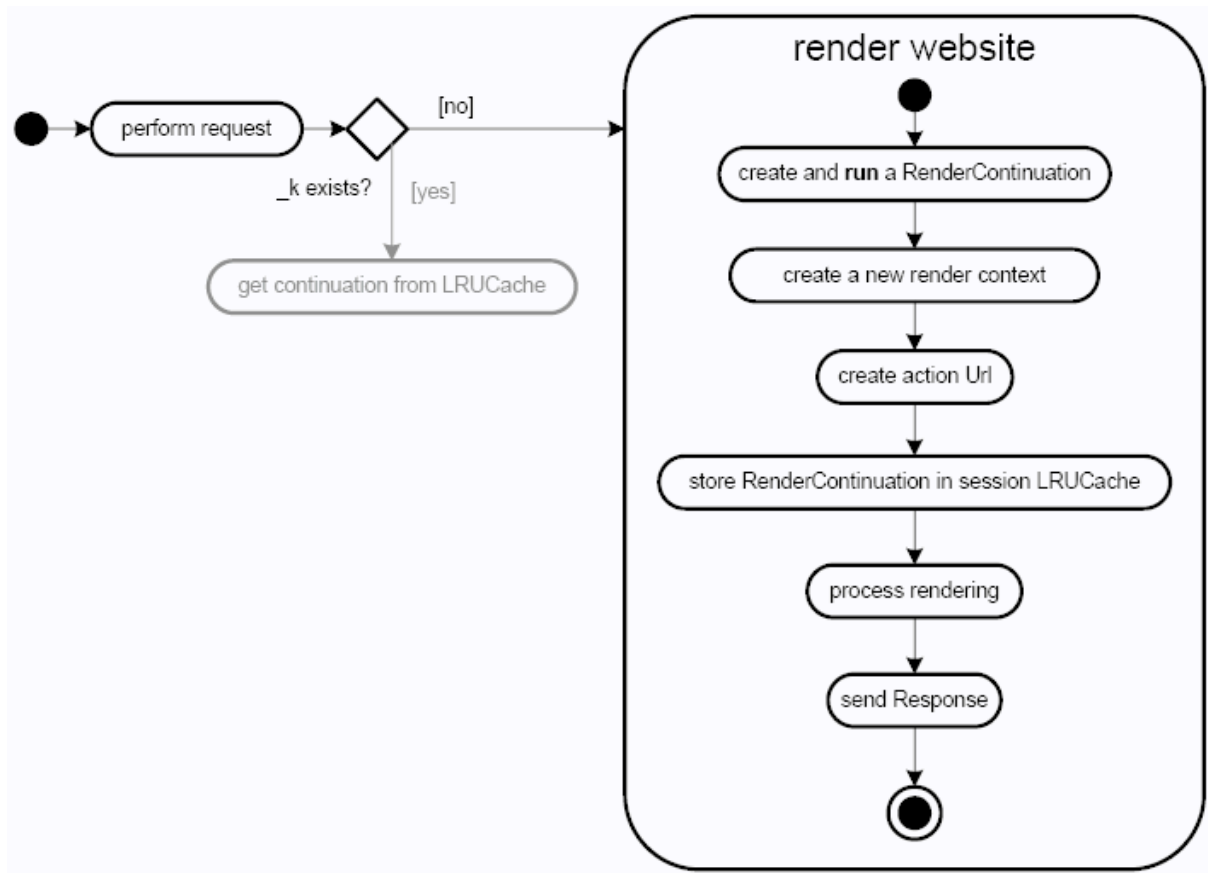


Figure 11.3: Request processing without an action field

Generally said: The method does nothing else other than rendering the Web site. In detail a **WARenderLoopMain** is created which initializes the application, creates the top level component, informs each component that the session started and starts a **WARenderLoop** to handle the request. This, in turn, creates and runs a **WARenderContinuation**. All further processing is surrounded by a notification handler so the process can abort the rendering at every position in the code by sending a signal (like an exception). Hereafter the handler creates and stores a new **WARenderingContext**. The **context** consists of all important information around the rendering like the action URL, a callback dictionary or the output document stream. During the creation, the render handler (**WARenderContinuation**) is registered in the session cache named as a unique action field (**_k**) and the session is stored in the application with a unique generated handler field (**_s**) as well. With all these parameters the action URL could look like this **http://localhost:8080/todo?_s=ogeMItuNDCcngcSA&_k=hvcGZwDH**. With this URL is it possible to retrieve the present handler later (after user interaction) and to call the second execution path with **#value: aRequest**. There the render handler does not visualize the Web site again but performs callbacks on the basis of the previous rendered Web site (see also the second path and the section "Rendering Tree"). All links within the Web site refer to the action URL above and in addition a number to identify the callback (block closure) in the rendering context which should be executed when the user clicks on the link. Following the rendering process (see "Rendering Tree" and "Canvas") the handler sends the response back to the client. Now the Web server waits for the next user request.

2. **The user clicks on a link** - After the starting request has been finished users see a Web site in their browsers. To interact with the Web application they click on a link like the **All** button in our **ToDo** Application. The anchor refers to the current session, the old **WARenderContinuation** and the callback for the link. The URL could look like this: **http://localhost:8080/todo?_s=ogeMItuNDCcngcSA&_k=hvcGZwDH&1**. In this way a new request is created and sent to Seaside. After retrieving the user session the method **#performRequest** is called. The current action field parameter is set and point out the **old WARenderContinuation**. In this, the second path is executed with the method **#value**: (see **WASession>>#performRequest**: and Figure 11.4).

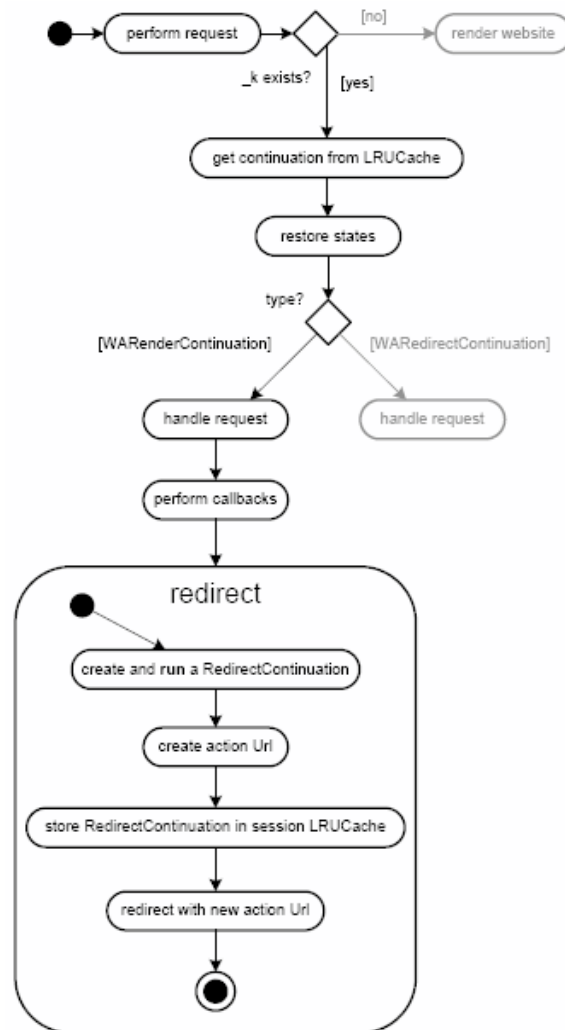


Figure 11.4: Request processing after the user clicks on a link

First the states are restored. These are states from the rendering tree or the model being saved in the render handler directly. With this users get a more realistic behavior when they click the back and forward buttons (see below **backtracking**). At this point of the execution the renderer no longer has the task to render the Web site but to perform the requested action from the user and to execute the registered callback to change the rendering tree or the model (for instance to increment a counter or to call another component). After all callbacks have been triggered the user would like to see the new part of the Web site. The simple solution could be to render with the same handler again, but this leads to many problems, like the registration of callbacks and the browser navigation from the user (back and forward button). To solve these and other problems, Seaside uses a cool trick: an internal redirect. For this purpose a new **WARedirectContinuation** is created and **#run**. Here the redirect handler stores itself in the LRU Cache of the session with a new action key and redirects to the Web server with a URL like http://localhost:8080/todo?_s=ogeMituNDCngcSA&_k=abZugHjk (same application entry point, **_s** and no callback identifier (number)). At this point the session cache has two entries, one redirect handler and one renderer. The redirect leads to the third possible path.

3. **Process an incoming redirect** - The redirect follows the same path as every time up to the method **#performRequest**. The action field is known and refers to our redirect handler. The second path of a redirect handler can be called with **#value**.

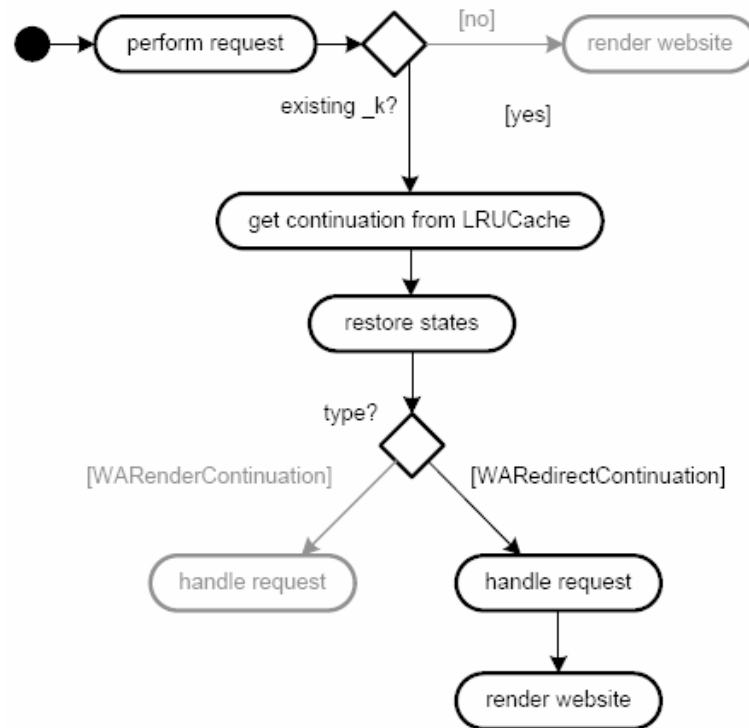


Figure 11.5: Request processing after an incoming redirect

In figure 11.5 you see the last of the three paths. At the beginning it is the same as the second path above. It gets the handler from the cache and restores the states. Then the redirect handler processes the incoming request and it does nothing else other than rendering the Web site anew. It is the same execution like in the first path, so it creates a new **WARenderContinuation** and a new **WARenderingContext**, registers itself in the cache and as already said - renders the Web site. In this moment the cache consists of three handler (two render and one redirect handler). Users see only the redirect handler URL in their address bar and so when they refresh the page, it is only rendered a second time with a new **WARenderContinuation** and the callbacks are **not** triggered again. We will not have any side effects when clicking on the refresh button or the link for a second time. Each time the Web site is rendered a new entry in the cache is inserted to retrieve the renderer for callback handling later. The cache is limited to about 20 items and so the memory usage is limited as well (see "Continuations"). Links always refer to the last render handler which has drawn the Web site.

That is it. We hope you have understood the main execution of the request/response processing. But one fact is not apparent, why is this all so complicated? It is actually simpler than it appears and will have many big advantages. At this point we have not considered the backtracking thus the navigation of a browser (back and forward button, history). You know that the refresh button with the redirect handler works. Seaside separates the callback processing and the Web site rendering. Each URL in the browser links to a redirect handler and each reference in an anchor links to a render handler, but how does the back button work? With the concepts described above it will work automatically. Every time users click on the back button they will see a Web site from the browser cache. Remember: Anchors refer to the renderer which has drawn them and this is the trick when the user clicks on an old link; the old renderer processes the callbacks, not the last one. With the ability to store the states on a handler directly, the callbacks process using the old data as well. This is a new user paradigm: **The back and forward button actually work!** Developers of Seaside Web applications do not have to exert much more effort to achieve this, they must implement a **#states** method which just returns an array with the objects to store. Should you be interested in more, have a look at the **WACounter** example. To reduce the backtracking path, use the method **#isolate**: which adds a **WATransaction** decoration to the workflow. This ensures that the component it decorates is not repeated once the transaction is completed.

In general, the request/response processing will work as explained above, but there are a few exceptions. For instance a **WATask** does not create content for a Web site but rather describes a workflow. Another simple trick is used to integrate the concept of a task: After the renderer has created the HTML root object, it sends an **#updateRoot**: to all visible components in the rendering tree (**WARenderContinuation>>#processRendering**:). This method is overwritten in **WATask** and the task registers the own **#go** method as a callback and redirects to this callback. So the render handler processes the callbacks first. Then the path is the same as was described in part two (**The user clicks on a link**).

This section was not easy to understand but if you know the main processing you can see the big advantages of Seaside. The browser's navigation works correctly and you can develop your Web application just like a desktop one. Only one disadvantage still exists: The URL is not RESTful. Seaside would not be Seaside if it did not have an answer. For more information see chapter "Last but not least - Static URLs".

Session Handling

Web applications rarely present static content exclusively, instead they offer interaction facilities for several users to work with the Web application at the same time. While interacting with the offered functionality, users will generate user specific data, for instance user names or individual settings. Additionally, a user may interact with the Web application in different sessions in several browser

instances and can follow a different path of interaction in each of them. Therefore session specific data, generated for each session, has to be managed by the Web application.

In order to handle these data the Web application needs to be able to determine the session of the current context, and to access its data globally. Of course, the value of the current session has to change at the same point in time when the current session itself changes. Generally, the current session value has to be globally accessible and dynamically changed according to the current context.

Smalltalk and especially Squeak do not provide a concept of dynamically scoped variables themselves. Therefore Seaside implements an own concept of dynamically scoped variables by the means of the **WADynamicVariable** class. In Smalltalk everything is an object, consequently the current session value is an object itself implemented in the class of **WACurrentSession**. It holds a session object of type **WASession**, embodying the current session and its data.

The dynamically scoped variables are built up on the existing mechanism of exceptions. This mechanism provides global access to different behavior according to the particular context. Exceptions can be accessed from everywhere to be thrown and to define the part of a routine, for instance a block in which context a special kind of exception should be treated in a specific way. In general, with exception, context specific behavior can be defined from any place in the code. Therefore exceptions are a suitable basis for realizing the concept of dynamically scoped variables.

In order to use the basic capabilities of an exception the inheritance hierarchy, shown in figure 11.6, is implemented. Due to the fact that a dynamically scoped variable is not related to an error, that should stop the execution it inherits from **Notification**, which is an exception that is resumable by default.

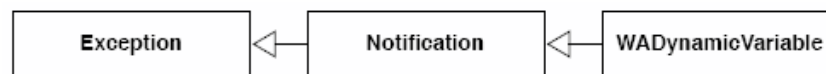


Figure 11.6: Class Hierarchy for WADynamicVariable

Due to this inheritance hierarchy a dynamically scoped variable just has to define some additional behavior to be used as a variable. Of course a variable has to provide a mechanism to access its value. The value of a **WADynamicVariable** can be retrieved by sending the message **#value**. Since it is dynamically scoped, it holds different values according to the current context. In order to determine the value for this context, the **WADynamicVariable** adopts the behavior of an exception and raises a signal, as shown in **WADynamicVariable>>#value**, when it is queried for its value.

```

value
  ^ self raiseSignal
  
```

WADynamicVariable>>#value

This signal is now to be handled, ideally with the value specified for the context in which this signal was emitted when the value of the **WADynamicVariable** was requested. As shown below, the context specific value can be defined by **WADynamicVariable class>>#use:during:**.

```

WADynamicVariable
  use: 'Seaside'
  during: [self complexCalculation].
  
```

Definition of a context specific value

When **WADynamicVariable>>#value** is called within **complexCalculation**, or a method invoked in this context, it will evaluate to **'Seaside'**. This behavior is again realized by using the exception concept. Generally, the exception handling for a given block context can be specified as shown below.

```

aBlock
  on: MyException
  do: [:n | n resume: anObject].
  
```

Exception Handling for aBlock

The block is executed and if, during the execution, one or more exceptions of type **MyException** are thrown then emit a signal. Each exception **n** is handled with the handler action of **[n resume: anObject]**, which defines that the execution will be resumed by returning **anObject**. The listing below shows the implementation of **WADynamicVariable class>>#use:during:**. The exception handling mechanism is used to define that the emitting of the signal that occurs when the **WADynamicVariable>>#value** gets evaluated inside the provided block, will be handled by resuming the execution with returning the provided object as context specific value.

```

use: anObject during: aBlock
  ^ aBlock
    on: self
    do: [:n | n resume: anObject]
  
```

WADynamicVariable class>>#use:during:

The interesting question is in which context the signal was emitted and whether a **#use:during:** handler was specified for that context. This question can be answered due to the reflective capabilities of Smalltalk. The virtual machine provides access to the

current context by the means of the pseudo variable **thisContext**. Each **#on:do:** handler context on the execution stack is queried to handle the signal. If no corresponding handler can be found the method **UndefinedObject>>#handleSignal:** will be called.

```
handleSignal: exception
  ^ exception resumeUnchecked: exception defaultAction

UndefinedObject>>#handleSignal:
```

The signal will finally be handled by the **UndefinedObject** that resumes the execution with the exception's **#defaultAction**. In case of **WACurrentSession** the **#defaultAction** returns nil, which means the variable is unset.

Although the dynamically scoped variables in Seaside are based on the powerful concepts of exception handling and introspection their use for the session handling is interestingly brief.

```
responseForRequest: aRequest
  currentRequest := aRequest.
  ^ self withEscapeContinuation: [
    WACurrentSession
      use: self
      during: [
        self withErrorHandler: [
          self performRequest: aRequest]]]

WASession>>#responseForRequest
```

The new session simply ensures that it becomes the current session by setting the value of the dynamically scoped variable **WACurrentSession** to **self** for the processing of the request in **self performRequest: aRequest**.

Rendering Tree

In the section Request/Response Processing we have ignored the rendering part because it fills a separate chapter. This section is concerned with the rendering itself and the internal structure of our Web applications. Also we do not consider the canvas and how the HTML is drawn (for more see the section "Canvas").

The first source code we are having a look at is **WASession>>#processRendering:**. The method is called after the context has been created in **#render** and it has got the current response object as parameter.

```
processRendering: aResponse
  | document htmlRoot |
  document := self newDocumentOn: aResponse.
  context document: document.
  htmlRoot := self newHtmlRoot.
  document open: htmlRoot.
  self root decorationChainDo: [ :each | each renderWithContext: context ].
  self writeOnLoadOn: document.
  document close: htmlRoot.
  context release

WASession>>#processRendering:
```

At first a new document stream (**WAHtmlStreamDocument**) is created on the response object and then stored in the current context. A **WAHtmlRoot** consists of all important information that should be written to the HTML head like the title, links to external CSS or additional scripts. A new instance of this information is written to the document. After that the real rendering is done, more about it later. The last three lines write additional scripts on the document stream, close the document with the last HTML tags (**</body></html>**) and release the context.

A Web application is built of components and these together result in a rendering tree. In figure 11.7 you can see a modified example of our **ToDo Application**.

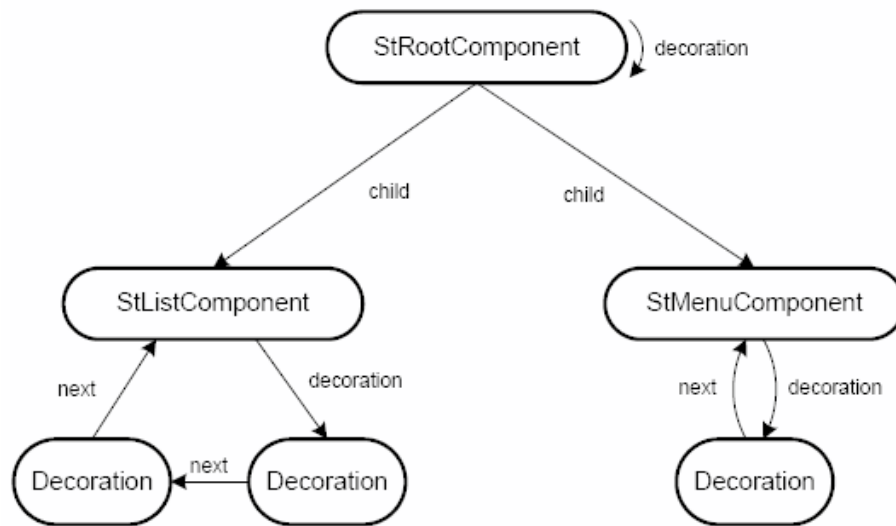


Figure 11.7: ToDo Application Rendering Tree

The current state of the user interface is represented by a tree of **WAPresenter** instances like **WAComponent**, **WADecoration** or **WATask**. The tree is walked several times in the rendering process. Starting from the root element it walks through the tree recursively. Therefore all components have to register their subcomponents in the **#children** method. In the example image the **StRootComponent** is the starting point and it has two children (**StListComponent** and **StMenuComponent**). So the render order is root, list and menu component. In addition, each component can have various decorations changing the look or the behavior of the owner. A decoration has a **next** pointer to the next one or to the component itself. In the rendering order decorations are processed before their owner. For instance the list component's decorations are rendered before the list. The complete rendering order looks then like root, decoration one and two, the list itself, the decoration of the menu and the menu component. A special decoration can influence the rendering tree directly. A **WADelegation** changes the render order to another component. This concept is important in the **#call:** and **#answer:** mechanism we will consider in section "Continuations" in detail.

Let us go back to the method **#renderWithContext:**. Now we can see that all decorations from the root component get the message with the context object. With the background knowledge of the rendering tree you can see that this is the starting point for walking through the tree. Each decoration points to another one and then to the component itself.

```

renderWithContext: aRenderingContext
| html callbacks |
callbacks := aRenderingContext callbacksFor: self.
html := self rendererClass context: aRenderingContext callbacks: callbacks.
(self showHalo and: [ aRenderingContext isDebugMode ])
  ifTrue: [ (WAHalo for: self) renderContentOn: html ]
  ifFalse: [ self renderContentOn: html ].
html flush

```

WAPresenter>>#renderWithContext:

In the method **#renderWithContext:** you will see an old good friend, **#renderContentOn:**, the entry function for each component from the view of a Web developer in Seaside. But first the method creates a new callback dictionary for the object and a new **WARenderCanvas** (see section "Canvas") which is used to render the Web site. You know this object as the parameter **html**. At the end the renderer flushes the last input to the document. When the renderer finds a **html render: aComponent** it traverses the next component recursively in respect to the decoration chain of this one. This goes on that way until the complete rendering tree is processed.

Continuations

The concept of continuations was invented in 1974 by C. Strachey and C.P. Wadsworth in order to give a formal meaning of the **goto** statement. Continuations are a way to represent the **rest of the computation** at a given point in the program, thus they represent the control state given to a point in the computation. Therefore the current execution context is objectified. This means the complete call stack, local variables and the instruction pointer during the execution of a method are stored in the continuation object. The global state and objects on the heap memory are not saved. At any point during the program execution, the continuation can be called and the control is transferred back from the caller to the point where it has been captured. For that reason, the concept is more than a simple **goto** statement and it seems to be very powerful for behavioral reflection. Continuations were mainly used in dynamically typed languages for cooperative threading, coroutines or writing iterators (C# iterators or Python generators). In the last few years a new use case has been established. Many problems in Web development, like stateless HTTP or browser navigation, can be solved with the help of continuations.

There are different kinds of continuations. The most important ones are the following three:

Delimited (partial) continuations

are a compromise between efficiency and expression. Developers define a boundary on the execution stack. The continuation is valid as long as the execution stack does not leave it. The costs are less than a first-class continuation because they only need to copy the subsection of the stack up to the boundary and the continuation can be called an arbitrary number of times.

One-shot continuations

are fast and implement common use cases, for instance non local exits, non-blind backtracking and coroutines. They can be called once and the system has only to store the instruction pointer. So they are valid as long as the execution does not leave the current continuation frame.

First-class continuations

are the most general kind. They need to copy the full execution stack and have no limitation in relation to the lifetime or the number of invocations thus they are the most expressive and expensive ones.

From the beginning on Scheme has been the first choice for a programming language to implement the concept, but in the course of time continuations were added to other languages like Smalltalk, Ruby and Python. This section explains continuation details for Smalltalk, the general use in Web development and particularly for the Web framework Seaside.

Continuations in Smalltalk

With Seaside, Smalltalk got its own first-class continuation implementation. Developers can obtain a new continuation object with the message `Continuation currentDo:[:cc | "Do something"]` which is equal to `#callcc:` in other languages. The current execution context, with the instruction pointer pointing immediately behind this call, is captured in the **current continuation** (block parameter `:cc`). In detail the whole execution stack is traversed recursively and serialized to an array saving the instance variable **value** of the `cc` object. Within the block, developers can store the continuation for later use or they can execute some code and invoke the `cc` directly. There are two ways to return to the point where the computation has been captured. In both cases, the current context is destroyed and the execution continues at the point where `Continuation currentDo:[:cc | "Do something"]` was called. The first method (`#value: anObject`) only returns the object, the second one (`#invoke: aBlock`) evaluates the block within the captured execution context and returns the result of the block. Remember, the instruction pointer refers to the point behind the capturing and so the final argument replaces the position of the continuation in the code.

Two examples should demonstrate the use of continuations in Smalltalk:

```
| continuation return |
Transcript clear.
return := Continuation currentDo: [:cc | continuation := cc. false].
return
  ifFalse: [
    Transcript
      show: 'Hello';
      space.
    continuation value: true.
    Transcript show: 'You'].
Transcript show: 'World'.
```

First Smalltalk Continuation Example

The listing above shows an example of how a continuation is created and stored in the local variable **continuation**. The block returns **false** and the computation goes the usual way into the condition path. First the word **Hello** is printed to the transcript and **continuation** is invoked with the value **true**. Thus the execution continues from the stored point in the computation. Then the value of **return** is set to true and the interpreter takes the same way again except the condition is evaluated to true and the program exits with the output from **World**. In this example the line `Transcript show: 'You'` is never called so the final result in the console is **Hello World**.

```
| result |
Transcript clear.
result := Continuation currentDo: [:cc |
  "Do Something"
  cc value: 'Hello World'.
  "Do Something Again"].
Transcript show: result asString.
```

Non Local Exit Example

The second example demonstrates a possible use of a non local exit (see also escape continuation in Seaside). The continuation will not be saved in a variable, instead a large calculation is done in the current continuation frame. After that the object `cc` is invoked with the argument **Hello World**. The context switches back to the saved point and the result is set to the string. Finally **Hello World** is shown on the console. Here the starting point of a calculation is saved and the continuation can be invoked anywhere in the process. This cancels the complete computation and returns the result to the start. The second **Do Something Again** is never called.

Smalltalk has many reflective capabilities. One of them is the pseudo-variable (Pseudo-variables are reserved identifiers that are similar to keywords in other languages) **thisContext**. It enables the access to the whole execution stack. The reification of the stack consists of instances of **BlockContext** and **MethodContext** storing all necessary information. The proper implementation of continuations seems to be simple with help of this variable. The following two methods are describing the core concept of the **Continuation** class.

```

initializeFromContext: aContext
| valueStream context |
valueStream := WriteStream on: (Array new: 20).
context := aContext.
[context notNil] whileTrue:
    [valueStream nextPut: context.
    1 to: context class instSize do: [:i |
        valueStream nextPut: (context instVarAt: i)].
    1 to: context localSize do: [:i |
        valueStream nextPut: (context localAt: i)].
    context := context sender].
values := valueStream contents.

```

Continuation>>#initializeFromContext:

The listing shows the main method for serializing the current context. The parameter is a representation of the current execution stack without the top element (the continuation itself). The context is serialized to a new write stream therefore the stack is iterated through all context objects. At the end the stream's contents are saved in the instance variable **value** as an array. The method is called every time a point in the computation is captured by a continuation.

```

value: anObject
    "Invoke the continuation and answer anObject as return value."
    self terminate: thisContext.
    self restoreValues.
    thisContext swapSender: values first.
    ^ anObject

```

Continuation>>#value:

This is the internal execution when a continuation is invoked. When returning to the captured point, the current context is out of date and so it can be destroyed by swapping **nil** values on the stack. Afterwards the serialized array values are restored and the context switches back. The message **#swapSender:** replaces the receiver's sender with the coroutine, more precisely with the stored point in computation.

Continuations in Web Development

In general, the main problems of Web applications are the stateless HTTP, a suitable mapping for the browser navigation and multiply submitted requests. Common approaches like the page-centric design separate the whole Web application into individual parts. They maintain the different states and requests with each client circuitously by the means of many hidden fields or GET parameters in the URL. Such applications are difficult to understand and modify.

With the help of continuations it is possible to solve these problems and to reach the same level as in desktop application development. The main idea is to use a language supporting first-class continuations and to make HTTP behave like a stateful protocol. Developers can return to well-known concepts and the Web application is defined as a single program where continuations take care of sliced execution.

For instance, when the Web application needs input from the user, it captures the current continuation. This object belongs to the corresponding user with a unique identifier and is stored in the Web server cache using the same approach that used to store session data. After a while the user responds, the captured continuation is restored and the input is returned as the value of the continuation.

First-class continuations can be invoked several times and with the uniqueness of each continuation the Web server gets the use of the browser navigation for free. Now the user can restore every continuation for an arbitrary moment in the previous conversation.

Every concept has advantages and drawbacks. On the one hand, the problem of stateless HTTP is solved, there are no side effects when a user submits the form a second time, the browser navigation works for free and the Web development is dramatically simplified (using standard development tools, easy to understand the program and to focus on solving the main problem). On the other hand, continuations can be expensive as the execution stack is cached several times, in most cases the URL is not RESTful and the working navigation will disorient some user experiences. In addition, continuations are hard for developers to understand.

To summarize, the continuation-based concept represents a real advancement in Web development frameworks and looks promising. The first implementations have been created in several languages with the Persistent Server-Side Scheme Interpreter, Continuity (Perl), ViaWeb (Lisp) and Seaside (Smalltalk).

Continuations in Seaside

Seaside has a reputation for using continuations for the control flow, thus it offers a procedural view to the Web developer. In previous versions, continuations were also used for transforming the asynchronous request/response cycle of HTTP to a linear flow. By using a clever refactoring it was possible to simulate the same behavior with normal objects and to reduce the usage of continuations (see section "Request/Response Processing"). Before the modification, a single user session has allocated about 200 KB and after the change the memory requirements were reduced by 75 percent. This illustrates that there are other solutions, too. But Seaside is not without continuations. Currently three different kinds are left in the framework. Figure 11.8 shows the class structure of **Continuation**. The subclasses have new names to distinguish them from the others thus they do not implement any additional

functionality.

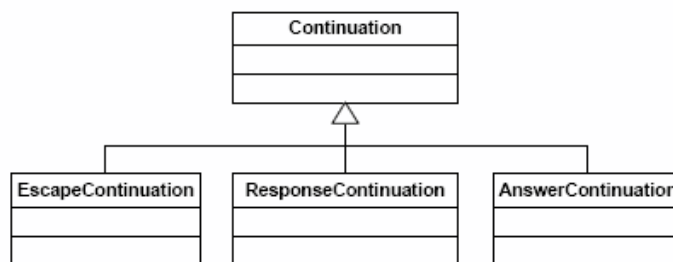


Figure 11.8: Continuation Class Hierarchy

EscapeContinuations

are only used once in the system and as its name implies they implement a non local exit (see also the example listing above). They can escape the current context to a surrounding one, especially used in the method **WASession>>#responseForRequest:**. Requests pass this method and after that they have the ability to abort all further execution and immediately return a response from everywhere in the request processing. The continuation is stored in the instance variable **escapeContinuation** of a session and can be called with **WASession>>#returnResponse: aResponse** which invokes the escape continuation with the response.

AnswerContinuations

are an important part of the **#call:/#answer:** concept. In Seaside a Web application consists of many components. Each of them can be replaced by another component temporarily (**#call:**). This is useful to show users an input form instead of the result list. So they fill in the input and the result is sent back to the Web server where the temporary component is removed and the result gets set to the previous component (**#answer:**). This concept provides the facilities to model a control flow over several pages with one piece of code. With **self call: aComponent** developers can pass the control from the receiver to another component and within the **#call:** method an **AnswerContinuation** is created to save the current point in the computation. The component behind **self** gets a **WADelegation** as decoration, calling the renderer to ignore this component and to render only the delegate. In addition the **aComponent** gets a **WAAnswerHandler** as decoration which stores the **AnswerContinuation** object. If the delegate executes **self answer: anObject**, the delegation is removed and the continuation is invoked with the answer object to continue with the execution after the **#call:** message.

ResponseContinuations

save the point of computation to continue there after a manual redirect from the developer. For internal redirects the server uses a handler without a continuation (**WARedirectHandler**). So there are a few use cases left over, for instance adding additional properties to the incoming request and using them later on like a special cookie or the page expired message when users access an invalid page. This kind of continuation is the last stored in the real session cache because it captures the current execution of the user and continues after the redirection at this point (**WASession>>#performRequest:**). **WASession>>#respond:** is used to create an own redirection.

Continuations are a very powerful concept and a candidate to solve specific problems. Often it is hard to solve a problem without them, but on the other hand it is hard to understand the implementation with them. Their use in Seaside shows that continuations are not the one and only solution for all specific Web development problems. In previous Seaside versions continuations consumed so much memory that a refactoring was mandatory. Thus, they were removed from many areas of the Web framework and replaced by a sophisticated process for the request/response cycle. But for some problems continuations are essential, for instance the concept of **#call:/#answer:**. Primarily with the help of continuations it was possible to implement the mechanism very easily and to provide the facilities to model a control flow over several pages with one piece of code.

Canvas

Every Seaside developer knows **renderContentOn: html** and how to implement this method. In this section we have a look at the **WASessionCanvas** and how the **html** creates valid XHTML output. In general, **html** is the canvas on which the developer draws the Web site.

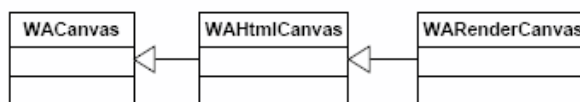


Figure 11.9: Canvas inheritance tree

On the one hand, figure 11.9 shows the inheritance tree of the canvases. From left to right, the **WACanvas** implements the general use of brushes which can be used to draw on a canvas (see below), **WAHtmlCanvas** adds an own XML dialect (XHTML) and knows all about HTML and last but not least the **WARenderCanvas** inserts the knowledge about callbacks to the canvas. The last class is always instantiated to the variable **html** and can be used by the developer for rendering.

On the other hand the brushes are used to draw the HTML on the canvas, the document and the final response to the user.

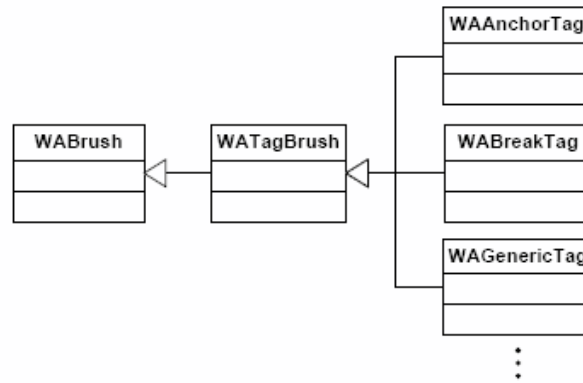


Figure 11.10: Brush inheritance tree

Figure 11.10 shows details from the brush inheritance tree. The top level class is **WABrush**, a generic brush that can be used on a **WACanvas**. In addition, the canvas and the brush both store the parent brush to produce nested entries on the output. **WATagBrush** is the abstract superclass for all XML element classes and adds generic behavior, attributes and common events used later by the subclasses. At the lowest layer many concrete classes implement the brushes for HTML, thus each tag is drawn with an own new brush. For instance, the method `html heading` instantiates a **WAHeadingTag** object used by developers to control the style and with the last message `#with:` the object is rendered on the canvas and the document itself. Let us have a look at some examples:

```

html: aString
  "Emit aString unescaped onto the target document."

self flush.
self document nextPutAll: aString displayString

WAHtmlCanvas>>#html:

```

WAHtmlCanvas>>#html: is shown above. This method is not the normal case because it is writing the given string on the document stream directly without any brush. But the interesting part is the `self flush`. If the current brush of the canvas is not nil, thus he is not written on the document, `#flush` will trigger the brush to close and write himself on the stream. This is done with the message `self with: nil` sent to an instance of a subclass from **WATagBrush** (see below).

```

with: anObject
  "Render anObject into the receiver.
  Make sure that you call #with: last in the cascade,
  as this method will serialize the tag onto the output document."

  self openTag.
    super with: [
      self before.
      anObject renderOn: canvas.
      self after ].
  self closeTag

WATagBrush>>#with:

```

The listing shows the main transform method from Smalltalk code to XHTML. At first the open tag is written on the stream. In a single tag like `
` the method `#isClosed` returns true and the open tag closes automatically. Developers create a break tag with the single line `html break`. This code sends no `#with:` message and the current brush is not yet written to the document. In this case the next `#flush` arranges it so that the current brush (break) is written on the stream before the new brush is rendered (see above `self with: nil`). At the end of each rendering process a final flush is sent to draw the last brush too. Back to the listing above, the next part calls `#super with:`. There the block is getting back to the canvas where nested blocks are rendered in the right order. The methods `#before` and `#after` can be overwritten to create additional content in the standard case that they do nothing. Because an object can be an instance of every Smalltalk class, the method `#renderOn: aRenderer` is implemented in many of them. For instance, **UndefinedObject** does nothing at all (in that way the code `self with: nil` works), **Object** writes the name of the class with `#text:` on the Web site (have a try and execute `html text: SmallInteger`) and **BlockContext** evaluates itself.

```

renderContentOn: html
  html horizontalRule.
  html heading
    level2;
    with: [html text: 'Hello World'].

Example>>#renderContentOn:

```


In the second example we will think about the theoretical concept again. There you see a simple Web site with a horizontal rule and a **Hello World** headline. The HTML should look like this `<hr /><h2>Hello World</h2>`. First a **WAHorizontalRuleTag** brush is created and set to the current brush. Before this happens a **#flush** message is sent to the canvas writing the last current brush to the document. Remember: This is just the case if the brush does not send **#with:** before explicitly. In our case there is no current brush at the moment. `html heading` creates a **WAHeadingTag** object and thereby flushes the previous heading tag on the stream. Remember again: **WAHorizontalRuleTag** overwrites the method **#isClosed** with true and just a single XML Tag is written. **#Level2** sets the size of the heading and **#with:** triggers the write process, thus the nested block `html text: 'Hello World'` is executed in the right order and the results looks like the expected one above.

The last topic of this section is about **callbacks**. Special brushes like **WAAnchorTag** can receive **#callback:** messages. In those a callback is registered in the current context and a number is added to the URL of the reference link. With this additional information the link is rendered to the Web site.

Summary

This chapter was the most difficult one to write and perhaps is also the most difficult to read. We hope you have learned something useful about the inner details of Seaside. We considered the most important concepts like the request/response processing, dynamic scoped variables and continuations. Furthermore you have learned what `_s` and `_k` in the URL mean, how the components are mapped to a rendering tree and how you can draw your HTML on the Web site. This description was just an introduction for Seaside's processing details. For more information have a look in the source code and create your own experiences.

[Back](#)

12 - Last But Not Least

What you are going to learn

- [Advanced Seaside](#)
- [Configurations](#)
- [Decorations](#)
- [Static URLs](#)
- [Seaside Unit Tests](#)
- [Coding Conventions](#)
- [Links](#)
- [Conclusion](#)

Advanced Seaside

In this chapter, we present additional concepts and mechanisms that did not make their way into our tutorial. These topics are important but optional, and they do not fit in with our ToDo Application. For that reason, we describe many of them in short and link to external resources. At the end of this chapter, you can find a Seaside link collection.

Configurations

We have discussed configurations in many of the previous chapters. Every time you did something on the config page, such as register an entry point, you altered the standard configurations of Seaside. The configuration framework is designed to allow the reuse of Seaside components in different applications. Configurations can be customized by the system administrator, and for a component, the configuration seems like a Dictionary-like collection of associations. The [Seaside documentation](#) shows you how to manipulate the existing ones and to create configurations for your own use.

Decorations

Seaside provides a decoration framework for components, so that you can modify the basic behavior or look of a component. The intent of the decorator pattern is in the following quote from the book "Design Patterns - Elements of Reusable Object-Oriented Software; Gamma, Helm, Johnson, Vlissides"

Decorator

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

Decorations can be added to a **WAComponent**. In chapter 11 "The Rendering Tree" you have learned how decorations are linked together. They are executed before the component itself and so they can influence the basic behavior or look. Decorations are independent of another and easy to reuse. With the message **#addDecoration:** you can add a new decoration to the component respectively to the decoration chain. Usually this will be done in the current **#initialize** method of our component. To implement an own decoration you must subclass **WADecoration**. You have three possibilities to alter the behavior by overriding one of the following methods:

#renderContentOn:

can be used as ever to render new html content around the decorated component. The minor difference is to call explicit **#renderOwnerOn:** to let the owner (possible another decoration or the component) render its html output. The developer can decide whether or when to send this message. Without sending it only the decoration is rendered.

#handleAnswer:

intercept the answer processing and you can modify or stop the answering of a call. For instance a component returns an object that must be validated before the caller can proceed. Therefore a decoration is ideal to do this.

#processChildCallbacks:

is a special case where the developer can alter the callback processing of the owner (mostly the component). This is useful to implement own behavior that can influence the whole processing like authentication (see **WABasicAuthentication**).

Seaside offers many built-in decorations which are either used internally or they are implementing common use cases for developers. The use of them is very simple encapsulated in the Web framework. For an overview have a look at all subclasses of **WADecoration**. Here we present in short three different kinds of decorations following the possibilities to alter the behavior or look of a component.

WAMessageDecoration

adds a heading at the top of the component. The method **#renderContentOn:** is overwritten with a heading tag followed by the **#renderOwnerOn:** method. To use this decoration you can create the class explicit and add it to your component in **#initialize** with **#addDecoration:** or you take simply the method **WAComponent>>#addMessage:** aString which does the part described above automatically.

WAValidationDecoration

is a good alternative to our approach for a login. It implements a new answer handler which tests the given object from the **#answer:** method with a validation block. If an error occurs the object can raise a **WAValidationNotification** with the message **Object>>#validationError:**. In our case every **StUser** object could validate itself and send an error if the login fails. This error will be shown to the user in a single line above the input form (compare **StMessageComponent**). This decoration can be added to a component with the method **#validateWith: aBlock**.

WATransaction

is an example for a decoration which implements an own **#processChildCallbacks:** method. The decorated component is part of a transaction that means it is not repeated once the transaction is completed or in other words it does not support rollbacks. For that reason the decoration must control the callback processing to redirect an incorrect request to another page. For instance when users have submitted a form you do not want them to use the browser's back button. Normally you do not use this class directly but rather you should take the method **#isolate:** of **WATask**.

To illustrate the use of decorations for our ToDo Application we want to add a little copyright string to all main components (**WALoginComponent**, **WRegisterComponent** and **WALoggedInComponent**). The first approach could be to implement a new component with only one **#renderContentOn:** method and a copyright string. All the components above must insert in their complex logic a new component; change the render method, and so on. Remember an easier way is a new decoration. First we create a new class for our copyright decoration.

```
WADecoration subclass: #StCopyrightDecoration
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'STTutToDoApp'
```

StCopyrightDecoration

Remember to inherit from **WADecoration** and not from **WAComponent**. Then we only need to implement the **#renderContentOn:** method.

```
renderContentOn: html

self renderOwnerOn: html.
html div
  class: 'copyright';
  with: [html
    html: '©';
    text: ' Copyright ',
        Date today year asString,
        ' Software Architecture Group'].
```

```
StCopyrightDecoration>>#renderContentOn:
```

That is it. We call the owner to render first and then we render our copyright html string in an own div with the CSS class "copyright" (see chapter "External Resources"). As a little gimmick we generate the current year for the user. On the other side we must add the decoration to our three components (see above). Therefore we call only the method **#addDecoration:** with a new instance of **StCopyrightDecoration** in our initialize methods. The listing below shows the new initialize method of **StLoginComponent**.

```
initialize

  super initialize.
  self addDecoration: StCopyrightDecoration new.

StLoginComponent>>#initialize
```

You could see that decorations are an option to change the look and the behavior of components with the additional advantage of loose coupling.

Another introduction to this topic was written by [C. David Shaffer](#).

Static URLs

Sometimes, a Web application need a static URL to link or exchange content with other users. In Seaside, this problem is not trivial because all requests are generated automatically. The solution is to overwrite the method **#initialRequest:** which is called every time a new session is started. All visible presenters (components) receive this message with the request as argument and can check it for parameters. For instance, our tutorial has URLs that can be bookmarked. Please try to call the tutorial with the parameter chapter and a number, like <http://www.hpi.uni-potsdam.de/swa/seaside/tutorial?chapter=10>. You will see that it starts with the right chapter. The following source code implements this behavior in our tutorial. Take care of the class **StRootComponent**, which has nothing to do with the ToDo Application.

```
initialRequest: aRequest

  | theClass |
  super initialRequest: aRequest.
  aRequest fields at: 'chapter' ifPresent:
    [:chapterNumber |
      theClass := StChapter allSubclasses
        detect: [:each | each number = chapterNumber asNumber]
        ifNone: [].
      theClass ifNotNil:[self chapter: theClass new]].

StRootComponent>>#initialRequest:
```

Seaside Unit Tests

David Shaffer has extended the SUnit framework to support testing Seaside components. His tutorial describes the usage of [SeasideTesting in detail](#).

Coding Conventions

The three most important [coding conventions](#) (Portability, Formatting, Presentation) are described on the Seaside homepage. If you are new to Seaside, try to do your best to observe these guidelines.

Links

Here are a few links to further material on the topic of Seaside. We think these are good starting points for your own needs.

Weblogs about Seaside<http://www.seaside.st/community/weblogs/>**Mailing List**<http://www.seaside.st/community/maillinglist/>**Seaside**<http://www.seaside.st/>**Other tutorials**<http://www.seaside.st/documentation/tutorial/>**Documentation**<http://www.seaside.st/documentation/>**Conclusion**

We hope that this tutorial was a useful introduction to Seaside, and that it has helped you to implement your first own application. For feedback, please send an email to the authors.

And now, we wish all of you a great time on the Seaside.

[Back](#)

About Us

What you are going to learn

- [Authors](#)
- [Feedback](#)
- [Software Architecture Group](#)
- [Copyright](#)
- [Imprint](#)

Authors



**Michael
Perscheid**



**David
Tibbe**



**Martin
Beck**



**Stefan
Berger**



**Peter
Osburg**



**Jeff
Eastman**



email address: first.last@hpi.uni-potsdam.de

Feedback

For feedback please send an email to Michael Perscheid.

[michael.perscheid\(at\)hpi.uni-potsdam.de](mailto:michael.perscheid(at)hpi.uni-potsdam.de)

Software Architecture Group

The Software Architecture Group, led by Prof. Dr. Robert Hirschfeld, is concerned with fundamental elements and structures of software. Methods and tools are developed for improving the comprehension and design of large complex systems.

[Software Architecture Group](#)

Copyright

Copyright © 2008 by Michael Perscheid, David Tibbe, Martin Beck, Stefan Berger, Peter Osburg, Jeff Eastman, Michael Haupt, and Robert Hirschfeld

The contents of this book is protected under Attribution-Noncommercial-No Derivative Works 3.0 Unported license.

You are free:

to Share

to copy, distribute and transmit the work

under the following conditions:

Attribution.

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial.

You may not use this work for commercial purposes.

No Derivative Works.

You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this Web page <http://www.creativecommons.org/licenses/by-nc-nd/3.0/>
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

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

Imprint

Hasso-Plattner-Institut fuer Softwaresystemtechnik GmbH

Software Architecture Group

Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam

Phone: +49 (0)331 5509-220

Fax: +49 (0)331 5509-229

E-mail: [office-hirschfeld\(at\)hpi.uni-potsdam.de](mailto:office-hirschfeld(at)hpi.uni-potsdam.de) (For *feedback* mail to [michael.perscheid\(at\)hpi.uni-potsdam.de](mailto:michael.perscheid(at)hpi.uni-potsdam.de))

Internet: <http://www.hpi.uni-potsdam.de/swa/>

Authorized Representative Managing Director:

Prof. Dr. Christoph Meinel

Registry Office: Potsdam District Court

Register Number: HRB 12184

VAT Identification Number in accordance with § 27 a of the German VAT Law :
DE812987194

Persons responsible for content in accordance with §6 MDStV:

Prof. Dr. Robert Hirschfeld