From Technologies to Solutions

ICEfaces 1.8: Next Generation Enterprise Web Development

Build Web 2.0 Applications using AJAX Push, JSF, Facelets, Spring, and JPA

Rainer Eschen

PACKT PUBLISHING

ICEfaces 1.8

Next Generation Enterprise Web Development

Build Web 2.0 Applications using Ajax Push, JSF, Facelets, Spring, and JPA

Rainer Eschen

BIRMINGHAM - MUMBAI

ICEfaces 1.8

Next Generation Enterprise Web Development

Copyright © 2009 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, Packt Publishing, nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2009

Production Reference: 1261009

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847197-24-5

`www.packtpub.com`

Cover Image by Filippo Sarti (`filosarti@tiscali.it`)

Download at WoweBook.Com

Credits

Author
Rainer Eschen

Reviewers
Dhrubojyoti Kayal
Ted Goddard

Acquisition Editor
Sarah Cullington

Development Editor
Darshana Shinde

Technical Editor
Tarun Singh

Copy Editor
Sneha Kulkarni

Indexer
Hemangini Bari

Editorial Team Leader
Abhijeet Deobhakta

Project Team Leader
Priya Mukherji

Project Coordinator
Zainab Bagasrawala

Proofreader
Sandra Hopper

Production Coordinator
Adline Swetha Jesuthas

Cover Work
Adline Swetha Jesuthas

Download at WoweBook.Com

About the Author

Rainer Eschen has a degree in computer science. He looks back on more than 20 years of programming experience. Since 1994, he works as an IT professional with a focus on consulting and architecture. He has also been part of "the source" for three years, working as Sun Sales Support Engineer and Sun Java Center Architect at Sun Microsystems, Germany.

Today, he is focused on architectures using the Spring framework in conjunction with the Java Persistence API (JPA). ICEfaces is a central part of the web development for several years now. Meanwhile, the development of the Facelets components takes up a lot of space.

Besides ICEfaces, the use of Flex has become increasingly important to him. After a successful integration of ICEfaces, Flex, and Spring using Spring BlazeDS, he currently designs pure Flex and AIR clients with Spring backends. His design allows the ICEfaces and Flex/AIR clients to use the same Spring backend.

User experience and software ergonomics are constant companions in all his decisions. If possible, he uses multimedia technologies to improve the user experience of web applications. For this, he had a deeper look at semiprofessional audio, video, and 3D production concepts over the last years. According to him, the most promising are humanoid avatars with lip-sync animations using computer-generated voices. They are a cheap but efficient tool for an optimal user acceptance.

Download at WoweBook.Com

Acknowledgments

First of all, a big thank you to my wife Silvia Regina, my son Lucas Eric, and my daughter Lisa Estelle. A lot of evenings and weekends were blocked for writing this book. Many thanks for the support, understanding, and love. I love you all.

This book exists because I won the ICEfaces Technical Blog Award in October 2008. I got a nice T-shirt and my first Apple gear, an iPod Touch, from ICEsoft. Thanks to Tracy Gulka for the hard work to get it right before Christmas.

In November 2008, my editor—Sarah Cullington—asked me to "develop a book that shows readers how to create and deploy rich Internet applications with ICEFaces." I never thought about writing a book on ICEfaces. Although ICEfaces got momentum in 2008, a useful introduction was still missing. Actually, it was a good opportunity to gain more experience in writing and to have my first book on the market. So, I took the chance of writing the first ICEfaces book. Sarah was a great help to get all this done. A special thanks to the rest of the Packt Publishing team that helped me realize this book. You did a great job. I am pretty happy with the result. It was a pleasure to work with you on this book.

Actually, this book is a documentation of a full-blown ICEfaces web application example, which I have named ICEcube. So, for two-thirds of my scheduled time, I was a developer. It was a very tough schedule because the result had to be something really useful for the Enterprise AJAX developer. Micha Kiener, Head of Research & Innovation, mimacom, and Ted Goddard, Senior Software Architect, ICEsoft, put in ideas that helped to get things running. Thanks for your time (despite the heavy project load).

I also want to thank Wilbur Turner, VP of Sales and Customer Support, ICEsoft, as well as Robert Lepack, VP of Marketing and Product Management, ICEsoft, and also David Krebs, CTO, mimacom, for the open communication and their willingness to support the book.

Download at WoweBook.Com

Last but not least, I thank the ICEfaces community for its support over the past few years. Although I was not a very active forum member, I hope the book can give something back. Special thanks to all the members who made suggestions for the outline. A lot of your suggestions were pretty interesting and worth writing about. Sadly, this is a beginner's book and the number of pages is limited. I will keep all this in mind for my future projects.

Download at WoweBook.Com

About the Reviewers

Dhrubojyoti Kayal works as an Agile developer architect for Capgemini Consulting. He has more than eight years of experience designing and developing Enterprise Java applications and products. He is an open source evangelist and author of the book *Pro Java™ EE Spring Patterns: Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework – Apress* (Aug 2008).

Prior to Capgemini, Dhrubojyoti worked with Cognizant Technology Solutions, Oracle, and TATA Consultancy Services.

Ted Goddard is the Chief Software Architect at ICEsoft Technologies and is the technical lead for the JavaServer Faces Ajax framework, ICEfaces. Following a PhD in Mathematics from Emory University, which answered open problems in complexity theory and infinite colorings for ordered sets, he proceeded with post-doctoral research in component and web-based collaborative technologies. He has held positions at Sun Microsystems, AudeSi Technologies, and Wind River Systems, and currently participates in the Servlet and JavaServer Faces expert groups.

Download at WoweBook.Com

Download at WoweBook.Com

Table of Contents

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Preface

ICEfaces is the technology leader in the integration of AJAX with the JEE stack. Its vendor, ICEsoft, offers a wide support for application servers, portal servers, and important open source frameworks. So, you do not have to bother yourself with the integration aspects of your project. You can focus on the implementation of business logic and its presentation in the web browser instead.

If you plan a rock-solid Web 2.0 implementation based on the JSF standard, ICEfaces is an ideal solution. Important enhancements of the JSF 2.0 standard are already available in ICEfaces 1.8. So, you can expect minimal porting efforts.

ICEfaces technologies such as skinning, multimedia, and AJAX Push already deliver the standards of tomorrow and ease the development of collaborative web applications. ICEfaces can even be mixed with modern RIA concepts based on Adobe Flex. A community of almost 90,000 developers (at the end of 2009) proves the quality and potential of the ICEfaces implementation.

This book is an introduction to the ICEfaces framework for enterprise developers with JSF experience. It describes how you can use ICEfaces components to build Web 2.0 applications with a desktop-like character.

The book examples will explain to you how ICEfaces helps integrating AJAX into a JEE stack (using AJAX Push, JSF, Facelets, Spring, and JPA/Hibernate) without using a line of JavaScript code. The book has a special focus on the Facelets technology that is now a part of the JSF 2.0 specification. This will help you to write your own components for a better reuse and maintenance.

By the end of the book, you will have a solid understanding of how to build modern and ergonomic web interfaces that fully integrate with the modern Java Enterprise stacks. You will be able to design and implement reusable and maintainable presentation components that allow creating customizable, skinnable, and multilingual web applications.

Download at WoweBook.Com

What this book covers

This book focuses on the implementation of desktop-like web applications that generate a high user acceptance. Each chapter describes a different aspect of how ICEfaces can be used to achieve this.

Chapter 1 gives a short introduction of modern JEE web development. It shows why we use AJAX and JSF today, and why ICEfaces is an ideal framework to use.

Chapter 2 helps you to set up tools and frameworks that are used to create and execute the sample code. We will have a look at a Windows XP environment using the Sun Microsystems JDK, the Eclipse IDE, the Maven 2 Build System, the Jetty web container, and the MySQL Database system. There is a special focus on the installation and the use of the ICEfaces plugin for Eclipse.

Chapter 3 takes a look at what a desktop-like presentation means to modern web applications. We will have a look at the design principles and start with a common page layout based on the Facelets templating.

Chapter 4 presents the ICEfaces components that help us to implement an intuitive navigation. The layout ideas from the previous chapter are further developed using the ICEfaces components.

Chapter 5 shows how your web application provides feedback to the user. We will have a deeper look at the special Facelets components that deliver similar functionality to that which a desktop developer would use in his/her applications. Additionally, we will have a look at the ICEfaces components that deliver a desktop-like behavior so that the user gets fast results with minimal effort.

Chapter 6 discusses data presentation components. We will focus on dynamic data tables that are fed by the database, and offer sortable and resizable columns. The second part of the chapter describes how to integrate Google Maps, videos, and Flash animations into your web application.

Chapter 7 describes the partial submit concept that ICEfaces offers to update forms on the fly. We also take a look at dialog-based validation and how advanced form elements, such as calendars and rich text editors, can be used.

Chapter 8 offers a model on how to implement the idea of user settings with the help of ICEfaces. We will take a deeper look at the language and skin management.

Chapter 9 takes a deeper look at the implementation details of Facelets components that were used in the previous chapters. We will discuss some fundamental design principles that help to create reusable and maintainable components without ever writing JSF custom components from scratch.

Download at WoweBook.Com

Chapter 10 discusses the principle of AJAX-based push technology and shows how easily it can be used with ICEfaces. For your amusement, we will implement the multiuser ICEmapper game, which uses Google Maps for the presentation.

Who this book is for

If you are an enterprise developer who wants to add the latest Web 2.0 features to a JSF project, this book is for you. You need a basic knowledge of the Spring framework configuration through annotations and the usage of the JPA annotations.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `<ui:insert>` tag allows us to set defaults that are used if we do not define something for replacement."

A block of code is set as follows:

```
<ui:component>
  <c:if test="#{context.dynamicMenu}">
    <icefusion:dynamicMenu/>
  </c:if>
  <c:if test="#{!context.dynamicMenu}">
    <icefusion:menu/>
  </c:if>
  <icefusion:menuIcons />
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<f:view locale="#{context.locale}">
<f:loadBundle basename="icefusion.icefusion" var=
  "icefusion"/>
<f:loadBundle basename="icecube.icecube" var="icecube"/>
<head>
```

Any command-line input or output is written as follows:

```
#java -jar start.jar
```

Download at WoweBook.Com

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The server works fine if you see the Jetty logo and a note, **Welcome to Jetty 6**."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

> Downloading the example code for the book
>
> Visit `http://www.packtpub.com/files/code/7245_Code.zip` to directly download the example code.
>
> The downloadable files contain instructions on how to use them.

Download at WoweBook.Com

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Download at WoweBook.Com

Download at WoweBook.Com

1
Next Generation Java Development

Why do we use **JavaServer Faces** (**JSF**)? Why do we add Asynchronous JavaScript and XML (AJAX) technologies to it these days? This chapter will have a quick look at the past, present, and future of modern Enterprise Java to answer these questions.

Enterprise Java has radically changed from its beginning in the late 1990s up to now. One of the new trends is the growing use of open source frameworks in the central parts of modern software architectures.

ICEfaces is one of those open source frameworks that speeds up the change in frontend development. This chapter will also discuss some of the ICEfaces features that show how the speed up is achieved.

Past problems

Enterprise Java was invented at Sun Microsystems in the 1990s. Everything started with a standardized and effective architecture to establish web container development. The first part of the **Java 2 Enterprise Edition** (**J2EE**) was born, and it brought us **Servlets** and **JavaServer Pages** (**JSP**).

The reference implementation for this, namely Tomcat, became the de facto standard from the early beginning. Even a decade after this, Tomcat is still one of the most used web containers. Surprisingly, it was even chosen by the Spring framework team to be the deployment platform of choice.

Download at WoweBook.Com

The success of the web container concept led the Sun engineers to think about a wider concept: a standardized infrastructure that allowed secure and transactional communication and data exchange. This should even work in heterogeneous environments with different operation systems, data representation standards, and network infrastructures.

There were already solutions, such as the **Common Object Request Broker Architecture** (**CORBA**) that was the most successful in those days. However, none of them had a native support for the upcoming Web, or had delivered a dedicated development model based on a pure object-oriented language.

So, the second part of J2EE became the **Enterprise Java Beans** (**EJB**) container. It allowed us to use an application server that had to implement certain services and **Application Programming Interfaces** (**APIs**) to conform with the J2EE specification. Application developers should be freed from thinking about infrastructure and allowed to concentrate on business logic. For this, the EJB component model defined two bean types:

- **Session beans**, which are used to implement the business workflow of the application
- **Entity beans**, which are defined to handle data persistence

In theory, this is a pretty nice concept; but it failed.

Over the years, J2EE became the de facto standard for distributed application development. However, some of the implementation details for such application servers produced time-consuming development processes. The J2EE developers had to develop and use code generators in cases such as overcoming the required code verbosity, for example.

Rod Johnson, the father of the Spring framework, was one of the first who deeply analyzed and criticized J2EE. The most important design problem of J2EE was the lack of object-oriented concepts, such as inheritance. So, **Plain Old Java Objects** (**POJO**) could not be used for implementation and this made the concrete design very complex.

New development concepts, such as unit testing and continuous integration, that came with the agile software development did not really work with the old J2EE ideas. So, the Java community developed new ideas to establish a lightweight Java development model. The business objects that were based on EJB technology should become less dependent on their containers, follow the POJO model, and deliver feasible reusability.

Download at WoweBook.Com

State of the art

For delivering feasible reusability, the **Dependency Injection (DI)** was established. DI allows one to write POJOs that do not know in which environment they live during runtime. Instead of establishing relations to other objects themselves, they get those relations injected. For this, attributes are defined, which follow the Java Bean model: You define getter and setter methods for attributes, following a certain naming convention. The container uses the setter to write instances of dependent objects into attributes with the same class type. If a POJO needs to communicate with a dependent object, it uses its own getter method to get the reference.

This lightweight development model is a central part of the Spring framework that is on the way to replace the old J2EE architecture. The current Spring release is able to replace almost 90% of the old EJB container functionality. Best of all, it can be used with a standard web container such as the good old Tomcat. It is also possible to use Spring in standalone applications; for example, applications based on the **Rich Client Platform** (**RCP**) ideas of the Eclipse team.

The return to pure web container environments for the deployment allows us to create and manage simpler infrastructures. This additionally helps to reduce costs and allows a faster time to market.

Frontend development

In the beginning of J2EE, the JSP development model was invented to ease the design of web pages. It allowed the separation of the presentation from the application logic. They had to be mixed only when using servlets.

Struts

The previous development model missed a useful concept of reusability and maintenance. So, the Apache Group offered Struts, a **Model View Controller** (**MVC**) framework, which soon became the de facto standard for frontend development. It is still in use in a lot of projects that started in the early J2EE years.

Download at WoweBook.Com

JSF

The **Java Community Process** (**JCP**) recognized the demand for standardization and developed JSF some years later.

The JSF concept uses the existing servlet and JSP implementations, and adds a modern component and event model to it. JSF allows us to follow the classical request-response development model that we know from JSP. For the existing JSP implementations, a migration to JSF can be done step-by-step.

JSF also allows us to follow the ideas of desktop application development. This kind of implementation becomes increasingly important because of the demand for desktop-like presentations in web applications.

AJAX

AJAX technology is the key for implementing desktop application behavior in web applications today. It allows the changing of the web browser from a simple "show an information page" to a "let me use a web-based application" user model.

A clever combination of dynamic HTML with a JavaScript-based backend communication (for example, using XML messages) allows us to skip the reloading of pages. This helps to mimic the behavior of desktop applications in the browser.

As AJAX is established through widely accepted frameworks such as Dojo, it is possible to use such a behavior in other web development environments such as JSF. However, the integration quality of both the worlds is very important for an application's maintenance. Tests of different JSF frameworks in the past have shown that there are a lot of tweaks. ICEsoft, the company behind the ICEfaces framework, is a technology leader in such integration aspects.

The use of AJAX technology is not without problems. Although it mimics the behavior of desktop applications, the result is still HTML and **Cascading Style Sheets** (**CSS**), plus a lot of JavaScript. This can soon be a maintenance nightmare because JavaScript in the AJAX context is fragile and is often used in places it was not designed for. For this, a JSF framework should have renderers that shield a developer from JavaScript coding manually. ICEfaces is such a framework. However, even ICEfaces (with a dedicated renderer for each supported browser) is not able to create the same presentation or behavior in any case.

Another limitation of AJAX is the presentation of complex data. HTML has images, but offers no interactivity or immediate feedback for in-place manipulation. You have to send an HTML form to change parameters on the server side for regeneration before the image changes in the client side.

Download at WoweBook.Com

This is contradictory to the behavior of typical desktop applications. However, the demand for such functionality is rising in web browser environments.

Rich Internet Applications (RIA)

There are new technologies on the horizon that try to solve the challenge of complex data presentation. Flex, based on the Flash platform, was the first RIA solution. In comparison to other RIA technologies, it is the most mature one.

The Flash player, which is used as the Flex runtime, was invented a decade ago and is the most widespread web browser plugin ever. It has a rock-solid architecture and is used in mission-critical environments. Flex adds a development model to it that allows you to develop applications using the **Eclipse IDE** (**Flex Builder**). So, you are no longer bound to the Adobe Flash tools.

> Adobe invests a lot of money to integrate Flex with modern Java technologies, such as the Spring framework, using BlazeDS (`http://blog.rainer.eschen.name/2008/07/02/flex-supports-spring-are-you-ready-to-skip-web-20/`) or Spring-Flex (`http://www.springsource.org/spring-flex`). This helps us to use Flex for more complex data presentations inside of the JSF environments. Both solutions can already be used in the ICEfaces projects. Spring-Flex is more modern and recommended to use.

Additionally, there is the JSF-Flex project (`http://jsf-flex.googlecode.com/`) that allows a tight integration of the Flex components in JSF environments. It implements special JSF tags to use the MXML components' Flex delivers. So, the JSF development model can be kept when you use Flex for complex data presentations. This is a very promising idea from the maintenance point of view. However, the project uses Dojo and we have to mix different AJAX frameworks if we integrate it with ICEfaces. A mixture of AJAX frameworks is always problematic.

JSF-Flex was published at the time of writing this book. So, it is not clear if it can be integrated with ICEfaces. It is also not clear whether a Spring-Flex integration with JSF-Flex will be possible. A fully integrated software stack with these frameworks would be a powerful tool to implement very complex data presentations based on the ICEfaces, Flex, and Spring technologies.

The first design studies with pure Flex clients based on JEE (Java Enterprise Edition) backends exist. So, the change from the mixed JSF/Flex presentations to a pure Flex presentation is foreseeable. The possibility to port such an implementation without any effort to the desktop, using the **Adobe Integrated Runtime** (**AIR**), will speed up this trend.

Download at WoweBook.Com

For both Flex and AIR applications, we need a Flash runtime that has to be installed first. This can be a challenge if administrators on the customer side do not manage Flash applications and also block those installations for their users. However, we can expect this problem to disappear with the increasing number of Flex and AIR applications that will be used in production over the next years.

ICEfaces features

If you are thinking about using JSF as the new frontend technology for your project, it is a good time to start right now with a first prototype. The technology is mature enough to be used in mission-critical environments. The upcoming JSF 2.0 specification, JSR 314 (`http://jcp.org/en/jsr/detail?id=314`), improves the JSF model so that faster development and easier integration with other important frontend technologies will be possible. We can expect all the teething troubles from the 1.x specification to be resolved.

The current ICEfaces release already delivers a lot of the expected changes from JSF 2.0. So, even if you go productive with your project in the near future, the porting efforts of your ICEfaces project to JSF 2.0 will be minimal. It is possible to use the current JSF 2.0 RI Beta instead of the 1.x RI implementations with ICEfaces. There is also the ICEfaces *glimmer* code tree for JSF 2.0 native support that you can play with (`http://blog.icefaces.org/blojsom/blog/default/2009/07/01/On-the-road-to-ICEfaces-2-0/`).

JSF reference implementation support

If we have a look at today's JSF implementations, we can see that we have two framework layers:

- The JSF **Reference Implementation (RI)**
- The vendor-specific add-ons

The most well-known reference implementations are:

- Sun Microsystems RI (`https://javaserverfaces.dev.java.net`)
- Apache MyFaces (`http://myfaces.apache.org`)

All of the important JSF vendors support both. So, your choice can be made dependent on the deployment environment. If an application server already delivers the necessary JSF libraries, like Glassfish V2 does with the Sun RI, you may stay with these. Often, you get extras such as stability, tighter integration, or better performance with such a delivery.

Download at WoweBook.Com

The ICEfaces framework is an example for the vendor-specific add-on layer. ICEsoft defines it as an *Ajax extension for JavaServer Faces.*

The next sections will show why ICEfaces is an excellent example for developer productivity in JSF environments with an AJAX support.

Interoperability

The ICEfaces framework follows a standard, namely JSF, that allows us to use the framework in common JEE environments without extra efforts. For best interoperability and no vendor lock-in, the ICEsoft engineers offer excellent support to integrate the framework with:

- Other important open source frameworks
- All important IDEs
- All important application servers
- All important portal servers

Framework integration challenges

Today, you have choice. There are several JSF-based open source projects you can choose from. It is no problem to run pure open source development projects these days. However, you pay a price for missing integration efforts and pure documentation in the open source landscape. One of the worst experiences of the past few years was to integrate such open source frameworks without support by the framework developers.

The problem

Although, you have full access to the code, you will not have much time to analyze it. All of the important frameworks are complex due to the number of classes they deliver. The time pressure in your project, and also a natural reservation against analyzing black boxes keep you from a deep understanding. Mostly, the documentation is mean. Only a few open source projects deliver a documented architecture. Javadocs, if written, are seldom precise enough to compensate for the lack of architectural information. So, the analysis has to become a guess.

Handmade integration workarounds by other framework users often cover only a certain focus. So, even if you get workarounds in the source code, you have to transfer this solution to your problem; and such integration efforts in projects can still lack stability.

Download at WoweBook.Com

To search for the cause of an integration problem is a hopeless venture in these cases. The most annoying thing with this is the fact that you cannot be sure if a certain problem is because of your development style or the deficient integration.

Before our team decided to choose ICEfaces in August 2007, we did a deep analysis of several competing open source products. All lacked the required stable integration with Facelets, the JSF de facto standard for templating. This was the most important requirement to us because of the flexibility we could have using the Facelets component design ideas.

At that time, ICEfaces was the only framework with a fully integrated Facelets distribution that was updated with every release. Our prototypes showed that, otherwise, we had to manage Facelets-specific source trees for our own patches. With every new release of a JSF framework, or an update of the Facelets framework, there was a likelihood of having to rework the patches. This kind of a maintenance hell was not an option for us.

The ICEfaces way

The ICEsoft engineers recognized earlier than others that the adoption of a framework depends on its integration quality. Hence, the ICEfaces framework can be integrated with other JSF frameworks by default. However, there is one limitation: This only works if you do not have to mix different AJAX technologies.

By contrast, non-JSF frameworks need more integration efforts. Hence, the number of supported frameworks is growing slowly.

Today, ICEsoft supports these open source frameworks:

- JavaServer Faces 1.x (`http://java.sun.com/javaee/javaserverfaces`)
- Facelets (`https://facelets.dev.java.net`)
- MyFaces Tomahawk (`http://myfaces.apache.org/tomahawk/index.html`)
- JBoss Seam (`http://www.jboss.com/products/seam`)
- Liferay Portal (`http://www.liferay.com`)
- Spring Web Flow (`http://www.springsource.org/webflow`)

Download at WoweBook.Com

IDE plugins

JSF development is comparable to desktop development—both have visual components and event handling. It would be nice to have an IDE that allows the user to work like a desktop developer, but that also has all the bells and whistles that modern web development delivers.

Although the offered functionality is not comparable to what you may know from Delphi or Visual Basic, ICEsoft offers IDE plugins that help to get something similar. So, you get (for example) code completion, syntax highlighting, or the possibility to visually design web pages.

There are some limitations with the "What You See Is What You Get" presentation, but the integration is mature enough to deliver a real benefit. As ICEsoft supports all of the important IDEs, you can stay with your preferred IDE. The ICEfaces download page (`http://www.icefaces.org/main/downloads/os-downloads.iface`) delivers plugins for the following IDEs. (You need an account for downloading.)

- Eclipse (`http://www.eclipse.org`)
- myEclipse (`http://www.myeclipseide.com`)
- Rational Application Developer (`http://www.ibm.com/software/awdtools/developer/application`)
- Netbeans (`http://www.netbeans.org`)

Application server support

ICEsoft also supports all of the important application servers, such as:

- Apache Tomcat (`http://tomcat.apache.org`)
- BEA Weblogic Server (`http://www.oracle.com/appserver/index.html`)
- JBoss Application Server (`http://www.jboss.org/jbossas`)
- IBM Websphere Application Server (`http://www.ibm.com/software/websphere`)
- Oracle Application Server Container for J2EE [OC4J] (`http://www.oracle.com/technology/tech/java/oc4j/index.html`)
- SAP NetWeaver (`http://www.sap.com/platform/netweaver`)
- Sun GlassFish / Sun Java System Application Server (`https://glassfish.dev.java.net`)
- Webtide Jetty (`http://www.mortbay.org/jetty`)

Download at WoweBook.Com

ICEsoft offers extended configuration descriptions (`http://support.icesoft.com/jive/category.jspa?categoryID=80`) and deployment examples that help to get your project running. For the samples, have a look at `/icefaces/samples` in the source distribution at `http://www.icefaces.org/main/downloads/os-downloads.iface`. You need an account to download.

Portal server support

Portlet development, which is standardized in JSR 168 (`http://jcp.org/en/jsr/detail?id=168`) in combination with the Liferay framework (`http://www.liferay.com/`), is supported for the following portal servers:

- BEA WebLogic Portal (`http://www.oracle.com/products/middleware/user-interaction/weblogic-portal.html`)
- JBoss Portal (`http://www.jboss.org/jbossportal`)
- Apache Pluto (`http://portals.apache.org/pluto`)
- JetSpeed 2 (`http://portals.apache.org/jetspeed-2`)

Similar to the application server deployment examples, you get portlet sample applications to start with.

Have a look at `/icefaces/samples/portlet` in the source distribution at `http://www.icefaces.org/main/downloads/os-downloads.iface`. You need an account to download.

If your project is using portal technology and you want to implement your portlets based on JSF, the ICEfaces framework is an excellent candidate to do so.

Components for ergonomic interface design

Although the ergonomics of user interfaces is defined by the way you choose page layout and navigation, the design quality of the visual components that you can use for it is not without influence. The ICEfaces framework offers everything that an ergonomic user interface design needs for today's web applications.

Download at WoweBook.Com

In highly competitive situations where web applications have to persuade customers within seconds, the functionality is not deciding in the first step. If your web application lacks ergonomics, you will lose a prospect. It seems that the ICEsoft engineers considered this during the components' design. They found an excellent combination of AJAX and JSF, and even a development model, that will cause minimal efforts for the developer.

Customer-specific skins

Individual solutions become more and more important to customers. Customization is a central part of modern web architectures. This is especially true for the visualization.

Modern implementations allow users to adapt the presentation to customer-specific colors, fonts, images, and so on. Even the layout has to be changeable in some cases. For this, we need components that separate presentation from business logic.

Web applications that support a change of the visualization during runtime are called *skinnable*. The ICEfaces framework delivers standard skin definitions to start with. However, you can also define your own skins through the change of component-specific CSS classes.

Server-initiated client updates

Push technology gets a revival through AJAX. We are able to update certain values inside of a web page when the server side changes. This can be done without initiation through the client.

ICEfaces has a seamless integration of this idea using AJAX in a JSF context. ICEfaces delivers the most advanced implementation to get the best of both worlds without the necessity to program a single line of JavaScript.

The push can be done to a group of clients simultaneously. This helps to establish collaborative computing.

Optimized page updates

Web pages are fully loaded only once. After this, the update is limited to certain parts of a page. Updates can be initiated by the server, and are done asynchronously.

Download at WoweBook.Com

The development model behind this is server-side AJAX. Although the client (namely, the web browser) uses AJAX, the developer does not have to program the client side. Instead, he uses the JSF notation on the server side. The rest is done by the framework. All of the context information, which is necessary to decide when and which parts of a page should be updated, is calculated automatically without any extra programming by the developer.

This is even true for all dependencies between update areas of a page. So, you do not have to define where to update, when to update, or which dependencies an update has to consider. You only define events in your JSF code without even using special tags. The context management for dependent visualization is done by the framework and is transparent to the developer. So, we can concentrate on the business logic.

ICEsoft calls this technology **Direct-to-DOM**. Direct-to-DOM manages a server-side **Document Object Model (DOM)**, which is similar to the web browser's DOM, to present a web page. All the changes that are necessary inside a page are done in the server-side DOM first, and then the AJAX update mechanism updates the client-side DOM. The client and server are kept in sync through the AJAX bridge.

The event control is kept on the server side. You do not have to care for browser releases or vendor-specific limitations in JavaScript or CSS implementations. ICEfaces delivers browser-specific renderers that circumvent all of the JavaScript problems. The results are code stability and also better security.

Community

No successful framework is without a vital community. The ICEfaces community is growing continuously. This is certainly an indication of the quality of the framework.

> You can get into contact with other ICEfaces developers at `http://www.icefaces.org/JForum/forums/list.page`. If you have any questions, the ICEfaces forum will help you out. Some of the ICEsoft engineers read and write in this forum on a regular basis. You need an account to use the forum.

The combination of JSF and AJAX has its own problems. Although the architecture behind ICEfaces is remarkable, the implementation has its challenges. The Issue Tracker at `http://jira.icefaces.org/` can help to check if your development style is the problem, or if it is the current ICEfaces release. Often, the ICEfaces forum can offer a workaround in the meantime. You need an account to use the Issue Tracker.

Download at WoweBook.Com

For more official communication, the ICEsoft team uses a company blog (also known as the ICEfaces Water Cooler). For latest news or programming tips and tricks, have a look at `http://blog.icefaces.org`.

Summary

This chapter had a quick look at the history of Java Enterprise, with a special focus on frontend development.

The Java community is changing from the heavyweight, EJB-based development model to a lightweight, POJO-like one. The comeback of pure web container deployments and the universal use of Dependency Injection allow a faster time to market.

ICEfaces, the JEE presentation framework based on the JSF standard, perfectly integrates into the new development model. It adds the AJAX Push and Direct-to-DOM technologies that allow you to develop desktop-like web applications without using special tags or JavaScript development.

Such web applications can be integrated with popular frontend and backend frameworks. You do not have to change your development environment to start with ICEfaces. The ICEfaces applications can be deployed to all of the established application servers and portal servers.

The next chapter will describe the installation and configuration of the Eclipse IDE for ICEfaces development. Additionally, we will have a first look at a modern software stack that uses ICEfaces as its presentation framework. The obligatory *Hello World!* example will also be discussed in the next chapter.

Download at WoweBook.Com

Download at WoweBook.Com

2 Development Environment

This chapter will take a look at the tools and frameworks that we will use in the chapters to come. We start with the setup of JDK, Eclipse IDE, Maven 2 Build System, Jetty web container, and MySQL Database system on Windows XP. With the help of the ICEfaces plugin for Eclipse, we will create our first web applications. Finally, we will have a look at the advanced JEE development stack based on ICEfusion (AppFuse), which will be used in the upcoming chapters.

Tools

This section will show you how to install the following on a Windows XP system:

- The JDK
- The Eclipse IDE
- The Maven 2 build system
- The Jetty web container
- The MySQL database system

If you are already familiar with all this, you may proceed with the *Additional Eclipse Configurations* section.

Java Development Kit (JDK)

Although the Windows update installer may have installed a Java runtime on your computer, we need a full-blown JDK installation. Eclipse, and also the Maven 2 build system, expects this. We will use JDK 1.6.0_15 for our examples.

Download at WoweBook.Com

Installation

The JDK can be downloaded from `http://java.sun.com/products/archive/j2se/6u15/index.html`. Click on the **Download JDK** link. The next page asks for a platform. Choose **Windows**. Do not forget to select the legal checkbox before you click on the **Continue** button. Click on the link **jdk-6u15-windows-i586-p.exe**, or select the checkbox and use the **Download Selected with Sun Download Manager** button to download the file.

After downloading the JDK, start the installation by double-clicking on the `jdk-6u15-windows-i586-p.exe` file. Follow the dialog instructions and use the suggested paths. After the installation, we have to set the environment variable `JAVA_HOME` and extend the `PATH` variable. For this, go to **Start | Control Panel | Performance and Maintenance | System | Advanced** and click on the **Environment Variables** button.

In the top list, click on the **New** button and create a `JAVA_HOME` variable with the value `C:\Program files\Java\jdk1.6.0_15`. In the bottom list, select the **Path** variable entry and click on the **Edit** button. Add `;C:\Program files\Java\jdk1.6.0_15\bin` at the end of the **Value**. Do not forget the semicolon that is used as a separator. Close all of the control panel dialogs that are open so that Windows activates your changes. You also have to close all of the opened command windows because such windows still use the old settings that were valid before you made your changes.

For testing the new environment, we will open a new command window and type:

```
javac -version
```

You should get the answer:

```
javac 1.6.0_15
```

You may have to wait for some time before the answer is created.

If you get the message that the `javac` command cannot be found, check your environment variable `PATH`:

```
set PATH
```

You should see the path `C:\Program files\Java\jdk1.6.0_15\bin` at the end of the `PATH` result.

For further installation details, have a look at `http://java.sun.com/javase/6/webnotes/install/jdk/install-windows.html`.

Download at WoweBook.Com

Eclipse IDE

Eclipse is today's most well-known and also the most important **Integrated Development Environment** (**IDE**). It delivers the base functionality for most of the commercial IDE products that you buy today.

The Eclipse ecosystem consists of a lot of subprojects, which deliver extensions (plugins) for the Eclipse kernel. Meanwhile, the number of plugins has reached a level that does not allow a beginner to start without a deeper study of them.

The plugin concept supports reusability. So, plugin A can use plugin B to deliver certain functionality. The Eclipse installer can solve such dependencies and install dependent plugins, if necessary. However, it cannot manage higher-level dependencies when you have to decide how to combine a number of plugins for a certain category, such as database development.

Customized distributions

For the management of higher-level dependencies between Eclipse plugins, there exist download services that offer online tools to create customized Eclipse distributions. You choose a certain Eclipse-based package and add plugins from several categories to it. Finally, the online system generates a single file that you can download and use for installation on your local machine. Your selection can be saved in a profile. You can edit the profile or share it with others. With the profile, you can generate a new installation file at any time.

The advantages of such a download service are:

- The plugins are tested with the current Eclipse base so that the customized distribution is stable in any case.
- You do not have to monitor plugin dependencies. If a plugin is useful for another one inside a category, you get a hint. With your confirmation, all necessaries are added automatically.
- You do not have to follow the market for every new plugin that may help your project. You can browse the categories instead. Most plugins have a detailed description, or even show extra hints from the service vendor.

Download at WoweBook.Com

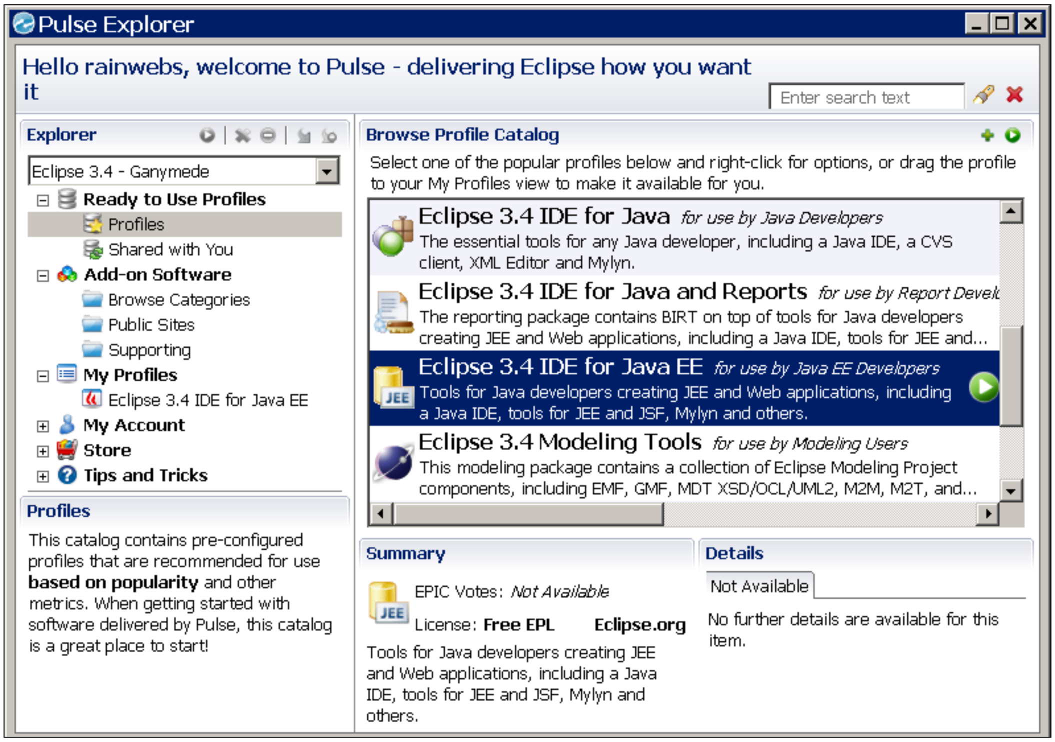
Through a shift in the target group marketing, some vendors now offer their service free of charge. Such free offers are useful for a single developer, or for a group of developers in small projects. If you plan for a more standardized IDE rollout management in bigger development teams, or if you need special service support, have a look at the payable offers instead.

Pulse download service

There are two products that are widely used today:

- Pulse (`http://www.poweredbypulse.com/`)
- Yoxos (`http://ondemand.yoxos.com/`)

If you are new to Eclipse, you may have a look at the free part of Pulse first. It offers Eclipse-based packages just as the download page of the Eclipse home page does. However, you manage these in a desktop application, the Pulse Explorer, which you have to download first. The category management is more clearly arranged, but is not as comprehensive as the Yoxos one.



Pulse Explorer

Hello rainwebs, welcome to Pulse - delivering Eclipse how you want it

Enter search text

Explorer

Eclipse 3.4 - Ganymede

- **Ready to Use Profiles**
  - Profiles
  - Shared with You
- **Add-on Software**
  - Browse Categories
  - Public Sites
  - Supporting
- **My Profiles**
  - Eclipse 3.4 IDE for Java EE
- **My Account**
- **Store**
- **Tips and Tricks**

Profiles

This catalog contains pre-configured profiles that are recommended for use **based on popularity** and other metrics. When getting started with software delivered by Pulse, this catalog is a great place to start!

Browse Profile Catalog

Select one of the popular profiles below and right-click for options, or drag the profile to your My Profiles view to make it available for you.

Eclipse 3.4 IDE for Java *for use by Java Developers*
The essential tools for any Java developer, including a Java IDE, a CVS client, XML Editor and Mylyn.

Eclipse 3.4 IDE for Java and Reports *for use by Report Devel*
The reporting package contains BIRT on top of tools for Java developers creating JEE and Web applications, including a Java IDE, tools for JEE and...

Eclipse 3.4 IDE for Java EE *for use by Java EE Developers*
Tools for Java developers creating JEE and Web applications, including a Java IDE, tools for JEE and JSF, Mylyn and others.

Eclipse 3.4 Modeling Tools *for use by Modeling Users*
This modeling package contains a collection of Eclipse Modeling Project components, including EMF, GMF, MDT XSD/OCL/UML2, M2M, M2T, and...

Summary

EPIC Votes: *Not Available*

License: **Free EPL** **Eclipse.org**

Tools for Java developers creating JEE and Web applications, including a Java IDE, tools for JEE and JSF, Mylyn and others.

Details

Not Available

No further details are available for this item.

Download at WoweBook.Com

Yoxos download service

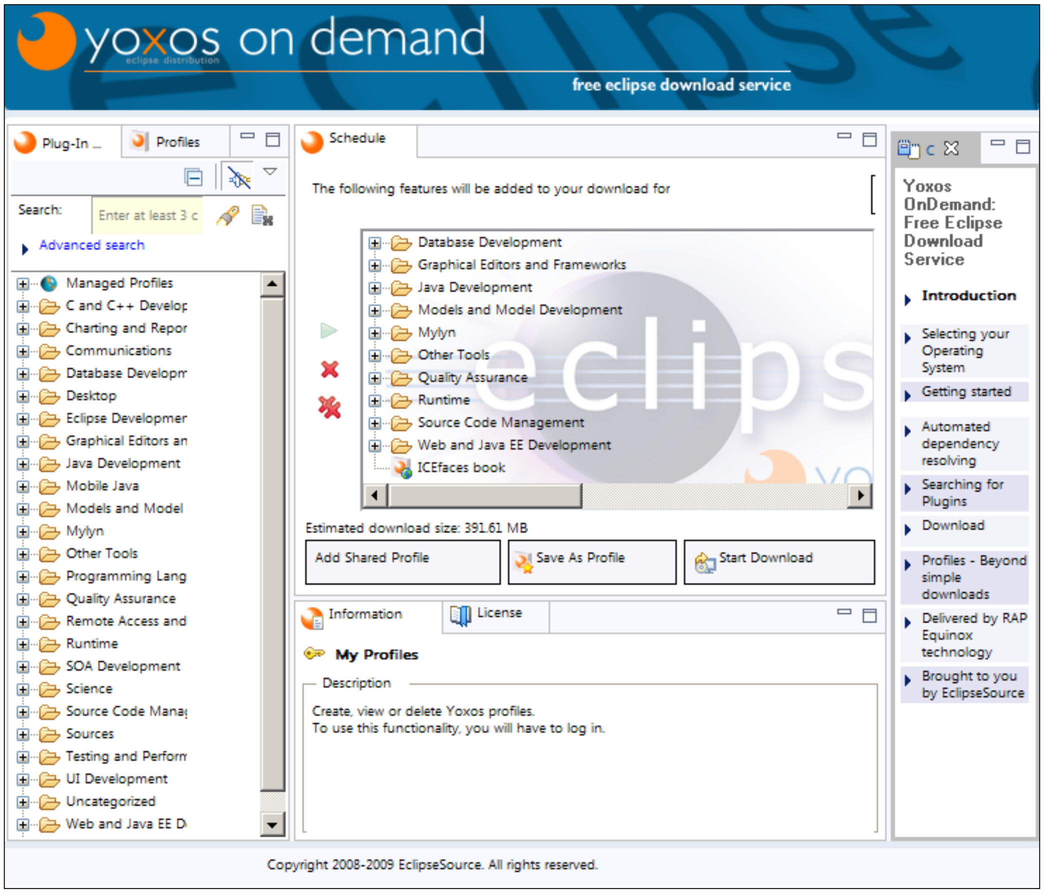
Yoxos offers an online catalog manager, which is supplemented with an Eclipse perspective that has nearly the same presentation. This kind of management seems to be a bit more handy. The comprehensive catalog seldom lets you think about adding plugins by hand. However, the ICEfaces plugin is not a part of the catalog and we actually have to do this later.

The ICEfaces book profile at Yoxos

Yoxos allows you to manage profiles and offers them to the public. To download the ICEfaces book profile use the following URL in your web browser:

```
http://ondemand.yoxos.com/geteclipse/rap?profil
es=868129468_1232759792533368664
```

In the Schedule tab, all dependent plugins are shown:



The following features will be added to your download for

Estimated download size: 391.61 MB

Create, view or delete Yoxos profiles.
To use this functionality, you will have to log in.

Copyright 2008-2009 EclipseSource. All rights reserved.

Download at WoweBook.Com

Instead of typing the key yourself, you can also choose the Yoxos ICEfaces book profile link at `http://blog.rainer.eschen.name/icefaces/icefaces-book-chapter-2/`.

When you click on **Start Download**, dialog box pops up with the download link on top. You can ignore the additionally shown feedback form.

Installation

You can take the ZIP file and expand it to your `Program Files` folder. It will create an Elipse folder with every plugin in it that the profile defines. You can start Eclipse by double-clicking on `eclipse.exe`, which can be found in the root of the folder structure. You may create a desktop link to it for later use.

Customization of the ICEfaces book profile

If you have used this Yoxos distribution for a while, you may miss a plugin or rethink the use of some of the existing ones. You can change the ICEfaces book profile for this.

To add plugins, use the **Plug-In Explorer** tab on the left for the selection, and the green arrow to add a plugin to the **Schedule** tab. Choose plugins you want to delete inside the **Schedule** tab and click on the red cross on the left of the list. To save the result, click on **Save As Profile**. You may have to create an account for this.

> If you use the online tool to customize the profile, do not forget to save all your changes before the pages are changed. If you forget to do this, all your changes will be lost. The tool has no memory for it.

Maven 2 build system

The Maven build system is the successor of Ant (`http://ant.apache.org/`), the first build system that is based on a pure Java implementation. Maven 1 implements new ideas that came up during the development of Java products at the Apache Software Foundation some years ago. Maven 2 is the result of a redesign based on the experiences of Maven 1 through the Java community. Maven 1 and Maven 2 are incompatible. To avoid confusion, we add the release number to the product name here.

Download at WoweBook.Com

The automatic management of transient dependencies is what makes Maven superior to Ant. Your build does not have to define all the dependencies that the project JARs have. Instead, you define the first dependency layer—the JARs you use directly. All the following layers, which describe the dependencies that the JARs from the first layer have, are managed by the build system. So, if you change a JAR release, all the necessary dependency changes are done by the build system.

This is done through a cascade of the **Project Object Model** (**POM**) files that describe such dependencies for a Maven artifact. A Maven artifact can be a JAR file, for example. However, you can also define the build of a WAR artifact. With its plugin concept, Maven allows you to do a lot of different things, such as the creation of POJOs from a database or the creation of Javadocs.

Installation

Maven can be downloaded at `http://maven.apache.org/download.html`. Take the 2.2.1 release ZIP archive and extract it to your `Program Files` folder. It will create the `apache-maven-2.2.1` folder. Inside, you will find the `bin` folder that delivers the start scripts. Open a command shell with this folder chosen and type the following command to test your Maven installation:

```
mvn -version
```

Normally, it shows the version number.

For your convenience, it is a good idea to add the Maven `bin` folder to your environment path. So, you can use the `mvn` command independently in every folder where a `pom.xml` can be found.

By default, there is no repository folder generated when you execute Maven the first time. However, the Eclipse Maven Plugin expects one.

To create such a repository folder, also type this command:

```
mvn clean
```

You will get a build error message because Maven can not find a POM file. But, it downloads the Maven Clean Plugin and with it, it generates a valid repository folder structure.

Jetty web container

For web development, we need an additional web container, also known as a servlet container. The Jetty web container is similar to Tomcat, which we discussed in the last chapter. A key feature of Jetty is its ability to run in an embedded mode, which allows you to integrate it in other environments.

Download at WoweBook.Com

Use in Maven 2 Environments

The Maven 2 build system is such an environment. Jetty can be used to deploy a Maven build without a dedicated installation. This is very useful if you want to have a quick look at a source distribution. Build the distribution with the following command:

```
mvn install
```

Now, you can deploy and test it with the following command:

```
mvn jetty:run
```

For more details, have a look at `http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin`.

Installation

Jetty can be found at `http://dist.codehaus.org/jetty/`. We use the 6.1.9 release. If you click on the folder link, you can find the `jetty-6.1.9.zip` file. Download the file and extract the archive to a folder which path has no spaces in it. Jetty faces problems with handling spaces in its installation path. So you can not use the `jetty-6.1.9` folder is created. Open a command window and change to the newly created folder. Type this to start the server:

```
java -jar start.jar
```

If you can see the **Started SelectChannelConnector@0.0.0.0:8080** log entry, the server is ready to use. Open your web browser and use `http://localhost:8080/test/` to open a page. The server works fine if you see the Jetty logo and a note, **Welcome to Jetty 6**.

MySQL Database Management System

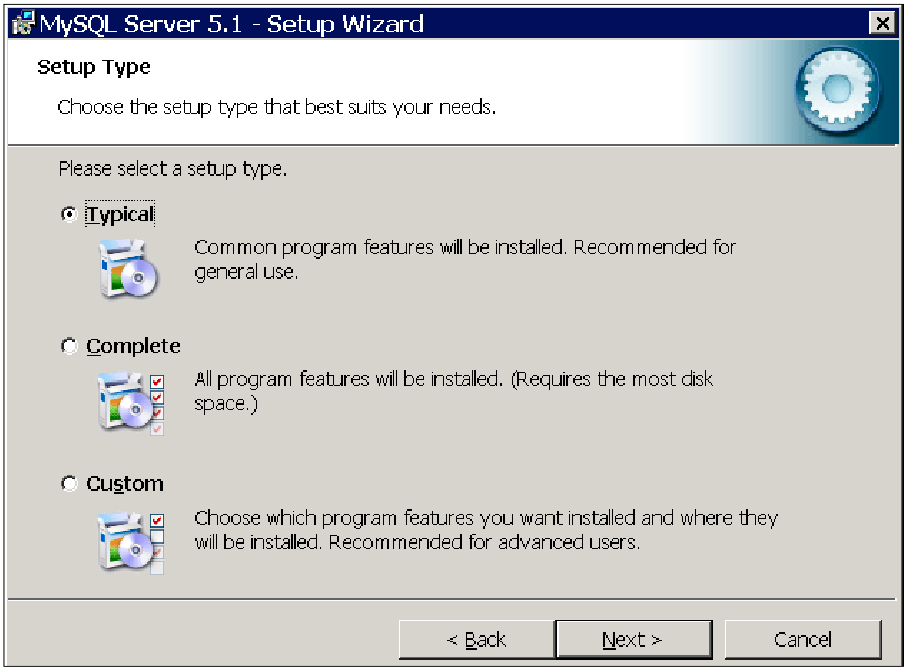
MySQL is the most used open source **Relational Database Management System** (**RDBMS**) on the planet. It is the standard database in the AppFuse project. So, for our ICEfusion implementation, we will use it too.

Installation of Community Server

Have a look at `http://dev.mysql.com/downloads/mysql/5.1.html#win32` for the download. We use the Community Server 5.1 edition. Take the **Windows MSI Installer (x86)** by clicking on the **Download** link on the right. The **mysql-5.1.39-win32.msi** installer can be started with a double-click.

For the **Setup Type**, choose **Typical** because we will not do anything special with the database.

Download at WoweBook.Com



MySQL Server 5.1 - Setup Wizard

Setup Type

Choose the setup type that best suits your needs.

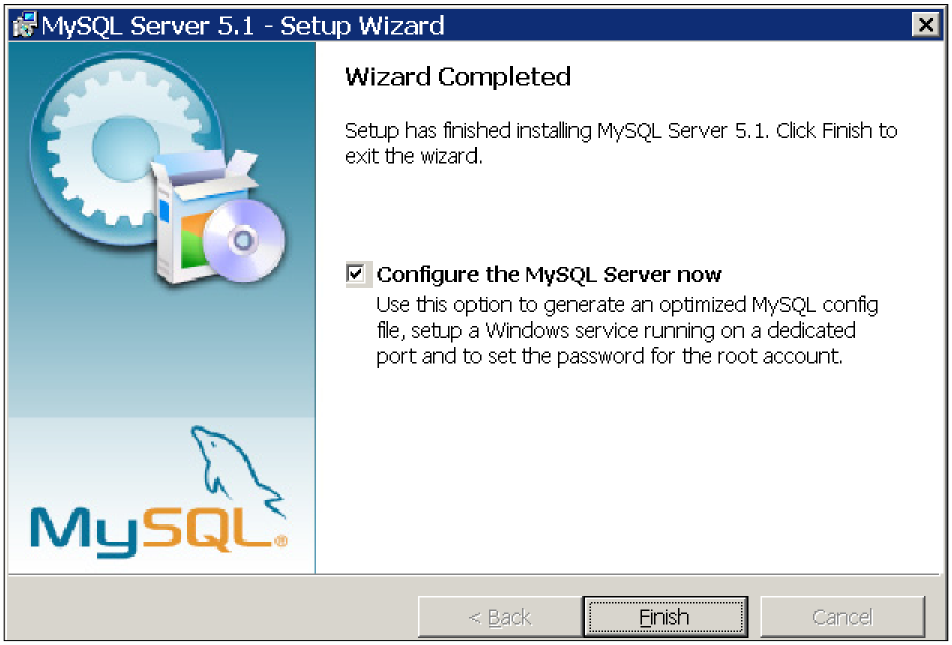
Please select a setup type.

(●) **Typical**
Common program features will be installed. Recommended for general use.

( ) **Complete**
All program features will be installed. (Requires the most disk space.)

( ) **Custom**
Choose which program features you want installed and where they will be installed. Recommended for advanced users.

At the end of the installation, keep the **Configure the MySQL Server now** checkbox selected. The database system has to be initialized first, as shown in the next screenshot:



MySQL Server 5.1 - Setup Wizard

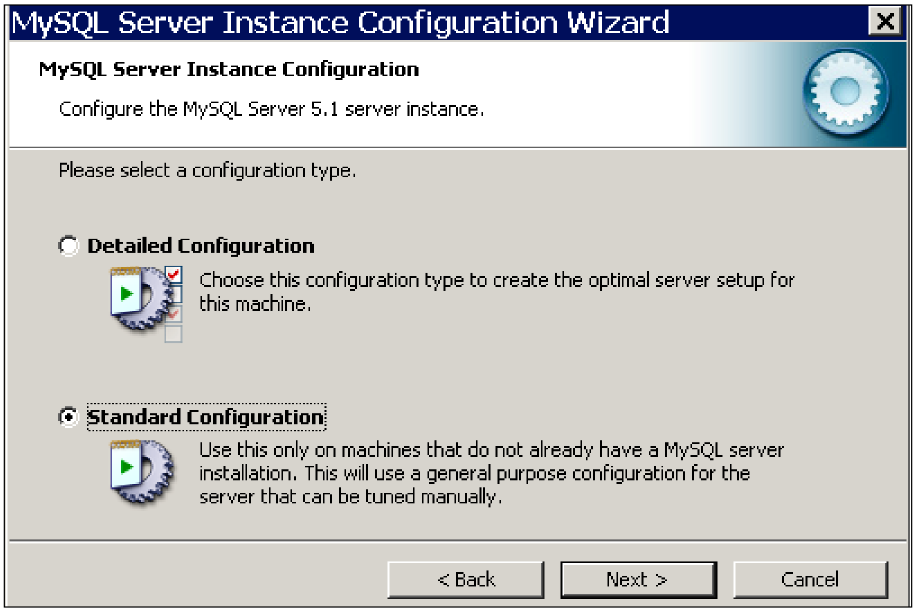
Wizard Completed

Setup has finished installing MySQL Server 5.1. Click Finish to exit the wizard.

[✓] **Configure the MySQL Server now**
Use this option to generate an optimized MySQL config file, setup a Windows service running on a dedicated port and to set the password for the root account.

Download at WoweBook.Com

For the **Configuration Type**, use **Standard Configuration**:



MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 5.1 server instance.

Please select a configuration type.

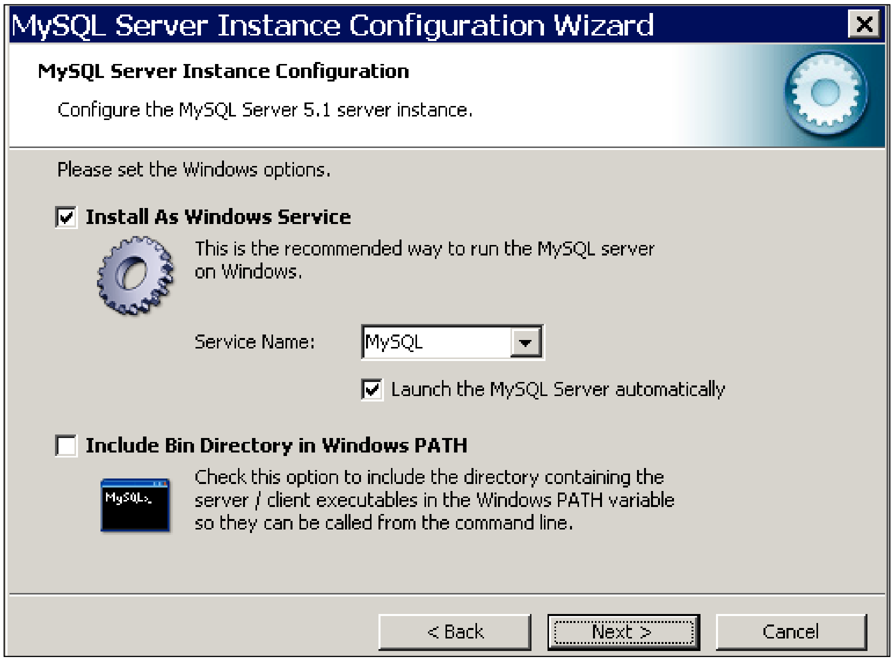
○ **Detailed Configuration**

Choose this configuration type to create the optimal server setup for this machine.

● **Standard Configuration**

Use this only on machines that do not already have a MySQL server installation. This will use a general purpose configuration for the server that can be tuned manually.

In the **Windows Configuration**, keep everything as is. Even if you do not like to install MySQL as a Windows Service, this is the simplest way to install it:



MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 5.1 server instance.

Please set the Windows options.

☑ **Install As Windows Service**

This is the recommended way to run the MySQL server on Windows.

Service Name: MySQL

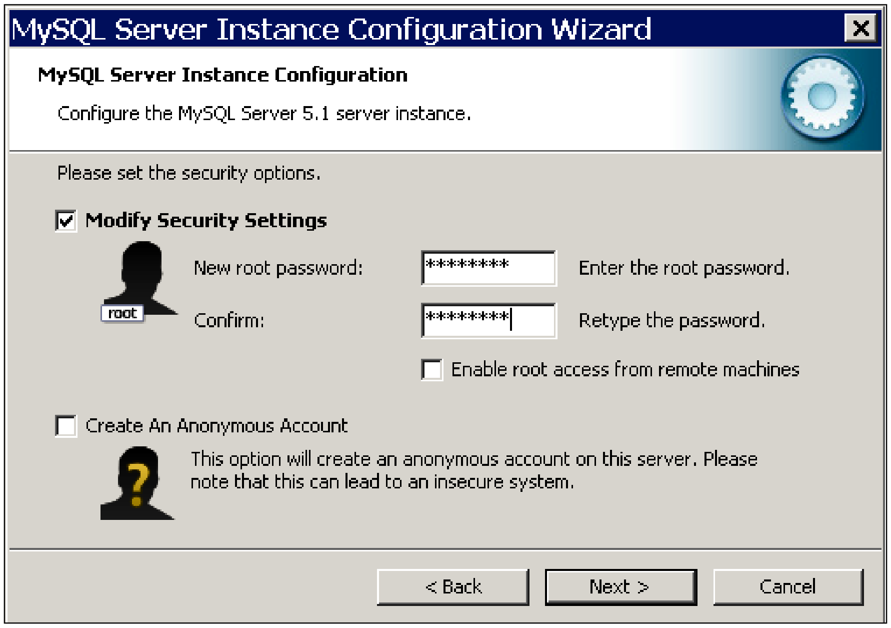
☑ Launch the MySQL Server automatically

☐ **Include Bin Directory in Windows PATH**

Check this option to include the directory containing the server / client executables in the Windows PATH variable so they can be called from the command line.

Download at WoweBook.Com

In the **Security Options**, keep all settings selected and set the password as `icefaces`:


MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 5.1 server instance.

Please set the security options.

[✓] Modify Security Settings

New root password: ********* Enter the root password.

Confirm: ********* Retype the password.

[ ] Enable root access from remote machines

[ ] Create An Anonymous Account

This option will create an anonymous account on this server. Please note that this can lead to an insecure system.

After the configuration is finished, we can test the installation. For this, we execute **MySQL Command Line Client** in the Windows start menu at **Programs** | **MySQL** | **MySQL Server 5.1**. First, enter the password `icefaces`. Now have a look into the RDBMS to test it. We ask for the number of users in the system:

```
select count(*) users;
```

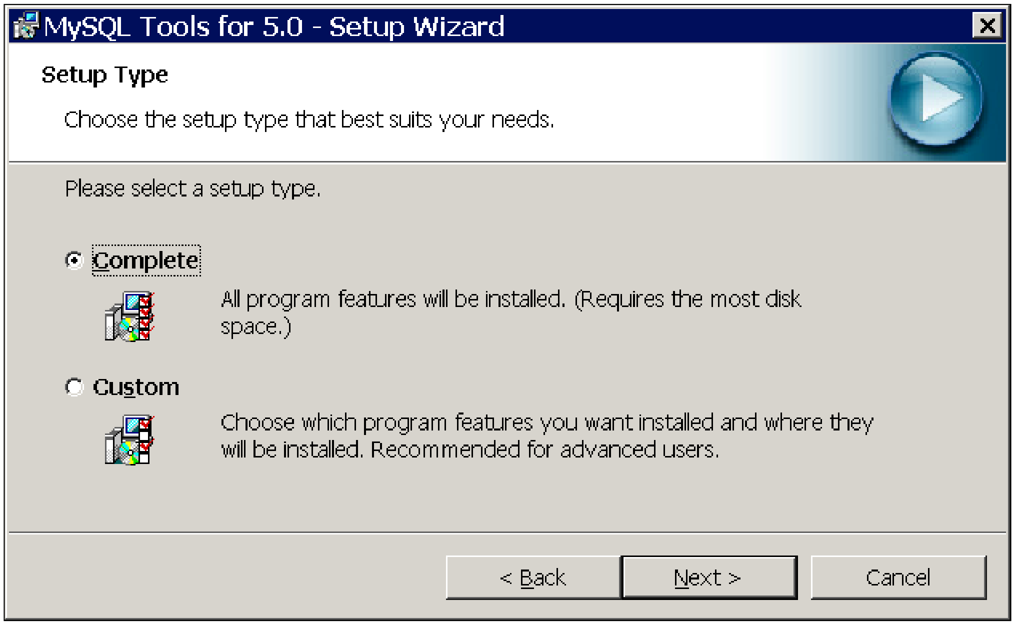
The result looks similar to this:

```
mysql> select count(*) users;
+--------+
| users  |
+--------+
|      1 |
+--------+
1 row in set (0.03 sec)
```

Download at WoweBook.Com

Installation of GUI Tools

Additionally, there is a GUI Tools Bundle 5.0 that you can find at `http://dev.mysql.com/get/Downloads/MySQLGUITools/mysql-gui-tools-5.0-r16-win32.msi/from/pick`. Ignore the form and have a look at the end of the page to click on the link **No thanks, just take me to the downloads!**. Start the `mysql-gui-tools-5.0-r16-win32.msi` installer through double-click. We use **Complete** in **Setup Type**, as shown in the next image:


MySQL Tools for 5.0 - Setup Wizard
Setup Type
Choose the setup type that best suits your needs.
Please select a setup type.
Complete
All program features will be installed. (Requires the most disk space.)
Custom
Choose which program features you want installed and where they will be installed. Recommended for advanced users.
< Back
Next >
Cancel

We do not necessarily need the MySQL GUI Tools. However, it is more comfortable to have a look into the database structures with these tools than using the Database Developer Tools of Eclipse.

Additional Eclipse configurations

Before we can create our *Hello world!* examples, we have to add some more functionality to the Yoxos distribution by hand. This section assumes that you are familiar with Eclipse configuration basics. Although the following descriptions are detailed, not all steps are explicitly described. For a general introduction into Eclipse, study the *Workbench User Guide* at `http://help.eclipse.org`.

Download at WoweBook.Com

ICEfaces plugin

Open source framework developers, who also deliver extensions for the common IDEs, deserve a closer attention. And ICEsoft is one of those teams. Although the corresponding Eclipse plugin has its limitations in the "What You See Is What You Get" presentation of JSF and Facelets markup, it has its strengths in other cases. So, using it is basically a good idea.

It is important to have a look at the ICEfaces release that is supported. Each ICEfaces release follows a corresponding Eclipse plugin some time later. For new projects, like our book examples, this is not important because the ICEfaces JARs can be installed with the plugin too. However, there can be differences with the existing projects if you use Maven to build the project and Eclipse only as a smart editor.

With the installation of the plugin, you get:

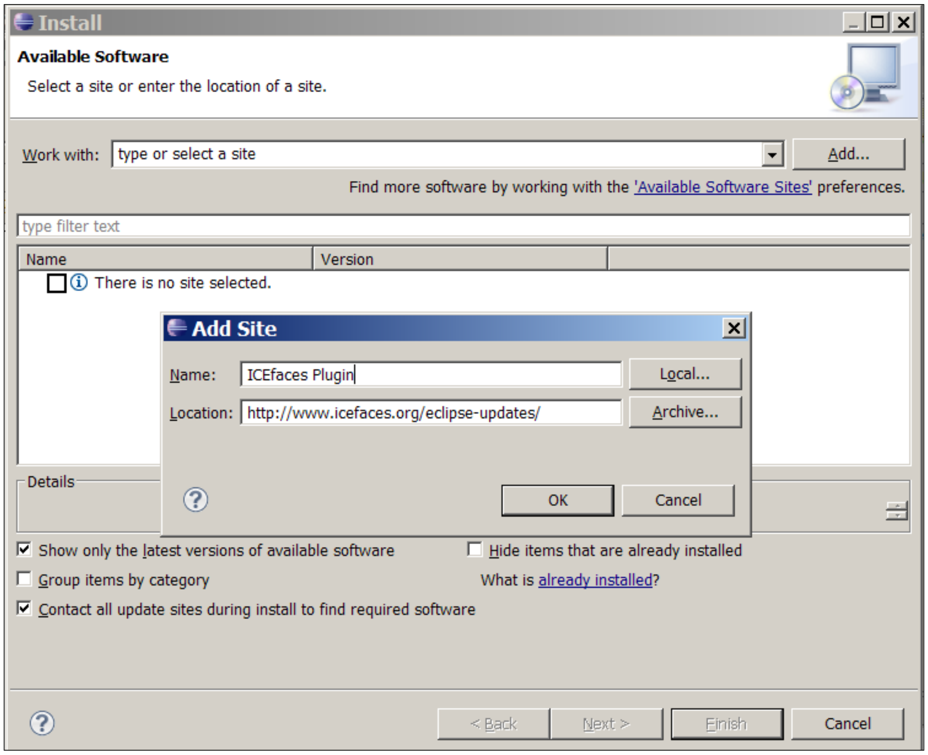
- ICEfaces and Facelets support for dynamic web projects
- Easy creation of the Eclipse projects with the ICEfaces and Facelets support
- Full integration for deployment on Eclipse-managed web containers
- Syntax highlighting, code completion, and visual tag management in the Web Page Editor

Installation

ICEsoft offers an update site. So, the download and installation by hand, that was necessary before release 3.6.2, is no longer necessary for the plugin.

Download at WoweBook.Com

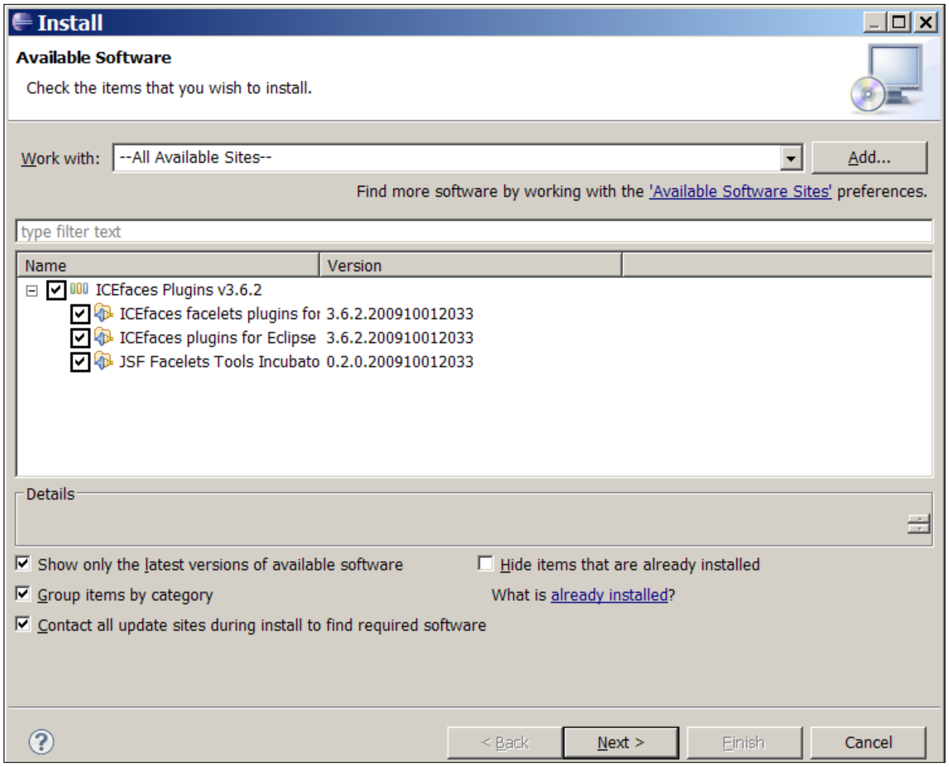
Start Eclipse and go to **Help** | **Install New Software...** | **Add**.


Install
Available Software
Select a site or enter the location of a site.
Work with: type or select a site
Add...
Find more software by working with the 'Available Software Sites' preferences.
type filter text
Name
Version
There is no site selected.
Add Site
Name: ICEfaces Plugin
Location: http://www.icefaces.org/eclipse-updates/
Local...
Archive...
OK
Cancel
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Contact all update sites during install to find required software
< Back
Next >
Finish
Cancel

The **Location** is set with the update site `http://www.icefaces.org/eclipse-updates`. Set a **Name** such as `ICEfaces Plugin` and click on the **OK** button.

In the next step, you can choose the items you want to install. You may have to change **Work with;** to **All Available Sites**. In the list, we check ICEfaces Plugins v3.6.2:

Download at WoweBook.Com



Available Software

Check the items that you wish to install.

Work with: --All Available Sites--

Find more software by working with the 'Available Software Sites' preferences.

type filter text

Name	Version
☑ ICEfaces Plugins v3.6.2	
☑ ICEfaces facelets plugins for	3.6.2.200910012033
☑ ICEfaces plugins for Eclipse	3.6.2.200910012033
☑ JSF Facelets Tools Incubato	0.2.0.200910012033

Details

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

A click on the **Next** button starts the download. You may get a Security Warning telling you that you are trying to install unsigned content. You can ignore this because the update site is managed by ICEsoft.

After the installation, Eclipse has to be restarted.

Maven 2 and the JDK

The Maven 2 support for Eclipse is already a part of the Yoxos distribution. However, the standard installation of Eclipse only references the JRE. Maven 2 needs the JDK instead.

Have a look at **Window** | **Preferences** | **Java** | **Installed JREs**. The list shows the referenced JRE. Click on the **Add** button and choose **Standard VM** from the list. In the next dialog box, click on **Directory** and choose the installation folder of the JDK.

The list of **Installed JREs** shows the JDK reference. Select the checkbox to make it the default one. You may also delete the old JRE entry.

Download at WoweBook.Com

To complete the JDK support for Maven, we explicitly have to set the JDK as virtual machine for Eclipse. Normally, we would add a command to the `eclipse.ini` that can be found in the Eclipse installation folder. But, this does not work. Instead, we add a command line parameter to the icon that is used to start Eclipse:

```
"C:\Program Files\eclipse\eclipse.exe" -vm "C:\Program files\Java\
jdk1.6.0_15\bin\javaw.exe"
```

If you want Eclipse to load and work faster you may also add these parameters:

```
-vmargs -Xms512m -Xmx512M -XX:PermSize=256m -XX:MaxPermSize=256m
-XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled
-XX:+CMSPermGenSweepingEnabled
```

Jetty server support

Eclipse allows us to manage our standalone Jetty inside the IDE. This allows us to deploy Eclipse web projects to it. We can also debug these with the standalone Jetty.

First, you have to open the **Servers View** with **Windows | Show View | Other**. Select **Server | Servers** from the tree to open the view. Next, we create a new entry for Jetty. For this, open the context menu inside the server view and choose **New | Server**.

Jetty is not a part of the standard distribution support. So, you have to click on the **Download additional server adapters** link. The dialog box shows a list of those adapters. Choose **Jetty Generic Server Adaptor** and accept the license agreement. After downloading, a restart of Eclipse is necessary.

At the time of writing this book, there was a repository problem with Eclipse 3.5 – the adaptor dependencies could not be resolved. If you still get errors during your installation, try to install the necessary plugin manually. Have a look at **Help | Install New Software... | Add** and use `http://www.webtide.com/eclipse` for **Location**.

We open the context menu inside the server view and choose **New | Server** again. The list now shows the Jetty entry. Click on it and choose **Jetty 6**. You may give the server a different name. The next dialog page lets you choose the JRE and the Jetty installation folder.

For **JRE**, use the JDK reference we installed for the Maven 2 plugin. For **Jetty Home**, use the folder you installed the standalone Jetty in. The next dialog page can be kept and is for information purposes. If you try to start a deployment in the web browser, use `localhost:8080`. In Internet Explorer or the Eclipse web browser, this will be `http://localhost:8080`.

Download at WoweBook.Com

We can skip the next dialog page because we have no Eclipse project at the moment that can be added to the Jetty server. So, click on the **Finish** button. The server view now shows the active Jetty entry.

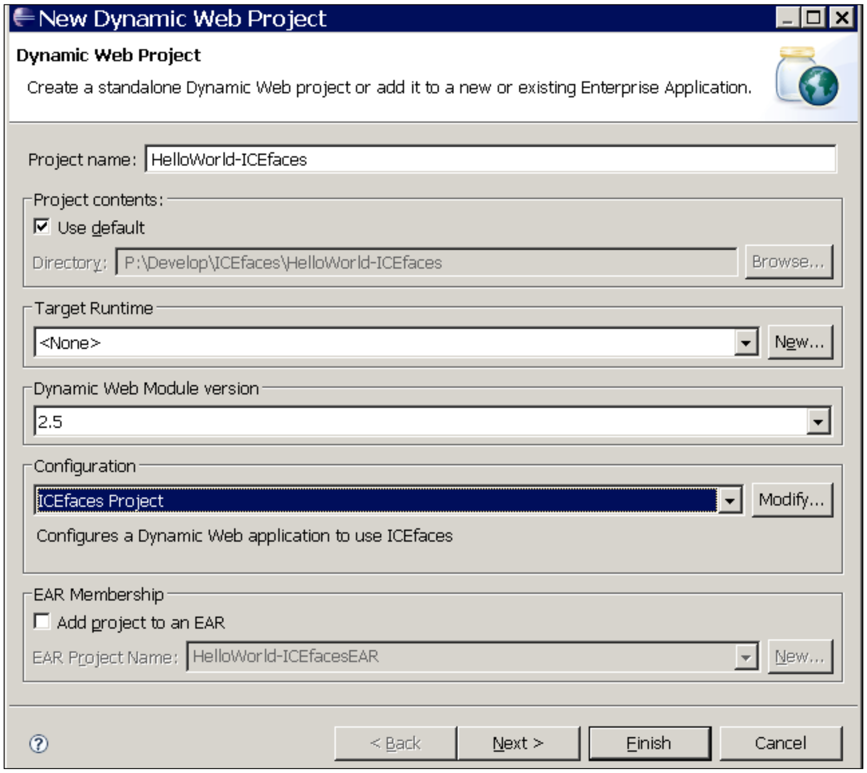
For further details, have a look at `http://docs.codehaus.org/display/JETTY/Web+Tooling+Support`.

The Eclipse web project samples

It is time for the mandatory *Hello World!* implementation. We will have two of them. The first one is based on the standard ICEfaces implementation. The second one adds Facelets to it. However, we will not really implement templating. Facelets will be discussed in detail in the next chapter.

ICEfaces

ICEfaces projects are specialized Eclipse dynamic web projects. So, we start with the creation of a dynamic web project and configure it for the ICEfaces use. Have a look at **File | New | Project... | Web | Dynamic Web Project** in the following screenshot:


New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: HelloWorld-ICEfaces

Project contents:

[✓] Use default

Directory: P:\Develop\ICEfaces\HelloWorld-ICEfaces

Target Runtime: <None>

Dynamic Web Module version: 2.5

Configuration: ICEfaces Project

Configures a Dynamic Web application to use ICEfaces

EAR Membership

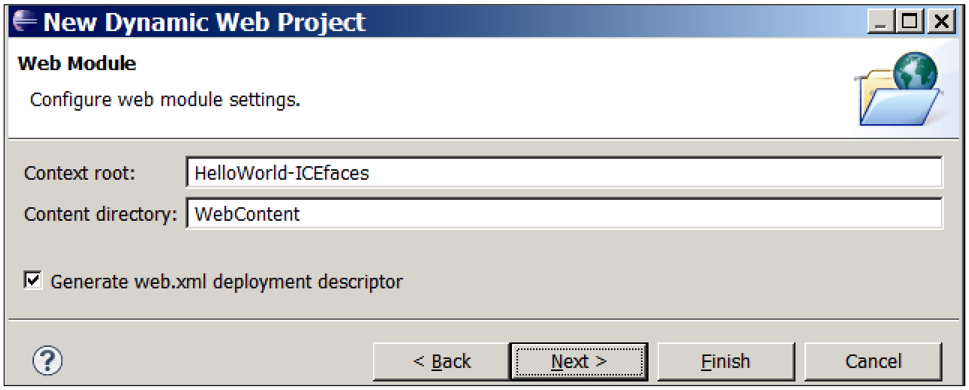
[ ] Add project to an EAR

EAR Project Name: HelloWorld-ICEfacesEAR

Download at WoweBook.Com

The important things to note in this dialog are **Dynamic Web Module version**, which has to be the latest release (**2.5** in our case), and **Configuration**. You can choose from different combinations using ICEfaces and Facelets. For our first example, we need the `ICEfaces Project` entry. Click on the **Next** button to continue.

All entries in the next dialog page, **Java**, can be kept as is and you can click on the **Next** button immediately. The third dialog page, **Web Module**, has to be set here:



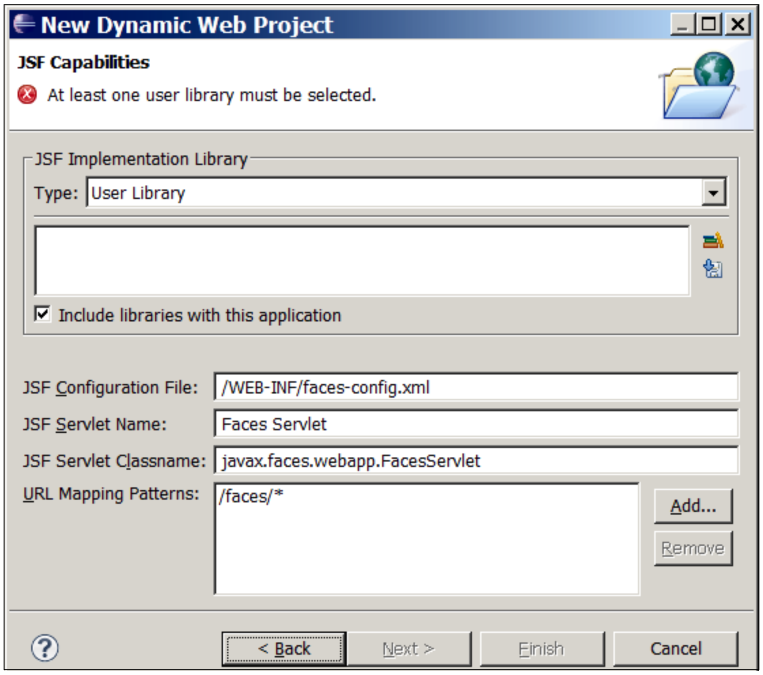
Configure web module settings.

Context root: HelloWorld-ICEfaces

Content directory: WebContent

[✓] Generate web.xml deployment descriptor

Within this dialog, it is important that the **Generate web.xml deployment descriptor** checkbox is selected. Click on the **Next** button for the library settings as follows:



At least one user library must be selected.

Type: User Library

[✓] Include libraries with this application

JSF Configuration File: /WEB-INF/faces-config.xml

JSF Servlet Name: Faces Servlet

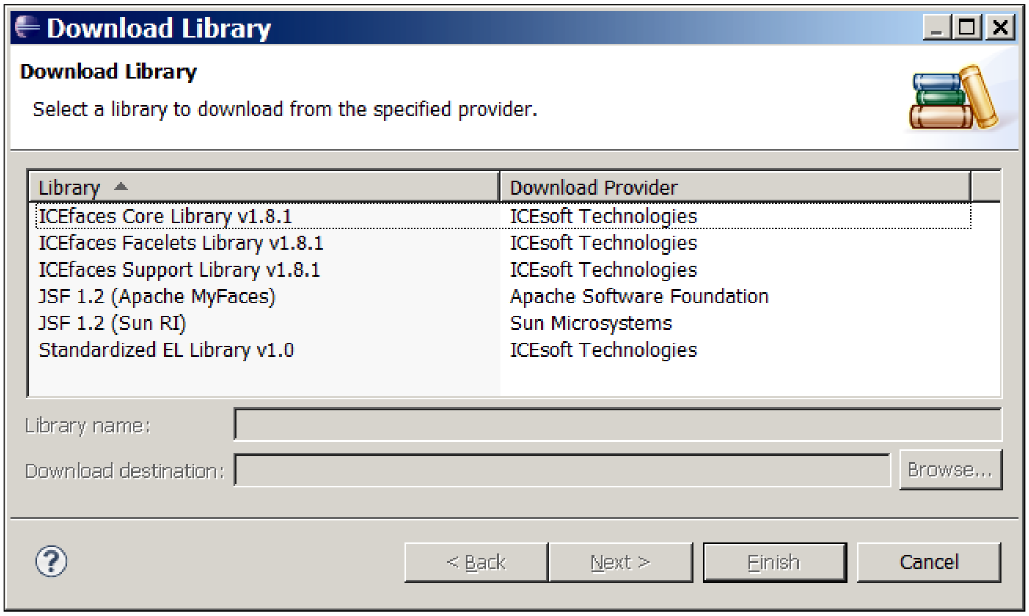
JSF Servlet Classname: javax.faces.webapp.FacesServlet

URL Mapping Patterns: /faces/*

Download at WoweBook.Com

We have to select and download the necessary libraries because these are not a part of the ICEfaces Eclipse plugin distribution anymore. Before you add them, have a look at the **Include libraries with this application** checkbox. This should be selected.

Click on the **Download library** button (disk icon on the right) for the libraries selection.



Download Library

Download Library

Select a library to download from the specified provider.

Library	Download Provider
ICEfaces Core Library v1.8.1	ICEsoft Technologies
ICEfaces Facelets Library v1.8.1	ICEsoft Technologies
ICEfaces Support Library v1.8.1	ICEsoft Technologies
JSF 1.2 (Apache MyFaces)	Apache Software Foundation
JSF 1.2 (Sun RI)	Sun Microsystems
Standardized EL Library v1.0	ICEsoft Technologies

Library name:

Download destination:

Our development is based on Apache MyFaces. For this, we select:

- ICEfaces Core Library v1.8.2
- JSF 1.2 (Apache MyFaces)

Each entry has to be downloaded separately. For each click on the **Next** button, select the **I accept the terms of this license** checkbox and click on the **Finish** button. After this, the download will start. The dialog will close automatically. Now, you can continue with the rest of the libraries list.

The last dialog page **ICEfaces configurations** can be kept as is, and left with an immediate click on the **Finish** button.

Download at WoweBook.Com

The Run on server configuration

The generated project is shown in the Project Explorer. If you open the context menu by clicking on the project name, you can open the settings dialog of **Run As | Run on Server**. Choose **Jetty 6** from the list and select **Always use this server when running this project**. Click on the **Finish** button.

If you have a look at the **Servers** view, you will notice that Jetty starts and tries to deploy the project. The **Console** view presents the log statements during the deployment. The web browser is started in parallel.

If the web browser shows an error, you have to wait until the deployment is done. After this, the browser reload should show a welcome message.

```
java.lang.IllegalStateException: No Factories configured for this
Application. This happens if the faces-initialization does not
work at all - make sure that you properly include all configuration
settings necessary for a basic faces application and that all the
necessary libs are included. Also check the logging output of your web
application and your container for any exceptions!
If you did that and find nothing, the mistake might be due to the
fact that you use some special web-containers which do not support
registering context-listeners via TLD files and a context listener is
not setup in your web.xml.
```

The `web.xml` file of the generated project is missing the following:

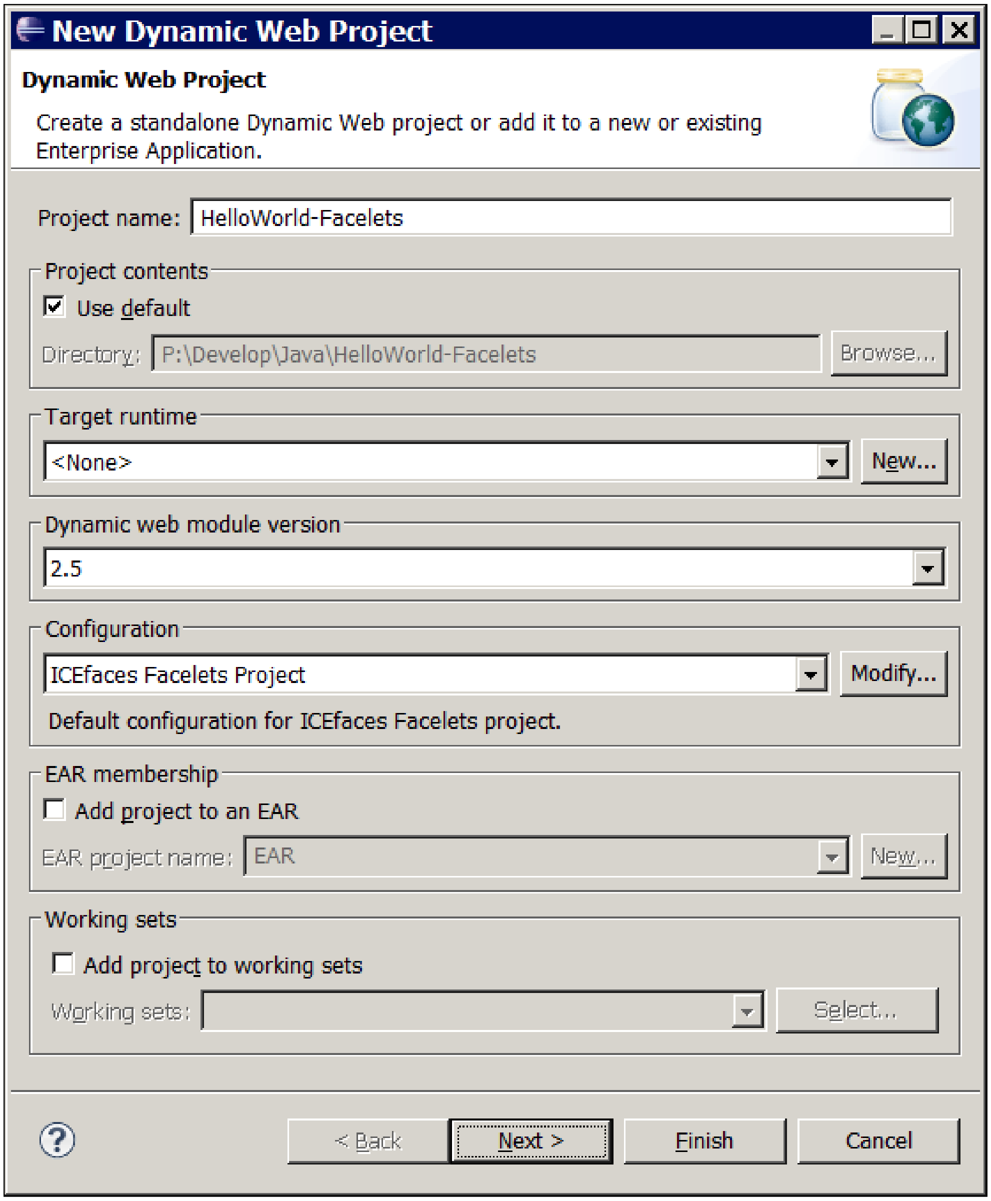
```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

If you add this and run the project again, the web browser will show **Welcome to ICEfaces**. You may have to reload the browser window in Eclipse. This is the standard output text of the `ICEfacesPage1.jspx` page. You can change the output to `Hello World!` to complete the project.

ICEfaces and Facelets

The creation of a `Hello World!` ICEfaces project using Facelets is similar to the creation of a `Hello World!` ICEfaces project without Facelets. So, we only describe the differences here. In the **Dynamic Web Project** dialog, you have to choose **ICEfaces Facelets Project** instead of `ICEfaces Project`:

Download at WoweBook.Com



New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: HelloWorld-Facelets

Project contents

[✓] Use default

Directory: P:\Develop\Java\HelloWorld-Facelets

Target runtime: <None>

Dynamic web module version: 2.5

Configuration: ICEfaces Facelets Project

Default configuration for ICEfaces Facelets project.

EAR membership

[ ] Add project to an EAR

EAR project name: EAR

Working sets

[ ] Add project to working sets

Working sets:

Download at WoweBook.Com

You have to download and select the following in the **JSF Capabilities** dialog and the **Download Library** dialog:

- ICEfaces Core Library v1.8.2
- ICEfaces Facelets Library v1.8.2
- JSF 1.2 (Apache MyFaces)

After the deployment and the browser reload, the page asks you to add some ICEfaces components.

The JEE development stack

You may wonder why we chose Maven 2, Jetty, or MySQL in the tools section. All three support an integration framework that lays the ground for our JEE application that we will start to develop with the next chapter.

For a realistic project scenario, the JEE application will integrate today's most important open source frameworks. These will deliver or support the next generation Java technologies such as:

- Dependency Injection
- Aspect-oriented programming
- Annotations
- Object-relational mapping and data access objects

These technologies will be used to implement non-functional requirements such as:

- Transaction management
- Persistence
- Security

Download at WoweBook.Com

AppFuse

To integrate all the JEE frameworks from scratch still requires a tremendous effort. However, we avoid this by using the integration framework AppFuse (`http://www.appfuse.org`).

AppFuse was started in 2002 by Matt Raible, primarily to support a book project. Now it is a production-ready kickstarter for modern JEE implementations. The framework supports the most important open source frameworks and even allows you to combine them individually. For more details about the framework support, have a look at `http://www.appfuse.org/display/APF/Reference+Guide`.

The edoras framework

During the writing of this book, an alternative to AppFuse became open source—the edoras framework (`http://www.edorasframework.org/`) by mimacom (`http://www.mimacom.ch/en/solutions/`), the company that is offering ICEfaces support in Europe. edoras has a strong focus on frameworks that are supported by ICEsoft; for example, Liferay. It also delivers Facelets components and other extensions to ICEfaces that ease the enterprise development.

ICEfusion

Although AppFuse has pretty nice features, it is missing a fundamental feature that we need—ICEfaces support. This is a feature that has been discussed by the AppFuse community since ICEfaces became open source. However, nobody has implemented it yet for the public.

As there was no real alternative to AppFuse before edoras became open source, I, with the support of Matt, developed a solution. The current implementation can be found at `http://icefusion.googlecode.com`.

ICEfusion is an adaptation of the basic JSF Maven Archetype that is delivered with AppFuse. ICEfusion keeps the existing backend technology that allows us to implement transaction management, persistence, and security. The frontend technology is almost fully replaced with a new ICEfaces and Facelets implementation.

All code changes follow the idea of keeping the original AppFuse structure, so that the AppFuse or ICEfaces updates can be integrated without effort. As the Maven build process delivers an extensive automation, a lot of code is actually generated during the build, and not all preconditions can be changed. So, some obsolete files are still a part of the project. You will even find configurations that circumvent the original ideas to get the new implementation behavior. However, that would never be implemented in this way in a project from scratch.

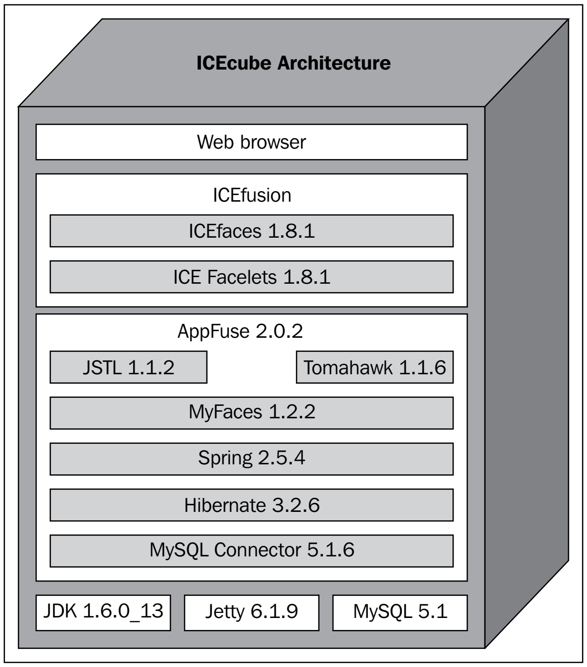
Download at WoweBook.Com

Nevertheless, for our book application, this is not dramatic. We will only have a look at certain packages or files that allow extending ICEfusion. So, you can ignore all this in the first step. However, if you think about using ICEfusion for your own project, you may need to have a look at these details.

ICEcube

Our example implementation is a full-blown web application. For this, it gets its own name: ICEcube. The implementation is based on the ICEfusion architecture and its Facelets components. ICEcube shows how to use and combine these components in certain presentation contexts.

The following image presents the architecture behind ICEcube:



The building blocks are:

- The Java platform
- The Jetty web container
- The MySQL database management system
- The AppFuse integration framework
- The ICEfusion ICEfaces adapter for AppFuse
- The web browser

Download at WoweBook.Com

The following sections will provide some more details about those parts of the architecture that were not discussed before.

The Spring framework

A central part of the architecture is the use of Dependency Injection. The Spring framework delivers everything necessary for this. This allows ICEcube to use Spring beans instead of JSF beans for implementing the backing bean concept.

Spring is used in all the layers. So, it is also used to establish transaction management and persistence using Hibernate for Object-Relational Mapping. Most of the configuration for this is done through annotations.

Spring Security is used to protect the web access. It is also used to manage the login, *Remember me*, and logout functionality.

Hibernate

Hibernate allows us to use POJOs for managing persistence. One of the biggest advantages of the AppFuse framework is the automated generation of all the necessary Hibernate artifacts during the build process.

So, the programming of POJOs is pretty simple. We define the POJO attributes and the getters and setters, and add some annotations. As long as we use simple CRUD (Create, Read, Update, Delete) processing, we do not have to invest more time for transaction management and persistence.

If you have to take action manually, you can use the Generic DAO (Data Access Object) concept. The creation of DAOs, which you use to manage the persistence of POJOs, is done through Generics. So, you can have the same DAO functionality with every POJO type and you do not have to implement such functionality yourself.

Annotations and Generics are the reason why we use JDK 6.

Apache Tomahawk

Normally, the ICEfaces framework delivers everything you need for presentation purposes. However, there are some JSF components that you may miss. Apache Tomahawk is a popular extension to the MyFaces implementation and can help here.

ICEsoft has spent some time integrating Tomahawk and ICEfaces, so that a mixture of JSF tags can be used. Not all Tomahawk tags can be used the same way that you know as a standalone Tomahawk implementation. Have a look at `http://support.icesoft.com/jive/servlet/KbServlet/download/731-102-1045/ICEfacesTomahawkCompSupport.html` for tips and possible limitations.

Download at WoweBook.Com

If you plan to use the ICEfaces skinning, it is not useful to mix ICEfaces tags with other JSF framework tags. The skinning expects some extras from the JSF components that other JSF frameworks do not support.

JSP Standard Tag Library (JSTL)

The **JSP Standard Tag Library** (**JSTL**) is not a part of the JSF 1.x specification. However, it is compatible with JSF and we can use it in the ICEfaces contexts too. You will primarily use it for control structures that are necessary if decisions have to be made outside of backing beans, so to speak, as part of the tag layer.

Summary

This chapter described which tools we will use in the next chapters. It showed how to install and configure them. For some, the interdependencies were also discussed.

You used Eclipse and ICEfaces plugins to create your first web application. It was possible to create a standard ICEfaces project and deploy this on an Eclipse-managed Jetty container. Additionally, we created a project with a Facelets support.

The tools selection was primarily done to support the development of our AppFuse/ICEfusion-based JEE application—ICEcube. Although AppFuse delivers everything for a real-world implementation, it lacks the ICEfaces support. I have started the ICEfusion project to overcome this limitation.

The ICEcube architecture integrates all of the important open source frameworks for transaction management, persistence, and security. It follows the next generation Java programming model: Dependency Injection, aspect-oriented programming, annotations, object-relational mapping, and generic data access objects.

In the next chapter, we will start developing ICEcube, beginning with the page layout for desktop-like user interface presentations.

Download at WoweBook.Com

3
User Interface Design

Before we take a more detailed look at the ICEfaces components, we will discuss the desktop character of modern web applications in this chapter. Desktop technology is about 40 years old now and there are best practices that can help us in our web application design.

An important part of the user interface design is the page layout. We will have a look at the corresponding design process using a mockup tool. The Facelets example from the last chapter will be extended to show how to implement such a mockup design using Facelets templating. Finally, we will have a look at the production-ready templating of ICEfusion.

Revival of the desktop

The number of desktop-like web applications is growing faster and faster. The demand for this is not a big surprise. Using full-featured desktops meant that users had to suffer from the limited-user model of the first generation Web. This usage gap is now filled by web applications that mimic desktop behavior.

However, there is a difference between the desktop and the Web. Although equipped with desktop-like presentations, web applications have to fulfill different user expectations. So, we have a revival of the desktop metaphor in the Web context; but it is mixed with user habits based on the first decade of the Web. Nevertheless, the demand for a purer desktop presentation is already foreseeable.

If you primarily followed the traditional web programming model in the past, namely the request-response pattern, you may first have to shift your mind to components and events. If you already have some desktop-programming experience, you will discover a lot of similarities. However, you will also recognize how limited the Web 2.0 programming world is in comparison to modern desktops.

Download at WoweBook.Com

The difference is understandable because desktop design has a long tradition. The first system was built at the end of the 1960s. There is a lot of experience in this domain. Best of all, we have established rules we can follow. Web design is still a challenge compared to desktop design.

Although this book cannot discuss all of the important details of today's desktop design, we will have a quick look at the basics that are applicable to nearly all user interface designs. We can subsume all this with the following question:

What makes a software system user-friendly?

Software ergonomics

Have you ever heard of the ISO standard 9241, *Ergonomics of Human System Interaction* (`http://en.wikipedia.org/wiki/ISO_9241`)? This standard describes how a system has to be designed to be human-engineered.

There are a lot of aspects in it, from the hardware design for a machine that has to be used by a human to the user interface design of software applications. A poor hardware or interface design can result in not only injury, but also mental distress that results in the waste of working time. The primary target is to prevent humans from damage.

The most important part of ISO 9241 for software developers is part 110, dialog principles. It considers the design of dialogs between humans and information systems with a focus on:

- Suitability for the task
- Suitability for learning
- Suitability for individualization
- Conformity with user expectations
- Self-descriptiveness
- Controllability
- Error tolerance

We will take a deeper look at these later.

ISO 9241-110 has its roots in a German industry standard based on research work from the early 1980s. I first had a look at all this during a study almost 20 years ago. Most interesting with part 110 is the stability of the theoretical model behind it. Independent of the technical advances of the IT industry in the last two decades, we can still apply these standards to modern web application design.

Download at WoweBook.Com

Challenges

The principles of ISO 9241-110 can help you to get better results, but they only serve as a rule. Even if you follow such principles slavishly, the result will not be automatically valuable.

Creating a useful interface is still a challenging business. You have to accept a process of trial and error, ask for customer feedback, and accept a lot of iterations in development before usability becomes your friend.

The technical limitations that derive from your framework decisions can be additionally frustrating. The problems that we have with today's AJAX technology are a good example of it, especially if you are already experienced with desktop development and its design rules.

Apply Occam's razor

Everything should be made as simple as possible, but not simpler.

Albert Einstein's quote mentions two important aspects in creative processes that are also true for user interface design:

- Reduction
- Oversimplification

Reduction

Have you ever realized how difficult it is to recognize what is important or necessary, and what is superfluous when you enter a new domain? Often, things seem to be clear and pretty simple at the first sight. However, such a perception is based on experiences that were made outside of the domain. There is a lack of essential experiences in a majority of the cases. You may have to invest several years to get the whole picture and develop an accurate understanding before you come to an adequate decision.

If your new domain is user interface design, these findings can help you to understand your customers better. If you keep questioning the eye-catching solutions that come to your mind and try to slip into the customer's role, you will get better results faster.

Download at WoweBook.Com

Oversimplification

Oversimplification makes user interfaces more complex to use. This phenomenon arises if the target group for a user interface is defined as less experienced than it is in reality. For advanced users, a beginner's design is more time-consuming to use.

In many cases, it is assumed that a bigger part of the users consists of beginners. However, reality shows us that advanced users make up the bigger part, whereas beginners and super users may have a portion of up to 10% each.

Designing a user interface for beginners that can be used by all users may be an intuitive idea at first sight, but it is not. You have to consider the advanced users if you want to be successful with your design. This is indeed an essential experience to come to an adequate decision.

User interface design principles

Besides the aforementioned recommendations, the following are the most influential principles for an adequate interface design:

- Suitability for the task
- Self-descriptiveness
- Controllability
- Conformity with user expectations
- Error tolerance
- Suitability for individualization
- Suitability for learning

Suitability for the task

Although it seems to be a trivial requirement, the functionality of a web application seldom delivers what the user requires to fulfill his needs. Additionally, the presentation, navigation, or lingo often does not work for the user or is not well-suited for the function it represents.

A good user interface design is based on the customer's lingo. You can write a glossary that describes the meaning of terms you use. A requirements management that results in a detailed use case model can help in implementing the adequate functionality. The iterative development of interactive user interface prototypes to get customer feedback allows finding a suitable presentation and navigation.

Download at WoweBook.Com

Self-descriptiveness

Ergonomic applications have an interface design that allows answering the following questions at any time:

- What is the context I am working in at the moment?
- What is the next possible step?

The answers to these questions become immediately important when a user is, for example, disrupted by a telephone call and continues his work after attending to it. The shorter the time to recognize the last working step, the better the design is.

A rule of thumb is to have a caption for every web page that describes its context. Navigational elements, such as buttons, show descriptive text that allows recognizing the function behind it. If possible, separate a page into subsections that also have their captions for a better orientation.

Controllability

Applications have to offer their functionality in a way that the user can decide for himself when and how the application is fulfilling his requirements. For this, it is important that the application offers different ways to start a function. Beginners may prefer using the mouse to select an entry in a pull-down menu. Advanced users normally work with the keyboard because hotkeys let them use the application faster.

It is also important that the user must be able to stop his/her work at any time; for example, for a lunch break or telephone call, without any disadvantages. It is not acceptable that the user has to start the last function again. With web application, this cannot be fulfilled in any case because of security reasons or limited server resources.

Conformity with user expectations

User expectations are, maybe, the most important principle, but also the most sophisticated one. The expectations are closely connected to the cultural background of the target group. So, the interface designer has to have a similar socialization.

We need to have a look at the use of words of the target language. Although target groups share the same language, certain terms can have different meanings; for example, the correct use of colors or pictures in icon bars is pretty important because we use these in contexts without extra explanation. However, there are cases when a color or an image can mean the opposite of what it was designed for.

Download at WoweBook.Com

The behavior of an application can also be a problem when it differs from the standards of real-world processes. The advantage of standardization is an immediate understanding of processing steps, or the correct use of tools without education. If an application does not consider this and varies, the standard users have to rethink every step before they can fulfill their duties. This needs extra energy, is annoying, and is pretty bad for the acceptance of the application in the long run.

If we look at the design itself, consistency in presentation, navigation, or form use is another important part. The user expects immutable behavior of the application in similar contexts. Contexts should be learned only once, and the learned ones are reusable in all other occurrences. Following this concept also helps to reuse the visual components during development. So, you have a single implementation for each context that is reused in different web pages.

Error tolerance

User acceptance also suffers if a user model does not allow mishandling of the application. The trial and error process that the user follows during the study of an application's behavior has to be error-tolerant.

It is a fact that we cannot write error-free code. User interface designers have to respect this, else they will lose user acceptance and user motivation. However, even if you have a focus on this, you skate on thin ice.

Today, it is debated as to whether an application has to show error dialogs at all. The user's frustration grows with every presentation of an error. So, if you cannot give up certain error dialogs, keep in mind that the user cannot recognize why the error occurred until your dialog box presents corresponding information. It is also important to tell the user what the next step to do is. Maybe he has to retry a function or change the last input, but there are also cases in which an administrator has to be contacted.

For this, also deliver the contact data in the error dialog to keep the time to solve a problem as short as possible for the user. It is pretty annoying if the contact data has to be searched inside the application, or even worse, in printed handbooks or other kinds of documentation that are not in direct access to the user.

Basically, it is a good idea to use defaults where possible and assume certain standard behavior of users. This helps in avoiding error dialogs. If you combine this with the corresponding personalization features, the user will be quite happy with your interface design.

Download at WoweBook.Com

Suitability for individualization

The user performance also depends on the flexibility that an application offers. The more the application can be adapted to individual solution strategies, the faster the user can produce a result. This helps to get even better results.

However, it can be problematic to understand how an *individual* should be interpreted. Quite often, this cannot be precisely described from the customer side. If possible, try to take on the role of a user and simulate work with the user interface like you would do it in production. Working with the user interface helps to recognize inflexibilities. If you add more flexibility to the user interface, keep in mind that it becomes more complex. Finding the right balance is a process. So, do not stop with the first iteration.

Suitability for learning

Nobody really wants to read long essays about how to use a web application. So, mostly, the user chooses the trial and error process to learn a new application. For this, the application has to be intuitive in its usage. It has to use the user's lingo. Additionally, it has to offer answers to questions that may come up during the usage of the application.

Modern applications offer different concepts for this. The most well-known is the context-sensitive help. Press a function key and you get information about what you can see in the application. For this, the context is calculated by the application and a corresponding help is selected in the end.

With technical progress, the text-heavy presentation of such help systems changes to a more multimedia-based content. Flash technology has pioneered this trend for the Web. Today, it is possible to add video and audio to the help system. Audio-visual presentations are more natural to users, and so we get a better benefit in shorter time.

The advances in computer-based training in the past allow a location and time-independent self-study of applications today. You can learn anything where you like and when you like. This allows the user to invest a minimal amount of time to become productive with an application.

Download at WoweBook.Com

Interface layout

Before we can implement a user interface, we have to create a design. We can differentiate between:

- The layout design: Defines the logical output sections of a web application
- The navigation design: Defines how to use the web application
- The skin design: Defines how a web application looks

Create drafts with a mockup tool

A pencil and a sheet of paper are the right tools to create a first draft. However, with more than one iteration, the redesign process becomes tedious. Even if you scan your paperwork and re-edit your design with a graphical editor, the process is not really handy.

Alternatively, you can use a **Graphical User Interface** (**GUI**) designer. A lot of IDEs deliver one, but if the result cannot be used in development, you invest a lot of time for nothing. Often, you do not need so many details for the ongoing discussion with customers and other stakeholders.

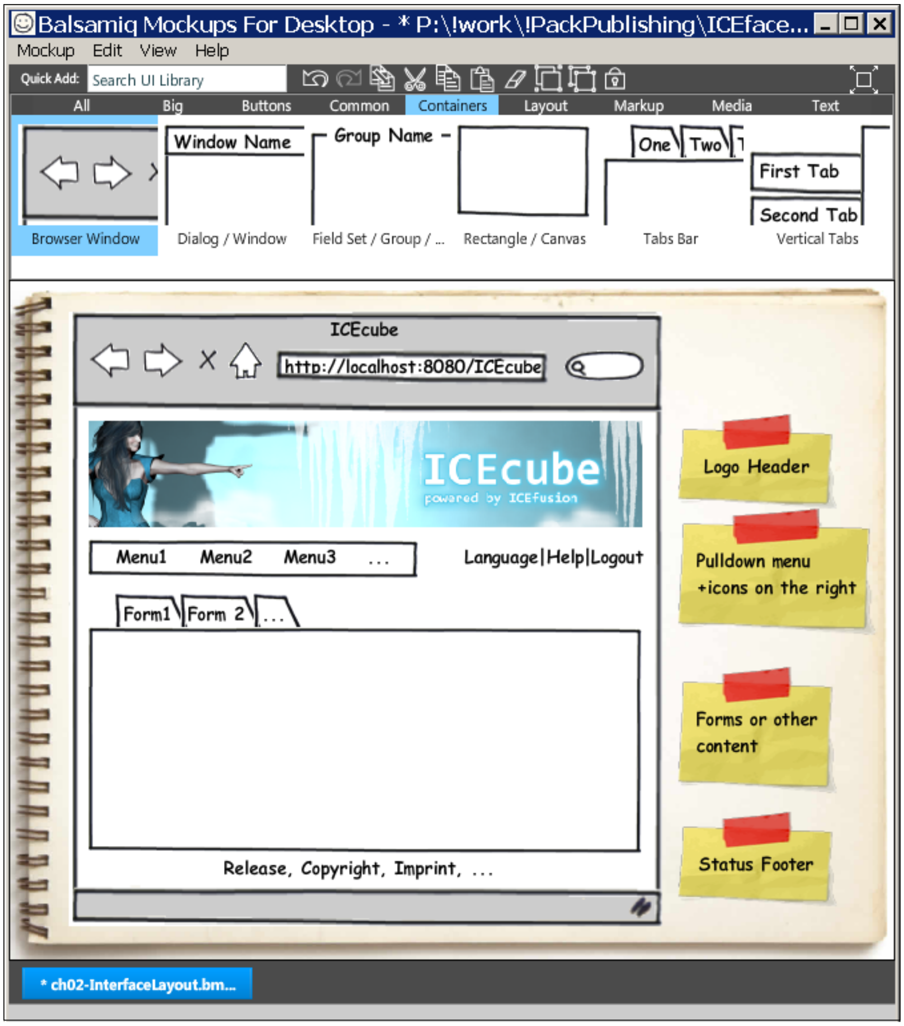
There is a new category on the horizon that allows you to create simple designs very fast: mockup tools. These offer simplified graphical elements representing user interface components you can combine in an image.

One of the most appealing tools in this category is Balsamiq Mockup (`http://www.balsamiq.com`). The presentation is similar to hand-drawn pictures, and the selection of user interface components and their customization is pretty fast. Best of all, the results are very useful to describe which components have to be implemented in a single web page and how the presentation (for example, position or size) will look. Experience shows that the additional text can concentrate on the backend communication to describe what has to be shown and managed in the user interface. It is seldom that the mockup design itself has to be detailed through additional text.

Download at WoweBook.Com

Layout design

The following screenshot shows a typical layout for a desktop created in Balsamiq Mockup:



The web page is separated into:

- Header
- Main navigation
- Content
- Footer

Download at WoweBook.Com

Although a lot of designers in Europe prefer to show the main navigation on the left, we choose to present it on the top. This allows the design to be more desktop-like.

Header

In most cases, the header is a part of a company's corporate design. For this, it is a good idea to manage a graphic that covers the complete section. When we talk about skinning, you will recognize that the header is an integral part of brand recognition.

Main navigation

Although the header follows traditional web design, the main navigation is more similar to desktop design. We use a common pull-down menu that offers the functionality of a web application. Besides this, additional icons offer global functions that should have direct access. This is a bit like the icon bar in a desktop application. However, it is not positioned under the pull-down menu to save space.

Saving space is one of the main differences between web design and desktop design. The browser, with its controls, already takes a certain amount of space that cannot be used for a web application. Additionally, the controls you get with ICEfaces are not as flexible in presentation as those normally used in desktop development. Although scrollable areas can be implemented, this is not comparable to the flexibility and usability of similar desktop components. With the current architecture of the Web, even AJAX cannot work wonders.

Content

The content section is the non-static part in this layout design. It is used for everything that has to be presented to the user. It may present:

- Forms to manipulate data of the backend; for example, for administration purposes
- Informational pages that show, for example, tables of data records for certain system objects
- More complex and interactive components that may allow a prompt result, such as a GoogleMap presentation

Web applications always present their content in the same window. The desktop, instead, allows us to use dialogs with a caption. This allows a better self-descriptiveness. Even modern web applications do not follow this presentation. There is no tradition in the web world for using dialogs, and user expectations are actually different.

Download at WoweBook.Com

Although the HTML `<title>` tag allows us to set a kind of caption, visually, it is never near the content it stands for. So, we need something better. There is the breadcrumb idea, a line of links that is a mixture between navigation history and caption. The caption is presented through the last link in the line. ICEfaces does not have a component for this, and managing this by hand can become a maintenance nightmare. If you try to be as near to the desktop presentation as possible, using a breadcrumb alongside a pull-down menu, this will deliver an inconsistent navigation. You may face some trouble with user expectations because of this.

A good alternative is to use a tabset. The caption can be set in the tab title. The tab frame itself delivers a visual grouping of your content just like a desktop dialog does. Although this kind of presentation may be a bit unusual, it becomes pretty useful if you recognize it for navigation purposes. It is easier for the user to navigate through web pages with similar context through a tabset than through a pull-down menu. For this, you can combine different menu entries into a single one that presents a tabset when it is clicked on.

Footer

The footer is primarily used for status information; for example, the release number of the current deployment. Additionally, you can add your copyright. If you are bound by law to show information, for example, about the legal form of your company, the footer is the right place for links that show the information in the content section. The same is true for important information that has to be shown on every page, like a privacy policy, the terms of use, or a link to an about page.

Facelets templating

To implement the layout design, we use the Facelets templating that is officially a part of the JSF specification since release 2.0. This book will only have a look at certain parts of the Facelets technology. So, we will not discuss how to configure a web project to use Facelets. You can study the source code examples of this chapter, or have a look at the developer documentation (`https://facelets.dev.java.net/nonav/docs/dev/docbook.html`) and the articles section of the Facelets wiki (`http://wiki.java.net/bin/view/Projects/FaceletsArticles`) for further details.

Download at WoweBook.Com

The page template

First of all, we define a page template that follows our mockup design. For this, we reuse the *HelloWorld* (Facelets) application from the last chapter. You can import the WAR file now if you did not create a Facelets project as given in the last chapter.

For importing a WAR file, use the menu **File | Import | Web | WAR file**. In the dialog box, click on the **Browse** button and select the corresponding WAR file from Chapter 2. Click on the **Finish** button to start the import. The run configuration is done as described in the last chapter. However, you do not have to configure the Jetty server again. Instead, it can be simply selected as your target.

We start coding with a new XHTML file in the `WebContent` folder. Use the menu **File | New | Other | Web | HTML Page** and click on the **Next** button. Use `page-template.xhtml` for **File name** in the next dialog. Click on the **Next** button again and choose **New ICEfaces Facelets.xhtml File (.xhtml)**. Click on the **Finish** button to create the file.

The ICEfaces plugin creates this code:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
<head>
  <title>
    <ui:insert name="title">
    Default title
    </ui:insert>
  </title>
</head>
<body>
   <div id="header">
       <ui:include src="/header.xhtml">
          <ui:param name="param_name" value="param_value"/>
       </ui:include>
   </div>
   <div id="content">
       <ice:form>
       </ice:form>
   </div>
</body>
</html>
```

Download at WoweBook.Com

The structure of the page is almost pure HTML. This is an advantage when using Facelets. The handling of pages is easier and can even be done with a standard HTML editor.

The generated code is not what we need. If you try to run this, you will get an error because the `header.xhtml` file is missing in the project. So, we delete the code between the `<body>` tags and add the basic structure for the templating. The changed code looks like this:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
<head>
  <title>
    <ui:insert name="title">
    Default title
    </ui:insert>
  </title>
</head>
<body>
  <table align="center" cellpadding="0" cellspacing="0">
    <tr><td><!-- header --></td></tr>
    <tr><td><!-- main navigation --></td></tr>
    <tr><td><!-- content --></td></tr>
   <tr><td><!-- footer --></td></tr>
  </table>
</body>
</html>
```

We change the `<body>` part to a table structure. You may wonder why we use a `<table>` for the layout, and even the `align` attribute, when there is a `<div>` tag and CSS. The answer is *pragmatism*. We do not follow the doctrine because we want to get a clean code and keep things simple. If you have a look at the insufficient CSS support of the Internet Explorer family and the necessary waste of time to get things running, it makes no sense to do so. The CSS support in Internet Explorer is a good example of the violation of user expectations.

Download at WoweBook.Com

We define four rows in the table to follow our layout design. You may have recognized that the `<title>` tag still has its `<ui:insert>` definition. This is the Facelets tag we use to tell the templating where we want to insert our page-specific code. To separate the different insert areas from each other, the `<ui:insert>` has a `name` attribute.

We substitute the comments with the `<ui:insert>` definitions, so that the templating can do the replacements:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
<head>
  <title>
    <ui:insert name="title">
    Default title
    </ui:insert>
  </title>
</head>
<body>
  <table align="center" cellpadding="0" cellspacing="0">
    <tr><td><ui:insert name="header"/></td></tr>
    <tr><td><ui:insert name="mainNavigation"/></td></tr>
    <tr><td><ui:insert name="content"/></td></tr>
    <tr><td><ui:insert name="footer"/></td></tr>
  </table>
</body>
</html>
```

The `<ui:insert>` tag allows us to set defaults that are used if we do not define something for replacement. Everything defined between `<ui:insert>` and `</ui:insert>` will then be shown instead. We will use this to define a standard behavior of a page that can be overwritten, if necessary. Additionally, this allows us to give hints in the rendering output if something that should be defined in a page is missing.

Download at WoweBook.Com

Here is the code showing both aspects:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
<head>
  <ice:outputStyle href="/xmlhttp/css/royale/royale.css" />
  <title>
    <ui:insert name="title">
    Please, define a title.
    </ui:insert>
  </title>
</head>
<body>
  <table align="center" cellpadding="0" cellspacing="0">
    <tr><td>
      <ui:insert name="header">
        <ice:graphicImage url="/logo.png" />
      </ui:insert>
    </td></tr>
    <tr><td>
      <ui:insert name="mainNavigation">
        <ice:form>
          <ice:menuBar noIcons="true">
            <ice:menuItem value="Menu 1"/>
            <ice:menuItem value="Menu 2"/>
            <ice:menuItem value="Menu 3"/>
          </ice:menuBar>
        </ice:form>
      </ui:insert>
    </td></tr>
    <tr><td>
      <ui:insert name="content">
      Please, define some content.
      </ui:insert>
    </td></tr>
    <tr><td>
      <ui:insert name="footer">
        <ice:outputText
```

Download at WoweBook.Com

```
            value="&#169; 2009 by The ICEcubes." />
        </ui:insert>
      </td></tr>
    </table>
  </body>
  </html>
```

The header, the main navigation, and the footer now have defaults. For the page title and the page content, there are messages that ask for an explicit definition. The header has a reference to an image. Add any image you like to the `WebContent` and adapt the `url` attribute of the `<ice:graphicImage>` tag, if necessary. The example project for this chapter will show the ICEcube logo. It is the logo that is shown in the mockup above. The `<ice:menuBar>` tag has to be surrounded by a `<ice:form>` tag, so that the JSF actions of the menu entries can be processed. Additionally, we need a reference to one of the ICEfaces default skins in the `<head>` tag to get a correct menu presentation. We take the Royale skin here.

If you do not know what the Royale skin looks like, you can have a look at the ICEfaces Component Showcase (`http://component-showcase.icefaces.org`) and select it in the combobox on the top left. After your selection, all components present themselves in this skin definition.

Using the template

A productive page template has a lot more to define and is also different in its structure. References to your own CSS, JavaScript, or FavIcon files are missing here. The page template would be unmaintainable soon if we were to manage the pull-down menu this way.

However, we will primarily look at the basics here. So, we keep the page template for now. Next, we adapt the existing `ICEfacesPage1.xhtml` to use the page template for its rendering.

Here is the original code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component">
  <head>
```

Download at WoweBook.Com

```
        <title>
          <ui:insert name="title">
            Default title
          </ui:insert>
        </title>
      </head>
      <body>
        <div id="header">
        <!--
          <ui:include src="/header.xhtml" >
            <ui:param name="param_name" value="param_value" />
          </ui:include>
        -->
        </div>
          <div id="content">
            <ice:form>
              <ice:outputText value="Hello World!"/>
            <!--
              drop ICEfaces components here
            -->
          </ice:form>
        </div>
      </body>
    </html>
```

We keep the `Hello World!` output and use the new page template to give some decoration to it. First of all, we need a reference to the page template so that the templating knows that it has to manage the page. As the page template defines the page structure, we no longer need a `<head>` tag definition.

You may recognize `<ui:insert>` in the `<title>` tag. This is indeed the code we normally use in a page template. It was no problem for the Chapter 2 example to have this structure in the code. Facelets has rendered the content in between because it did not find a replacement tag. Theoretically, you are free to define such statements in any location of your code. However, this is not recommended. Facelets has a look at the complete code base and matches pairs of corresponding `name` attribute definitions between `<ui:insert name="...">` and `<ui:define name="...">` tags.

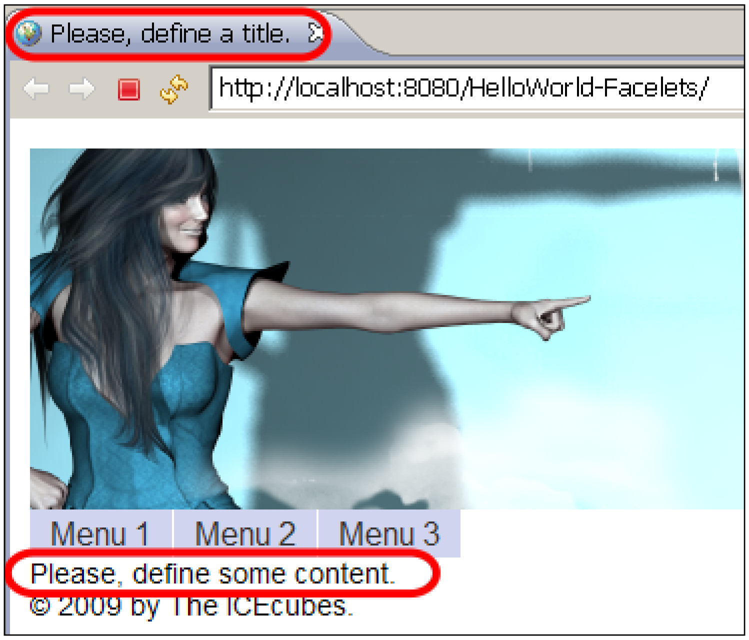
Here is the adapted code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
```

Download at WoweBook.Com

```
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
    <body>
    <ui:composition template="/page-template.xhtml">
      <div id="content">
        <ice:form>
          <ice:outputText value="Hello World!"/>
        </ice:form>
      </div>
    </ui:composition>
    </body>
    </html>
```

This code creates the following output:



Please, define a title.

http://localhost:8080/HelloWorld-Facelets/

Please, define some content.

© 2009 by The ICEcubes.

We can see our friendly reminders for the missing title and the missing content. The header, the main navigation, and the footer are rendered as expected. The structure of the template seems to be valid, although we recognize that a CSS file is necessary to define some space between the rows of our layout table.

Download at WoweBook.Com

However, something is wrong. Any idea what it is? If you have a look at the `hello-world.xhtml` again, you can find our `Hello World!` output; but this cannot be found in the rendering result. As we use the page template, we have to tell the templating where something has to be rendered in the page. However, we did not do this for our `Hello World!` output.

The following code defines the missing `<ui:define>` tag and skips the `<div>` and `<ice:form>` tags that are not really necessary here:

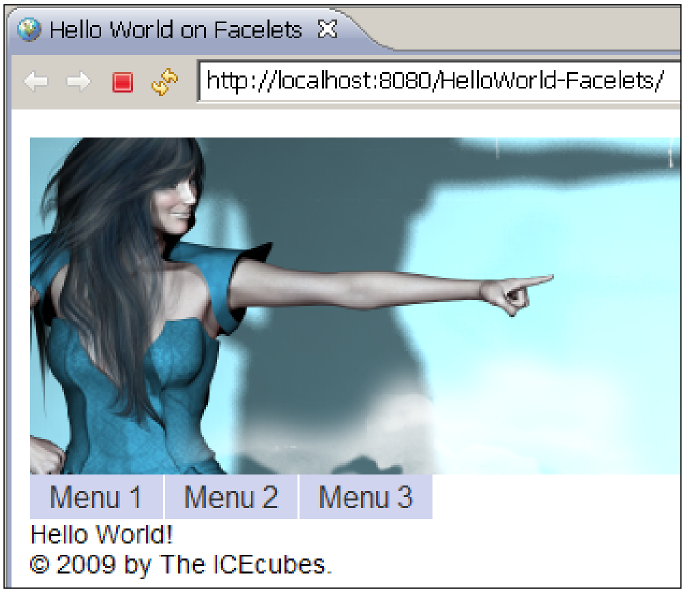
```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
<body>
<ui:composition template="/page-template.xhtml">
  <ui:define name="title">
    Hello World on Facelets
  </ui:define>
  <ui:define name="content">
    <ice:outputText value="Hello World!"/>
  </ui:define>
</ui:composition>
</body>
</html>
```

The code shows that Facelets supports textual output in two ways:

- Using a tag as we do with the `Hello World!` output
- Without a tag as we do with the `<title>` tag

Download at WoweBook.Com

The next screenshot shows the rendering result:



Hello World!
© 2009 by The ICEcubes.

The templating in ICEfusion

ICEfusion already delivers a standardized templating. This is based on experiences from productive development. We will use this for ICEcube and extend it in the coming chapters. To familiarize you with the ideas, we will first have a look at some of the implementation details.

Running ICEfusion

There is a ZIP archive for this chapter that delivers release 1.0.1 of ICEfusion. You can use this distribution for the following source code studies. For this, take care that the MySQL server is already running.

You can use the following command in Maven 2 to build the project in the `pom.xml` folder (or run `first-time-run.bat`):

```
mvn install
```

Download at WoweBook.Com

Ignore the errors that are shown during running the tests. The important thing with this run is the initialization of the database. After this, you can run Maven 2 again using the following command (or `run run.bat`):

```
mvn clean install jetty:run-war -Dmaven.test.skip=true
```

The tests will be skipped to prevent the errors and the Maven 2 internal Jetty is used for deployment. Use `http://localhost:8080` in your web browser to have a look at the application.

The ICEfusion files

The ICEfusion project follows the Maven 2 conventions. So, the ICEfusion extensions to AppFuse can be found in `/icefusion/src/main/webapp/icefusion/`. In this folder, you can find the Spring configuration files. Additionally, there are folders for JavaScripts (`/scripts/`), ICEfaces skins (`/styles/`), and the Facelets templating (`/taglibs/`). The folder names follow the AppFuse conventions.

The page layout files can be found in `/icefusion/src/main/webapp/icefusion/taglibs/commons/`. We'll have a look at the page template first (`/icefusion/src/main/webapp/icefusion/taglibs/commons/page.xhtml`).

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion="http://icefusion.googlecode.com/icefusion">
<f:view locale="#{context.locale}">
<f:loadBundle basename="icefusion.icefusion"
  var="icefusion"/>
<head>
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/page.css" />
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/icefaces.css" />
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/style.css" />
  <script type="text/javascript" src=
    "#{iceFusionConsts.contextPath}
    #{iceFusionConsts.scriptBase}/connectionStatus.js" >
  </script>
```

Download at WoweBook.Com

```
    <script type="text/javascript" src=
      "#{iceFusionConsts.contextPath}
      #{iceFusionConsts.scriptBase}/icefusion.js" />
    <link rel="shortcut icon" href=
      "#{iceFusionConsts.contextPath}
      #{iceFusionConsts.skinBase}/#{context.skin}/
      images/page.ico"/>
    <title>
      #{iceFusionConsts.application}
      #{iceFusionConsts.release} - <ui:insert name="title">
      This page has no title.</ui:insert>
    </title>
  </head>
  <body>
    <icefusion:connectionStatus />
    <table align="center" cellpadding="0" cellspacing="0"
      class="layout">
      <tr><td class="header">
        <ui:insert name="header">
          <icefusion:header/>
        </ui:insert>
      </td></tr>
      <tr><td class="navigation">
        <ui:insert name="navigation">
          <icefusion:navigation/>
        </ui:insert>
      </td></tr>
      <tr><td class="content">
        <ui:insert name="content">
          This page has no content.
        </ui:insert>
      </td></tr>
      <tr><td class="footer">
        <ui:insert name="footer">
          <icefusion:footer/>
        </ui:insert>
      </td></tr>
    </table>
    <ui:debug/>
  </body>
  </f:view>
  </html>
```

Download at WoweBook.Com

The code is similar to our example above. However, it references skin definitions that are varied via an Expression Language reference to a Spring bean, `context.skin`. The bean delivers a folder name. Each of the possible skin folders has the same folder and file structure. This allows us to switch between them without any adaptation in the templating. We will discuss this in detail in Chapter 8, *User Interface Customization.*

Another variation is the use of custom Facelets tags. We use these to define the different sections in the page layout. This is primarily done for maintenance purposes. In contrast to our Facelets example, the menu can then be managed in a dedicated file.

The new tags are managed via a Facelets tag library. We use the `icefusion` namespace to reference the tags. More details about the management of custom Facelets tags will be described in Chapter 9, *Reusable Facelets Components.*

Next, we will look at the code of the `icefusion` tags. The header can be found at `/icefusion/src/main/webapp/icefusion/taglibs/commons/header.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:graphicImage url="#{iceFusionConsts.skinBase}/
    #{context.skin}/images/logo.png" />
</ui:component>
</body>
</html>
```

This looks almost like our example, although the logo is managed via the skin selection. You may recognize the `<ui:component>` tag. This describes where the code for a Facelets tag starts and stops. Everything outside this tag is ignored.

The main navigation is defined in `/icefusion/src/main/webapp/icefusion/taglibs/commons/navigation.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
```

Download at WoweBook.Com

```
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:icefusion=
      "http://icefusion.googlecode.com/icefusion">
  <body>
  <ui:component>
    <c:if test="#{context.dynamicMenu}">
      <icefusion:dynamicMenu/>
    </c:if>
    <c:if test="#{!context.dynamicMenu}">
      <icefusion:menu/>
    </c:if>
    <icefusion:menuIcons />
  </ui:component>
  </body>
  </html>
```

The `navigation` tag already considers the mockup design. We have definitions for a menu and additional menu icons. The menu definition allows you to choose between a static menu definition and a dynamic one.

The corresponding code for the static menu tag can be found in `/icefusion/src/main/webapp/icefusion/taglibs/commons/menu.xhtml`:

```
  <html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ice="http://www.icesoft.com/icefaces/component"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:icefusion=
      "http://icefusion.googlecode.com/icefusion">
  <body>
  <ui:component>
    <ice:form>
      <ice:menuBar noIcons="true">
        <!-- Add your menu items here -->
        <!-- ICEfusion standard entries -->
        <ice:menuItem value=
          "#{icefusion['application.menu.extra']}">
          <ice:menuItem value="#{icefusion[
            'application.menu.extra.settings']}"
            action="settings"/>
          <ice:menuItem value="#{icefusion[
            'application.menu.extra.about']}"
```

Download at WoweBook.Com

```
              action="about"/>
          </ice:menuItem>
        </ice:menuBar>
      </ice:form>
    </ui:component>
    </body>
    </html>
```

The code for the dynamic menu is defined in `/icefusion/src/main/webapp/icefusion/taglibs/commons/dynamicMenu.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelGrid columns="2">
    <ice:form>
      <ice:menuBar noIcons="true">
        <ice:menuItems value="#{dynamicMenu.menuModel}" />
      </ice:menuBar>
    </ice:form>
  </ice:panelGrid>
</ui:component>
</body>
</html>
```

The menu entries are created by a special backing bean. The dynamic menu creates the same presentation as the static menu does.

The menu icons are defined in `/icefusion/src/main/webapp/icefusion/taglibs/commons/menuIcons.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
```

Download at WoweBook.Com

```
<body>
<ui:component>
  <div class="menuIcons">
    <ice:form>
      <ice:panelGroup columns="2">
        <ice:commandLink action="#{menuIcons.switchToEn}">
          <ice:graphicImage url=
            "#{iceFusionConsts.skinBase}/
            #{menuIcons.skin}/images/locale/en.png" />
        </ice:commandLink>
        <ice:commandLink action="#{menuIcons.switchToDe}">
          <ice:graphicImage url=
            "#{iceFusionConsts.skinBase}/
            #{menuIcons.skin}/images/locale/de.png" />
        </ice:commandLink>
      </ice:panelGroup>
    </ice:form>
  </div>
</ui:component>
</body>
</html>
```

Last but not the least, we will have a look at the footer code that can be found at `/icefusion/src/main/webapp/icefusion/taglibs/commons/footer.xhtml`:

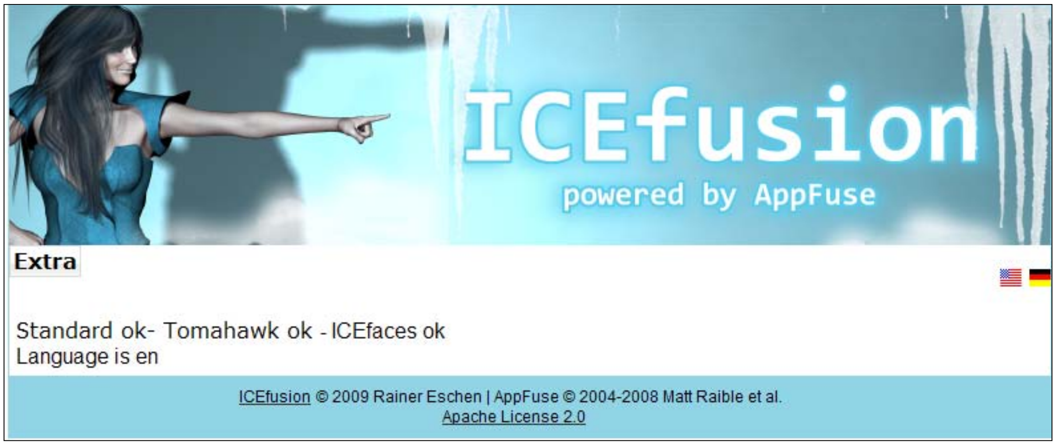
```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:outputLink value="http://icefusion.googlecode.com"
    target="_blank">
    <ice:outputText value="ICEfusion" />
  </ice:outputLink>
  <ice:outputText value=" &#169; 2009 Rainer Eschen  |
    AppFuse &#169; 2004-2008 Matt Raible et al." /><br/>
  <ice:outputLink value=
    "http://www.apache.org/licenses/LICENSE-2.0"
    target="_blank">
    <ice:outputText value="Apache License 2.0" />
```

Download at WoweBook.Com

```
      </ice:outputLink>
    </ui:component>
    </body>
    </html>
```

This footer also defines a copyright hint. It is extended with a link to the license text.

The following screenshot shows how the rendered result looks:



Standard ok- Tomahawk ok - ICEfaces ok
Language is en

ICEfusion © 2009 Rainer Eschen | AppFuse © 2004-2008 Matt Raible et al.
Apache License 2.0

In the next chapter, we will adapt the ICEfusion code base to create our ICEcube base from it. The ICEcube code base will then be iteratively extended with the sample code of the different chapters. At the end of this book, we will have a single web application that allows us to have a look at all samples in a single deployment.

Summary

The creation of desktop-like web applications is a challenging task. Following the principles of ergonomics is an important part on our way to a useful interface design. However, we have to expect several iterations before we get a suitable result. During this process, a mockup tool can help us develop page designs in a fast and easy way.

The Facelets templating can be used to implement flexible and maintainable page designs. With ICEfusion, we get a production-ready templating implementation that can be used for the creation of ICEcube. ICEcube can then be extended with the code examples from the chapters that follow.

Download at WoweBook.Com

Download at WoweBook.Com

4
Components for Navigation and Layout

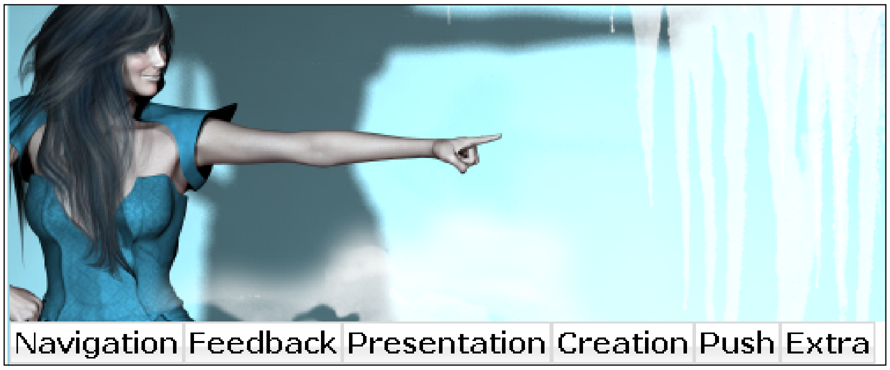
This chapter and those that follow describe important ICEfaces components in a compact manner. We start with the most important navigation and layout components here—those that allow us to define the common behavior of a web application and its layout structure.

Static pull-down menu

The most important navigation component in desktop applications is the pull-down menu. It groups the functional areas of an application to give a compact overview. The main menu entries are the navigational starting points. So, their labels have to be selected with care. One or two words have to describe what is behind the menu. The labels should be taken from the customer's domain. Their meaning should be clear and should leave no room for misunderstanding.

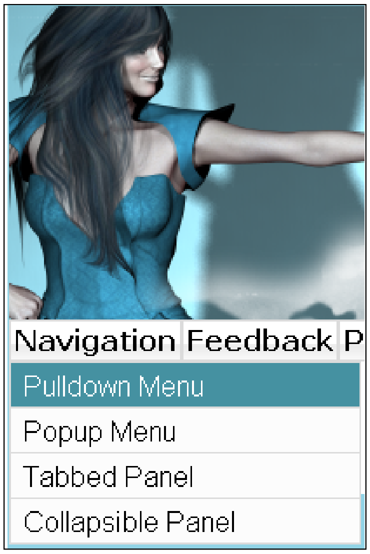
Download at WoweBook.Com

Here is a screenshot of the pulldown menu from ICEcube:



For ICEcube, the main menu entries show the content of this chapter as well as those that follow. Each submenu lists entries that allow you to select the different component examples of a chapter.

This chapter will tell you something about **Navigation**.



A pulldown menu is defined through different tags. We have a parent tag, `<ice:menuBar>`, and child tags `<ice:menuItem>` and `<ice:menuItemSeparator>`. Both can be combined as often as required to build a hierarchical structure.

Download at WoweBook.Com

For the presentation in the screenshots, the main navigation of ICEfusion (we had a look at it in the last chapter) was adapted (`/src/main/webapp/icefusion/taglibs/commons/menu.xhtml`).

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:form>
    <ice:menuBar noIcons="true">
      <ice:menuItem value="#{icecube[
        'application.menu.navigation']}">
        <ice:menuItem value="#{icecube[
          'application.menu.navigation.pulldownMenu']}"
          action="pulldownMenu"/>
        <ice:menuItem value="#{icecube[
          'application.menu.navigation.popupMenu']}"
          action="popupMenu"/>
        <ice:menuItem value="#{icecube[
          'application.menu.navigation.tabbedPanel']}"
          action="tabbedPanel"/>
        <ice:menuItem value="#{icecube[
          'application.menu.navigation
            .collapsiblePanel']}"
          action="collapsiblePanel"/>
      </ice:menuItem>
...
      <!-- ICEfusion standard entries -->
      <ice:menuItem value="#{icefusion[
        'application.menu.extra']}">
        <ice:menuItem value="#{icefusion[
          'application.menu.extra.settings']}"
          action="settings"/>
        <ice:menuItem value="#{icefusion[
          'application.menu.extra.about']}"
          action="about"/>
      </ice:menuItem>
    </ice:menuBar>
  </ice:form>
</ui:component>
</body>
</html>
```

Download at WoweBook.Com

The file describes the structure of the ICEcube pull-down menu. The original ICEfusion menu, **Extra**, was retained. The highlighted code marks the **Navigation** menu definition that corresponds to this chapter. The ICEcube example code also shows the menu entries of all the chapters that will follow.

The pulldown menu is used in a minimalistic style and without any icons. For a single `menuItem`:

- The `value` attribute describes the menu label. It is defined through a resource bundle ID to support multilingual presentations (`#{resource_bundle_id['label_id']}`).
- The `action` attribute defines which page context should be used when the menu entry is clicked on. JSF defines those contexts through a **view id**; but in our case, the term **navigation id** describes it better.

Resource bundles

We use two resource bundles here:

- `icefusion` for the **Extra** menu
- `icecube` for all the additional ICEcube menus

Basically, all ICEcube definitions (`*.java`, `*.xhtml`, `*.properties`, and so on) are separated from those that were created for ICEfusion. For this, you can find an additional resource bundle entry in the page template (`/src/main/webapp/icefusion/taglibs/commons/page.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:icefusion="http://icefusion.googlecode.com/
      icefusion">
<f:view locale="#{context.locale}">
<f:loadBundle basename="icefusion.icefusion" var=
  "icefusion"/>
<f:loadBundle basename="icecube.icecube" var="icecube"/>
<head>
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/page.css" />
```

Download at WoweBook.Com

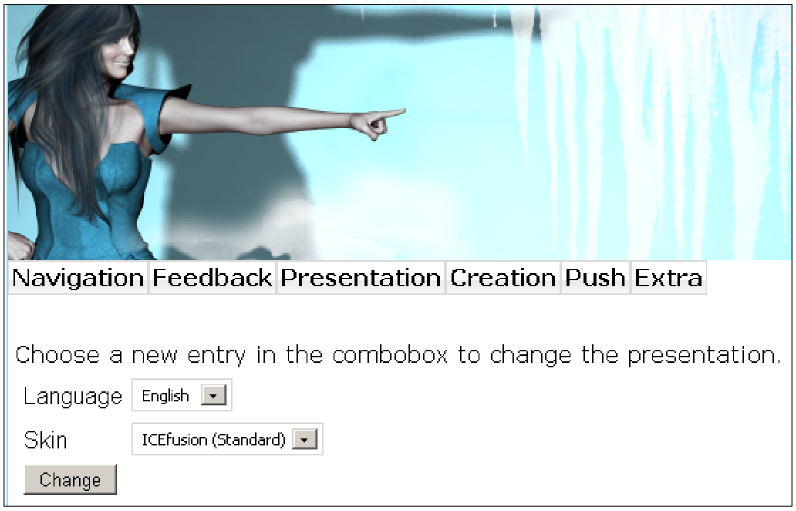
```
    <ice:outputStyle href="#{iceFusionConsts.skinBase}/
      #{context.skin}/icefaces.css" />
    <ice:outputStyle href="#{iceFusionConsts.skinBase}/
      #{context.skin}/style.css" />
    <script type="text/javascript" src=
      "#{iceFusionConsts.contextPath}
      #{iceFusionConsts.scriptBase}/connectionStatus.js" >
    </script>
    <script type="text/javascript" src=
      "#{iceFusionConsts.contextPath}
      #{iceFusionConsts.scriptBase}/icefusion.js" />
    <link rel="shortcut icon" href=
      "#{iceFusionConsts.contextPath}
      #{iceFusionConsts.skinBase}/#{context.skin}/
      images/page.ico"/>
    <title>#{iceFusionConsts.application}
      #{iceFusionConsts.release} - <ui:insert name="title">
        This page has no title.</ui:insert>
    </title>
  </head>
  <body>
    <icefusion:connectionStatus />
    <table align="center" cellpadding="0" cellspacing="0"
      class="layout">
      <tr><td class="header">
        <ui:insert name="header">
          <icefusion:header/>
        </ui:insert>
      </td></tr>
      <tr><td class="navigation">
        <ui:insert name="navigation">
          <icefusion:navigation/>
        </ui:insert>
      </td></tr>
      <tr><td class="content">
        <ui:insert name="content">
          This page has no content.
        </ui:insert>
      </td></tr>
      <tr><td class="footer">
        <ui:insert name="footer">
          <icefusion:footer/>
        </ui:insert>
      </td></tr>
```

Download at WoweBook.Com

```
        </table>
        <ui:debug/>
    </body>
    </f:view>
    </html>
```

The resource bundle files can be found in the Maven 2 resource folder structure (`/src/main/resources/icefusion/` and `/src/main/resources/icecube/`). All `*.properties` files with `icefusion` or `icecube` in their name are a part of our locale management. As we have defined **English** as the default locale, you have to edit the English language in the `icefusion.properties` or `icecube.properties` file. Although the `icefusion_en.properties` or `icecube_en.properties` file is empty by default, it could be used for definitions that should be used in English, but not for all the other locales that you do not have a properties file for.

The additional `locale` attribute definition in our page template allows us to change the language during runtime. If we change the `locale` attribute using the **Settings** page in the **Extra** menu, the new language is used with the next page reload:



Choose a new entry in the combobox to change the presentation.

Language: English

Skin: ICEfusion (Standard)

Chapter 8, *User Interface Customization*, will describe this in more detail.

Download at WoweBook.Com

Page Navigation

The navigation IDs that are used by the `action` attributes in `/src/main/webapp/icefusion/taglibs/commons/menu.xhtml` are defined in `/src/main/webapp/icecube/faces-config.xml`. (We will take a look at the **Navigation** menu only.)

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <navigation-rule>
    <from-view-id>/*</from-view-id>
    <navigation-case>
      <from-outcome>pulldownMenu</from-outcome>
      <to-view-id>
        /icecube/navigation/pulldownMenu.xhtml
      </to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/*</from-view-id>
    <navigation-case>
      <from-outcome>popupMenu</from-outcome>
      <to-view-id>
        /icecube/navigation/popupMenu.xhtml
      </to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/*</from-view-id>
    <navigation-case>
      <from-outcome>tabbedPanel</from-outcome>
      <to-view-id>
        /icecube/navigation/tabbedPanel.xhtml
      </to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
```

Download at WoweBook.Com

```
    <from-view-id>/*</from-view-id>
    <navigation-case>
      <redirect/>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

The `<from-outcome>` definitions correspond to the `action` attributes in `/src/main/webapp/icefusion/taglibs/commons/menu.xhtml`. So if you search for a page definition of a menu entry, you have to have a look at the `<navigation-rule>` definitions.

For an easier management of the menu pages, there exists a folder for each main menu entry. The definitions for the **Navigation** menu, for example, have `/navigation/` in their `<to-view-id>`.

Dynamic pull-down menu

There are a lot of similarities between static and dynamic menus. For example, they share the same data structure, but differ in its definition. Static menus can describe their structures completely in the `xhtml` file. Dynamic menus have a pretty short entry in the `xhtml` file, but a pretty long Java code in the corresponding backing bean.

The definition of the ICEfusion dynamic pull-down menu looks like this: (`/src/main/webapp/icefusion/taglibs/commons/dynamicMenu.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelGrid columns="2">
    <ice:form>
      <ice:menuBar noIcons="true">
        <ice:menuItems value="#{dynamicMenu.menuModel}" />
```

Download at WoweBook.Com

```
        </ice:menuBar>
      </ice:form>
    </ice:panelGrid>
  </ui:component>
  </body>
  </html>
```

To create a similar structure to the static pull-down menu, the ICEfusion code in the `xhtml` file can be retained. Instead, we will extend the `DynamicMenu` backing bean at `/src/main/java/com/googlecode/icefusion/ui/commons/navigation/DynamicMenu.java`:

```
  public List<MenuItem> getMenuModel() {
    if (!this.dynamicMenu.isEmpty()) {
      return this.dynamicMenu;
    }
    this.init();
    for (Entry<String, String> main : mainMenu.entrySet()) {
      MenuItem mainItem = this.addEntry(this.dynamicMenu,
        main);
      if (main.getKey().equals("navigation")) {
        for (Entry<String, String> navigation :
          navigationMenu.entrySet()) {
          MenuItem navigationItem = this.addEntry(mainItem,
            navigation);
        }
      }
      if (main.getKey().equals("feedback")) {
        for (Entry<String, String> feedback :
          feedbackMenu.entrySet()) {
          MenuItem feedbackItem = this.addEntry(mainItem,
            feedback);
        }
      }
      if (main.getKey().equals("presentation")) {
        for (Entry<String, String> presentation :
          presentationMenu.entrySet()) {
          MenuItem presentationItem = this.addEntry(
            mainItem, presentation);
        }
      }
      if (main.getKey().equals("creation")) {
        for (Entry<String, String> creation :
          creationMenu.entrySet()) {
```

Download at WoweBook.Com

```
          MenuItem creationItem = this.addEntry(mainItem,
            creation);
        }
      }
      if (main.getKey().equals("push")) {
        for (Entry<String, String> push :
          pushMenu.entrySet()) {
          MenuItem pushItem = this.addEntry(mainItem, push);
        }
      }
      // ICEfusion standard entries
      if (main.getKey().equals("extra")) {
        for (Entry<String, String> extra :
          extraMenu.entrySet()) {
          MenuItem extraItem = this.addEntry(mainItem,
            extra);
        }
      }
    }
    return this.dynamicMenu;
  }
```

The original code already defined the menu structure for the **Extra** menu. The method was extended for the ICEcube main menu entries. If the method is called for the first time, an initialization of a menu data structure is done before the loops for each main menu entry. Use this structure to create a corresponding ICEfaces structure for the XHTML file.

The menu data structure consists of several Map definitions that are set in the init() method of the DynamicMenu backing bean:

```
ArrayList<MenuItem> dynamicMenu = new ArrayList<MenuItem>();
LinkedHashMap<String, String> mainMenu =
  new LinkedHashMap<String, String>();
LinkedHashMap<String, String> navigationMenu =
  new LinkedHashMap<String, String>();
LinkedHashMap<String, String> feedbackMenu =
  new LinkedHashMap<String, String>();
LinkedHashMap<String, String> presentationMenu =
  new LinkedHashMap<String, String>();
LinkedHashMap<String, String> creationMenu =
  new LinkedHashMap<String, String>();
LinkedHashMap<String, String> pushMenu =
  new LinkedHashMap<String, String>();
// ICEfusion standard entries
```

Download at WoweBook.Com

```
LinkedHashMap<String, String> extraMenu =
  new LinkedHashMap<String, String>();
protected void init() {
  mainMenu.put("navigation", consts.getLocalized(
    "application.menu.navigation", "icecube"));
  mainMenu.put("feedback", consts.getLocalized(
    "application.menu.feedback", "icecube"));
  mainMenu.put("presentation", consts.getLocalized(
    "application.menu.presentation", "icecube"));
  mainMenu.put("creation", consts.getLocalized(
    "application.menu.creation", "icecube"));
  mainMenu.put("push", consts.getLocalized(
    "application.menu.push", "icecube"));
  navigationMenu.put("pulldownMenu", consts.getLocalized(
    "application.menu.navigation.pulldownMenu", "icecube"));
  navigationMenu.put("popupMenu", consts.getLocalized(
    "application.menu.navigation.popupMenu", "icecube"));
  navigationMenu.put("tabbedPanel", consts.getLocalized(
    "application.menu.navigation.tabbedPanel", "icecube"));
  navigationMenu.put("collapsiblePanel", consts.getLocalized(
    "application.menu.navigation.collapsiblePanel",
    "icecube"));
  feedbackMenu.put("popupDialog", consts.getLocalized(
    "application.menu.feedback.popupDialog", "icecube"));
  feedbackMenu.put("connectionStatus", consts.getLocalized(
    "application.menu.feedback.connectionStatus",
    "icecube"));
  feedbackMenu.put("tooltip", consts.getLocalized(
    "application.menu.feedback.tooltip", "icecube"));
  feedbackMenu.put("autocomplete", consts.getLocalized(
    "application.menu.feedback.autocomplete", "icecube"));
  feedbackMenu.put("dragAndDrop", consts.getLocalized(
    "application.menu.feedback.dragAndDrop", "icecube"));
  presentationMenu.put("dragAndDrop", consts.getLocalized(
    "application.menu.feedback.dragAndDrop", "icecube"));
  creationMenu.put("dragAndDrop", consts.getLocalized(
    "application.menu.feedback.dragAndDrop", "icecube"));
  pushMenu.put("dragAndDrop", consts.getLocalized(
    "application.menu.feedback.dragAndDrop", "icecube"));
  // ICEfusion standard entries
  mainMenu.put("extra", consts.getLocalized(
    "application.menu.extra", "icefusion"));
  extraMenu.put("settings", consts.getLocalized(
    "application.menu.extra.settings", "icefusion"));
```

Download at WoweBook.Com

```
    extraMenu.put("about", consts.getLocalized(
      "application.menu.extra.about", "icefusion"));
  }
```

After the initialization, we have a main menu entry map and several submenu maps that represent the subentries of the main menu entries. The `getLocalized()` method helps to set the localized label of single menu entries. Its last parameter defines which resource bundle has to be used.

If you want to extend ICEfusion with your own menu entry definitions, you have to:

1. Extend the data map for the main menu with labels.
2. Define data maps for the submenus.
3. Extend the `init()` method with the corresponding data structure initializations.
4. Extend the `getMenuModel()` method with the `for` loops that create ICEfaces structures from your data structures.

Do not forget that the order in which this is done in the code is important.

ActionListener

There is one important thing we did not take a look at: the action handling. The static code had an attribute for this. But how is this done in the dynamic code?

To simulate the action of the static menu, we define an `ActionListener` and use the `menuItem` `id` attribute as a placeholder for the navigation ID to use when a menu entry is clicked. Here is the code from the `DynamicMenu` backing bean for setting a submenu:

```
protected MenuItem addEntry(
  MenuItem parent,Entry<String,String> entry) {
  MenuItem menuItem = new MenuItem();
  menuItem.setValue(entry.getValue());
  menuItem.setId(entry.getKey());
  menuItem.addActionListener(
    new DynamicMenuActionListener());
  parent.getChildren().add(menuItem);
  return menuItem;
}
```

Download at WoweBook.Com

The specialized `ActionListener` is defined in `/src/main/java/com/googlecode/icefusion/ui/commons/navigation/DynamicMenuActionListener.java`:

```
public void navigation(String navigationId) {
  FacesContext context = FacesContext.getCurrentInstance();
  context.getApplication().getNavigationHandler()
    .handleNavigation(context, null, navigationId);
}
public void processAction(ActionEvent actionEvent) throws
AbortProcessingException {
  if (this.getNavigationId() == null) {
    this.setNavigationId(((UIComponent)actionEvent
    .getSource()).getId());
  }
  this.navigation(this.getNavigationId());
}
```

The code in the `processAction()` method refers the `id` attribute of its menu entry and uses this to initiate a navigation through the `navigation()` method. This works like the resolution of a navigation ID in the static menu. So, we can reuse the navigation ID definitions of the static menu in `/src/main/webapp/icecube/faces-config.xml`.

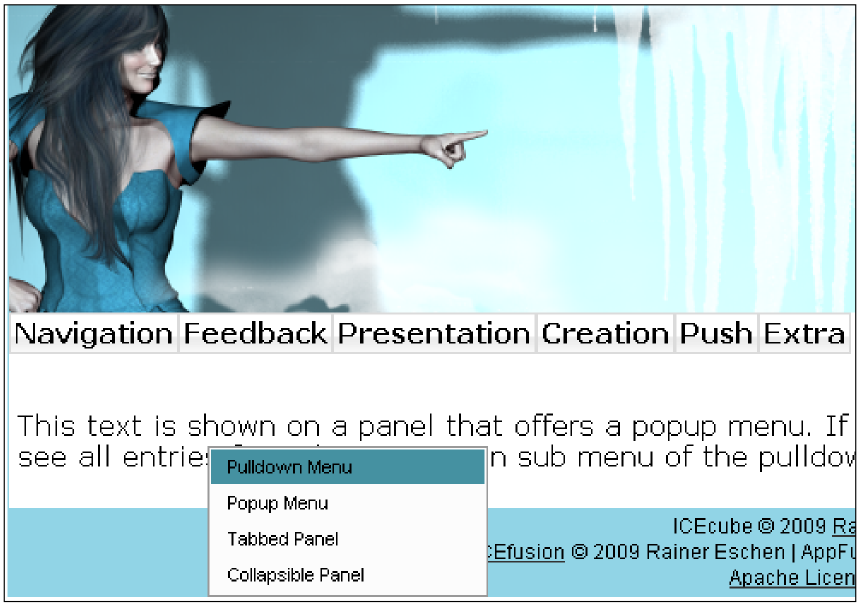
`DynamicMenuActionListener` even has a getter and a setter to manage an explicit navigation ID. So, the management of an `id` attribute of a menu entry is, in fact, coded as a fallback strategy in the case when no explicit navigation ID is set in the `ActionListener`, as we did in `DynamicMenu`.

Pop-up menu

Pop-up menus offer a certain functionality in the context of another component. For this, the user has to click on the context button of the mouse on the component and the menu is rendered. The usage of pop-up menus follows the usage of pull-down submenus.

Download at WoweBook.Com

Here is the sample implementation of ICEcube. It displays a pop-up menu when you click on the text panel that is shown on the page:



This text is shown on a panel that offers a popup menu. If
see all entrie n sub menu of the pulldov

ICEcube © 2009 Ra
Efusion © 2009 Rainer Eschen | AppFu
Apache Licen

The pop-up menu is defined in the demo page `/src/main/webapp/icecube/navigation/popupMenu.xhtml` through static menu entry definitions:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.navigation.popupMenu']}
  </ui:define>
  <ui:define name="content">
```

Download at WoweBook.Com

```
      <ice:form>
        <ice:panelGroup menuPopup="popup">
          #{icecube['application.menu.navigation
            .popupMenu.text']}
        </ice:panelGroup>
        <ice:menuPopup id="popup" noIcons="true">
          <ice:menuItem value="#{icecube[
            'application.menu.navigation.pulldownMenu']}"
            action="pulldownMenu"/>
          <ice:menuItem value="#{icecube[
            'application.menu.navigation.popupMenu']}"
            action="popupMenu"/>
          <ice:menuItem value="#{icecube[
            'application.menu.navigation.tabbedPanel']}"
            action="tabbedPanel"/>
          <ice:menuItem value="#{icecube['application
            .menu.navigation.collapsiblePanel']}"
            action="collapsiblePanel"/>
        </ice:menuPopup>
      </ice:form>
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

The definition is taken from the main menu **Navigation** of the pull-down menu. What is important with pop-up menus is the context component they are used with. The `id` attribute in the `menuPopup` tag is referenced by the `menuPopup` attribute of the `panelGroup` tag in our example. If you hover your mouse over the `panelGroup` area and click on the context mouse button, the pop-up will be shown. Clicking on one of the menu entries activates the rendering of the corresponding page defined through the `action` attribute and its navigation id.

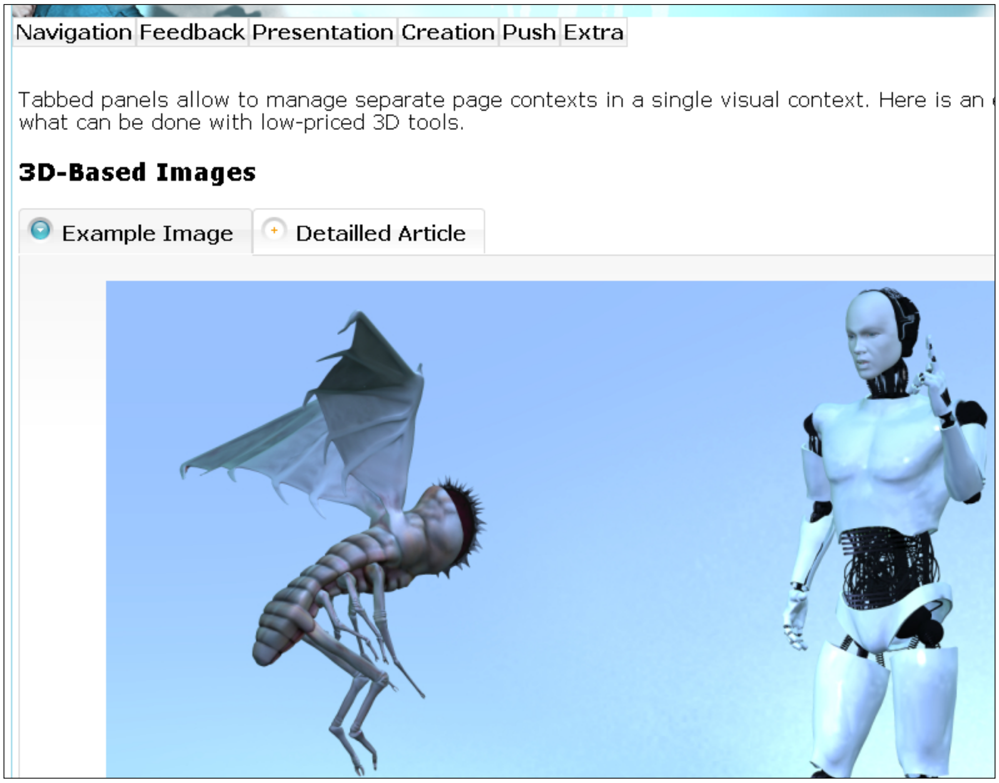
Tabbed panel

Tabbed panels allow you to segment dialog boxes in desktop applications. They are very useful if you have to present a lot of information simultaneously, but not necessarily in one page. As they show only a certain amount of information at a particular time, the user retains the overview.

We have a look at the tabbed panel in the context of navigation because it helps us to manage the caption of a web page in an elegant way. Additionally, it allows us to extend a single page with additional pages through tabs so that these can be combined in a single menu entry.

Download at WoweBook.Com

Here is a screenshot of the sample page in ICEcube:

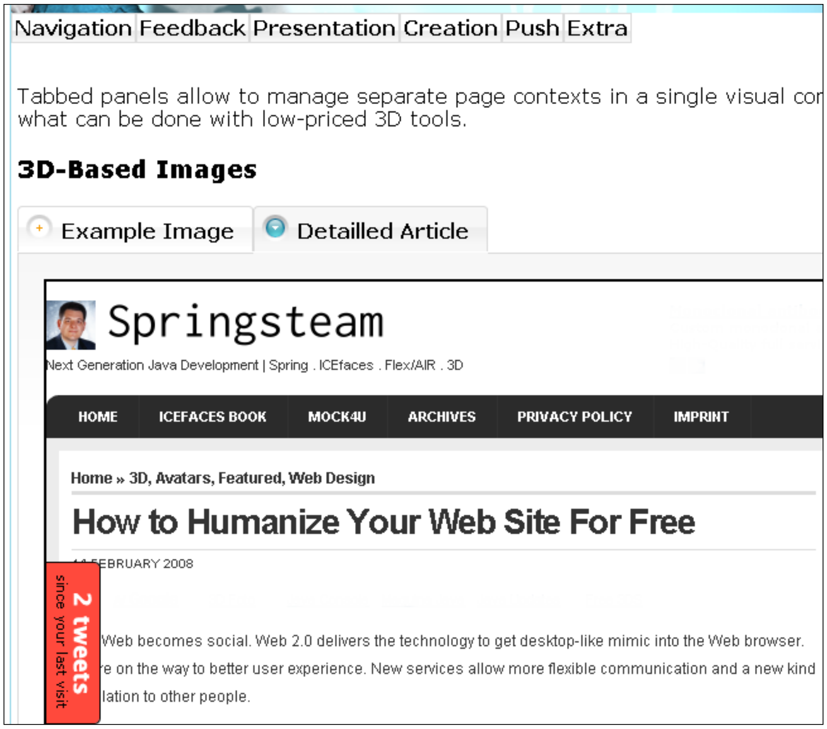


Tabbed panels allow to manage separate page contexts in a single visual context. Here is an
what can be done with low-priced 3D tools.

3D-Based Images

We have two tabs here. The first tab shows a 3D-based image that is a part of the ICEcube distribution (`/src/main/webapp/icecube/images/TheFlyBot800x600.jpg`). The second tab references a page of my blog about using 3D technology for your web pages, with a focus on human character creation (`http://blog.rainer.eschen.name/2008/02/14/how-to-humanize-your-web-site-for-free/`):

Download at WoweBook.Com


Navigation Feedback Presentation Creation Push Extra

Tabbed panels allow to manage separate page contexts in a single visual co
what can be done with low-priced 3D tools.

3D-Based Images

Example Image
Detailled Article
Springsteam
Next Generation Java Development | Spring . ICEfaces . Flex/AIR . 3D
HOME
ICEFACES BOOK
MOCK4U
ARCHIVES
PRIVACY POLICY
IMPRINT
Home » 3D, Avatars, Featured, Web Design
How to Humanize Your Web Site For Free
FEBRUARY 2008
2 tweets
since your last visit

Web becomes social. Web 2.0 delivers the technology to get desktop-like mimic into the Web browser.
e on the way to better user experience. New services allow more flexible communication and a new kind
lation to other people.

A tabbed panel is defined through different tags. We have a `<ice:panelTabSet>` parent tag that represents a container and one or more `<ice:panelTab>` child tags to present the separate pages.

In our example page (`/src/main/webapp/icecube/navigation/tabbedPanel.xhtml`), we use two `<ice:panelTab>` tags:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
```

Download at WoweBook.Com

```
        #{icecube['application.menu.navigation.tabbedPanel']}
      </ui:define>
      <ui:define name="content">
        #{icecube['application.menu.navigation.tabbedPanel
          .text']}
        <h3>#{icecube['application.menu.navigation
        .tabbedPanel.header']}</h3>
        <ice:form>
          <ice:panelTabSet>
            <ice:panelTab label="#{icecube['application
              .menu.navigation.tabbedPanel.tab1.header']}">
              <div align="center">
                <ice:graphicImage alt="#{icecube[
                  'application.menu.navigation.tabbedPanel
                  .image']}"
                  url="/icecube/images/
                    TheFlyBot800x600.jpg"/>
              </div>
            </ice:panelTab>
            <ice:panelTab label="#{icecube['application
              .menu.navigation.tabbedPanel.tab2.header']}">
              <iframe src="http://blog.rainer.eschen.name/
                2008/02/14/how-to-humanize-your-web-site-
                for-free/" width="100%" height="650px"/>
            </ice:panelTab>
          </ice:panelTabSet>
          </ice:form>
      </ui:define>
    </ui:composition>
    </body>
    </html>
```

The first tab uses a standard ICEfaces `<graphicImage>` output tag to show the image. The second tab embeds the blog page into an `<iframe>` with a predefined size to present the external web page.

Download at WoweBook.Com

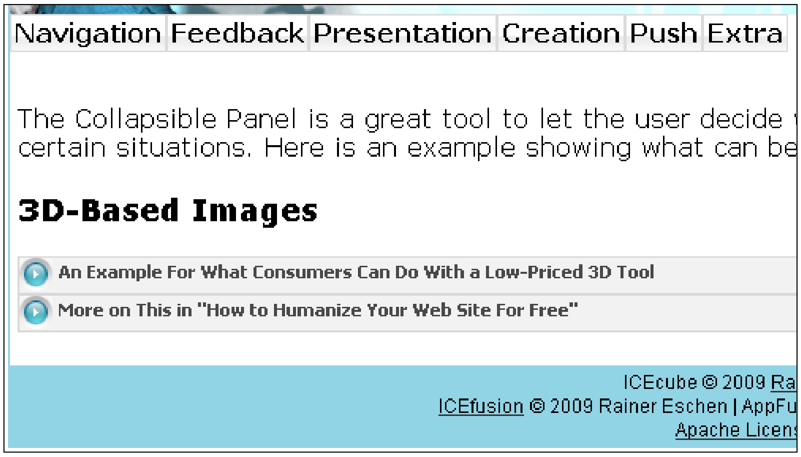
Collapsible panel

Space is an important design aspect with web applications. So, a component that allows hiding certain content on a web page during runtime is of a great help. The collapsible panel is one such component.

Besides the hiding of content, collapsible panels are also a means to visually group components. In a lot of cases, it is useful that we group something and also hide it.

[If you combine these design ideas with a session-spanning persistence of the hiding state, you get a self-managed user interface behavior.]

The ICEcube sample page for the collapsible panel looks like this:



Navigation Feedback Presentation Creation Push Extra

The Collapsible Panel is a great tool to let the user decide
certain situations. Here is an example showing what can be

3D-Based Images

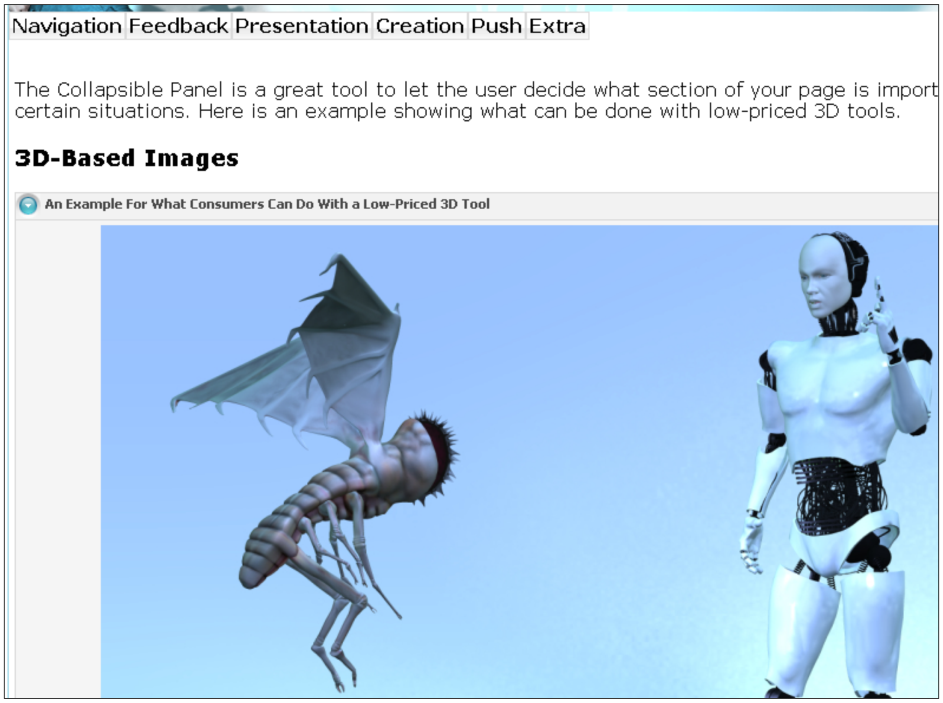
An Example For What Consumers Can Do With a Low-Priced 3D Tool

More on This in "How to Humanize Your Web Site For Free"

ICEcube © 2009 Ra
ICEfusion © 2009 Rainer Eschen | AppFu
Apache Licens

Download at WoweBook.Com

The screenshot shows two panels, both in a closed state. If we open the first panel, it looks like this:



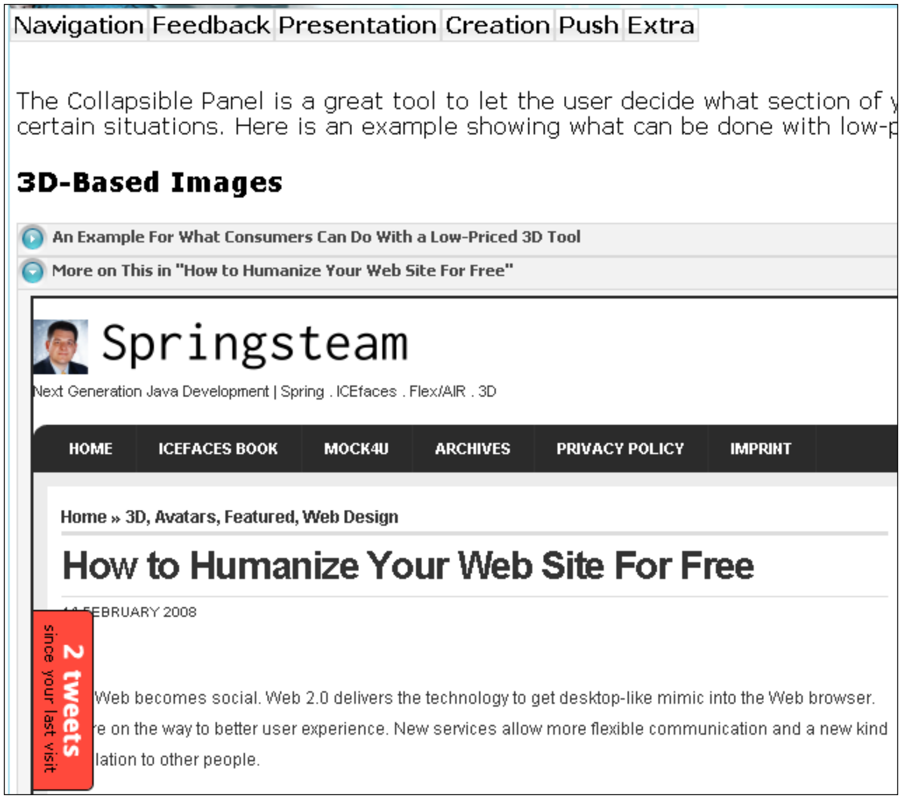
The Collapsible Panel is a great tool to let the user decide what section of your page is import
certain situations. Here is an example showing what can be done with low-priced 3D tools.

3D-Based Images

An Example For What Consumers Can Do With a Low-Priced 3D Tool

We reused the content from the ICEcube tabbed panel example. So, you can guess what the content of the second panel looks like:

Download at WoweBook.Com



The Collapsible Panel is a great tool to let the user decide what section of y
certain situations. Here is an example showing what can be done with low-p

3D-Based Images

An Example For What Consumers Can Do With a Low-Priced 3D Tool

More on This in "How to Humanize Your Web Site For Free"

Springsteam

Next Generation Java Development | Spring . ICEfaces . Flex/AIR . 3D

Home » 3D, Avatars, Featured, Web Design

How to Humanize Your Web Site For Free

EBRUARY 2008

Web becomes social. Web 2.0 delivers the technology to get desktop-like mimic into the Web browser.
e on the way to better user experience. New services allow more flexible communication and a new kind
lation to other people.

The corresponding page definition can be found at `/src/main/webapp/icecube/navigation/collapsiblePanel.xhtml`:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"   "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.navigation
```

Download at WoweBook.Com

```
          .collapsiblePanel']}
      </ui:define>
      <ui:define name="content">
        #{icecube['application.menu.navigation
        .collapsiblePanel.text']}
        <h3>#{icecube['application.menu.navigation
        .collapsiblePanel.header']}</h3>
        <ice:form>
          <ice:panelCollapsible expanded="true"
            toggleOnClick="true" >
            <f:facet name="header">
              #{icecube['application.menu.navigation
                .collapsiblePanel.panel1.header']}
            </f:facet>
            <div align="center">
              <ice:graphicImage alt="#{icecube[
                'application.menu.navigation
                .collapsiblePanel.image']}" url=
                "/icecube/images/TheFlyBot800x600.jpg"/>
            </div>
          </ice:panelCollapsible>
          <ice:panelCollapsible expanded="false"
            toggleOnClick="true" >
            <f:facet name="header">
              #{icecube['application.menu.navigation
                .collapsiblePanel.panel2.header']}
            </f:facet>
            <iframe src="http://blog.rainer.eschen.name/
              2008/02/14/how-to-humanize-your-web-site-
              for-free/" width="100%" height="650px"/>
          </ice:panelCollapsible>
        </ice:form>
      </ui:define>
    </ui:composition>
    </body>
    </html>
```

Similar to the ICEcube tabbed panel example, the first panel uses a standard ICEfaces `<graphicImage>` output tag to show the image. The second panel embeds the blog page into an `<iframe>` with a predefined size to use for the presentation of the external web page. To set a header for each panel, we have to use a `<facet>` tag named `header`.

Download at WoweBook.Com

For a better user experience, the collapsible panel got an extension with the ICEfusion implementation. With the standard ICEfaces implementation, a collapsible panel shows a simple header with a bold title and a minimally different background color. Normally, you cannot recognize that this is a clickable area to open something. For this, a special extension to the following CSS classes was made. It used the CSS file for personal skin definitions to overwrite the ICEfaces defaults (`/src/main/webapp/icefusion/styles/[skin_name]/style.css`):

```
/* Collapsible panels with visual opener or closer */
.icePnlClpsblColpsdHdr  {
  background-color: #F2F2F2;
  background-image: url( "css-images/PnlHdr_collapsed.gif" );
  background-position: left top;
  background-repeat: no-repeat;
  padding-top: 2px;
  padding-left: 25px;
  cursor: pointer
}
.icePnlClpsblHdr  {
  background-color: #F2F2F2;
  background-image: url( "css-images/PnlHdr_down.gif" );
  background-position: left top;
  background-repeat: no-repeat;
  padding-top: 2px;
  padding-left: 25px;
  cursor: pointer
}
```

Most important is the addition of images that correspond to those the tabbed panel is using to show an opened or closed state. As we have no special component construct to which we can add an image, we define a background image that is not repeated. The old background is now lost. For this, we add a standard background color that corresponds to the primary color of the images.

If you have a deeper look at the header presentation, you will recognize that some parts of the images miss a background color. The tabbed notebook headers are not designed with a complete background color set. For a perfect design, you will have to define your own images.

Download at WoweBook.Com

Summary

We learned how effective navigation and layout presentation can be done with the few ICEfaces components that were described in this chapter. Using the predefined structures of ICEfusion, it is a simple task to create a desktop-like navigation. Even if you need a more complex dynamic menu definition, the ICEcube code shows that the extensions are minimal and manageable.

In the next chapter, we will continue to have a look at components that support a desktop-like presentation. The chapter focuses on dialog components and other means to give a feedback to the user during runtime.

Download at WoweBook.Com

5
Components for Feedback and Behavior

This chapter takes a look at the components that allow you to give feedback to users. We will take a detailed look at the types of feedback that are common to desktop applications. Additionally, we will take a look at the components that allow you to give a web application a certain kind of behavior. This behavior helps the user to get fast results with less effort.

Pop-up dialog boxes

Pop-up dialog boxes are used to present processing results to the user. They can also be used to ask questions and let the user decide what the next processing step will be. They can even interrupt the current processing; for example, when errors occur.

There are three kinds of dialog boxes you will find in any modern application:

- Show a message
- Show an error
- Ask a question

Desktop systems such as Windows offer special components that let you use such dialog boxes in your application. You may only have to add text parameters to use them and ask for the dialog results. ICEfaces lacks this luxury.

Nevertheless, it is possible to get such comfort with ICEfaces too. For this, ICEfusion delivers special Facelets tags that we can use in our ICEcube application. But before we use these tags, we will take a look at the `<ice:panelPopup>` tag and how it is used to create the ICEfusion tags.

Download at WoweBook.Com

The panelPopup tag

The `panelPopup` tag allows us to present information in a predefined presentation area of the browser window. Such a pop-up is virtually presented in a layer above the current browser content. It can be decorated with a header and a frame so that it looks similar to a dialog box of a desktop application.

Here is a simple code example for a message dialog:

```
<ice:form>
  <ice:panelPopup autoCentre="false"
    draggable="false" modal="true"
    rendered="#{myBackingBean.show}"
    visible="#{myBackingBean.show}">
    <f:facet name="header">
      <ice:panelGrid>
        <ice:outputText value="My Title"/>
      </ice:panelGrid>
    </f:facet>
    <f:facet name="body">
      <ice:panelGrid>
        <ice:outputText value="My Text"/>
        <ice:commandButton value="OK"
          action="#{myBackingBean.buttonOk}"/>
      </ice:panelGrid>
    </f:facet>
  </ice:panelPopup>
</ice:form>
```

The `panelPopup` tag has attributes that define a certain behavior. So, it is possible to simulate modal dialogs when we use the `modal="true"` attribute. A modal dialog locks the application until a dialog button is clicked. This stops every other activity the web application may offer in the current presentation context.

The `panelPopup` tag also allows the presentation of movable dialogs that help display extra information besides the main browser content. For this, we define the `modal="false"` and `draggable="true"` attributes. Such dialogs have to be movable because they hide a certain area of information from the main browser content.

One important difference in desktop application frameworks is the necessity to define such dialogs as a part of the web page definition. For this, you need to have explicit visibility management in your backing bean for every dialog that is used in your page.

Download at WoweBook.Com

We use the `rendered` and `visibility` attributes in combination to control the dialog presentation. You may wonder why the `visibility` attribute cannot be used alone. It can be used alone, but then you may have to expect some side effects. If just the dialog is invisible, its content tags are still a part of the rendering process. An invisible dialog is still rendered. Experience has shown that such a dialog can tweak the presentation of the main browser content. So, to be on the safe side, always set the `rendered` attribute with the value of the `visibility` attribute.

There is another tweak with the current ICEfaces release. An activated `autoCentre` attribute creates unexpected behavior in certain browsers. When you scroll the browser window (for example, if your main browser content is longer than the visible browser area), the dialog leaves its relative position. Tests show that this attribute is not really necessary to get a centered dialog presentation.

The `panelPopup` tag seems to have a strong dependency on the quality of CSS support in web browsers. It is recommended that you test different attribute settings in your target browsers and play with the CSS classes for `panelPopup` to find a stable implementation for your project. The ICEfusion tags may help here too.

The content of a `panelPopup` tag is defined through two facets:

- Header
- Body

You are free to define what you want inside these facets. But do not forget to surround your tags with a grouping tag to follow the single tag rule for facets. It is recommended to use `<ice:panelGrid>` for this. Tests have shown tweaks when using `<ice:panelGroup>` in Mozilla browsers.

To get the behavior of a dialog box, we also need a button that closes the dialog box. The action method for this looks as follows:

```
public String buttonOK() {
  this.setShow(false);
  return null;
}
```

With these basics in mind, you can develop a questions dialog that manages, for example, a `Boolean answer` attribute via two dialog button actions, or you can extend the code to offer a stacktrace via a collapsible panel when the dialog box has access to a `Throwable exception` attribute. This is indeed how the ICEfusion dialog tags are implemented.

Download at WoweBook.Com

ICEfusion dialog tags

The ICEfusion dialog tags try to deliver a bit of the comfort that desktop developers have. However, we still need to have an eye on the self-managed visibility for dialogs.

We have dialogs for:

- Simple messages using `<icefusion:messageDialog>`
- Error handling using `<icefusion:errorDialog>`
- Asking questions using `<icefusion:questionDialog>`

To keep their usage simple, all dialogs work in a modal mode and use the same attribute names for the same contexts. Wherever possible, they offer default values that can be overwritten on demand. The corresponding backing beans offer a simple event handling for their buttons by default. This can be redefined with external references to other backing beans. Special `Interfaces` exist for this.

You can find the ICEfusion XHTML files for the dialogs in `/src/main/webapp/icefusion/taglibs/commons/dialog/` and the corresponding backing beans in `/src/main/java/com/googlecode/icefusion/ui/commons/dialog`. Each dialog also shows an image. The images are a part of the skin management and can be exchanged. Have a look at `/src/main/webapp/icefusion/styles/[skin]/images/*Dialog.png`. The look of the dialogs can be changed in `/src/main/webapp/icefusion/styles/[skin]/style.css`. Have a look at the `icePnl*` and `icePanel*` classes.

MessageDialog

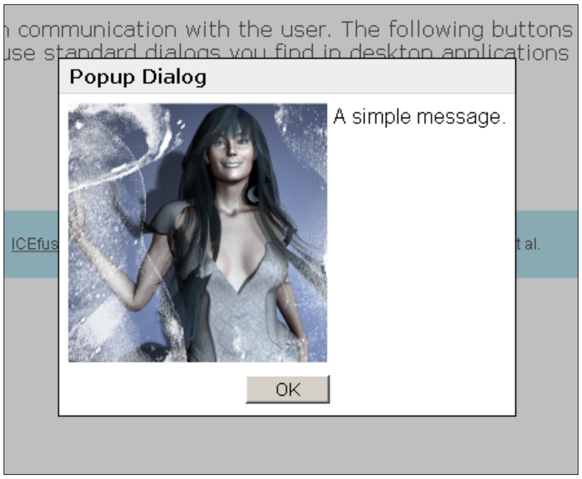
The syntax for the message dialog is:

```
<icefusion:messageDialog title="my_title" text="my_text"
  eventBean="#{my_button_event_bean}" />
```

All attributes are optional. However, this makes no real sense for the `text` attribute that delivers a **Please, define a text here.** The default for title is **Information**. The `eventBean` allows you to manage the button event by another bean instead of `MessageDialog`.

Download at WoweBook.Com

Here is an example of the message dialog. The ICEcube example page can be found in the menu at **Feedback | Popup Dialog**:


communication with the user. The following buttons
use standard dialogs you find in desktop applications
Popup Dialog
ICEfus
t al.
OK

A simple message.

The corresponding definition from the ICEcube demo looks like this (`/src/main/webapp/icecube/feedback/popupDialog.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.feedback.popupDialog']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.feedback.popupDialog.text']}
    <ice:form>
      <ice:panelGrid>
        <ice:panelGrid columns="2">
          <ice:commandButton action=
            "#{popupDialog.showMessage}" value=
```

Download at WoweBook.Com

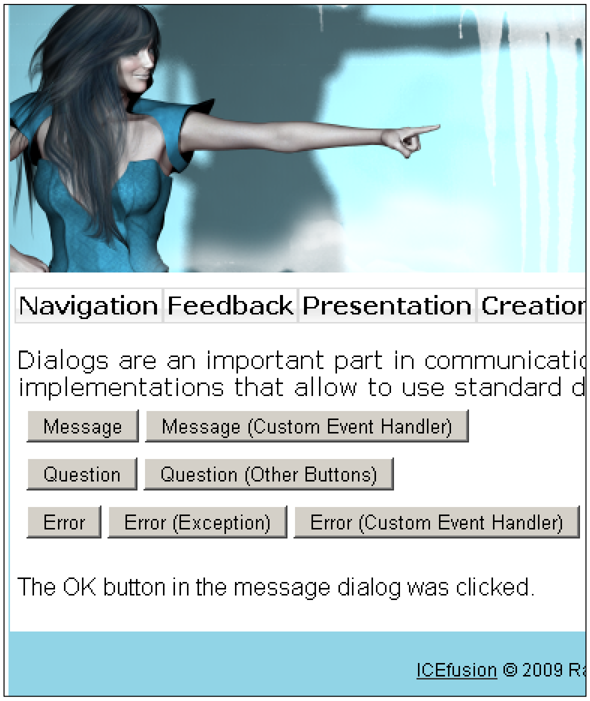
```
              "#{icecube['application.menu.feedback
              .popupDialog.messageDialog.button']}" />
            <ice:commandButton action=
              "#{popupDialog.showMessageCustomHandler}"
              value="#{icecube['application.menu
                .feedback.popupDialog.messageDialog
                .customHandler.button']}" />
          </ice:panelGrid>
        </ice:panelGrid>
      </ice:form>
      <ice:outputText value=
        "#{icecube[popupDialog.buttonClickedMessage]}" />
      <icefusion:messageDialog title=
        "#{icecube['application.menu.feedback.popupDialog']}"
        text="#{icecube['application.menu.feedback
          .popupDialog.messageDialog.text']}"
        eventBean="#{popupDialog.messageDialogHandler}"/>
      </ui:define>
   </ui:composition>
   </body>
   </html>
```

The code shows two implementations using a common `<icefusion:messageDialog>` tag for:

- The default `messageDialog` behavior
- A customized `messageDialog` using your own event handler implementation

Both implementations can be used through separate buttons. When a dialog is closed, the result is shown through an `outputText` tag like this:

Download at WoweBook.Com


Navigation Feedback Presentation Creatior
Message
Message (Custom Event Handler)
Question
Question (Other Buttons)
Error
Error (Exception)
Error (Custom Event Handler)
ICEfusion © 2009 Ra

Dialogs are an important part in communicatio
implementations that allow to use standard d

The OK button in the message dialog was clicked.

The events for the **Message** button and the **Message (Custom Event Handler)** button are managed in the page backing bean, `PopupDialog` (`/src/main/java/com/googlecode/icecube/feedback/PopupDialog.java`):

```
package com.googlecode.icecube.feedback;
import org.springframework.beans.factory.annotation
  .Autowired;
import org.springframework.beans.factory.annotation
  .Qualifier;
import com.googlecode.icefusion.ui.commons
  .BackingBeanForm;
import com.googlecode.icefusion.ui.commons.dialog
  .IMessageDialog;
import com.googlecode.icefusion.ui.commons.dialog
  .MessageDialog;
public class PopupDialog extends BackingBeanForm implements
  IMessageDialog {
  @Autowired
  @Qualifier("messageDialog")
  private MessageDialog messageDialog;
  public String showMessage () {
    this.setMessageDialogHandler(this.messageDialog);
    this.messageDialog.setShow(true);
    return null;
```

Download at WoweBook.Com

```
    }
    public String showMessageCustomHandler () {
      this.setMessageDialogHandler(this);
      this.messageDialog.setShow(true);
      return null;
    }
    public String messageDialogButtonOk() {
      this.setButtonClickedMessage(
        "application.menu.feedback.popupDialog
        .messageDialog.customHandler.result.ok");
      this.messageDialog.setShow(false);
      return null;
  }
```

The `showMessage()` and `showMessageCustomHandler()` methods manage the button clicks. Both use a reference to the `messageDialog` backing bean to set the `show` attribute.

We use the Spring `@Autowired` functionality to create a reference between two beans. All backing beans in ICEcube and ICEfusion are Spring managed. As all backing beans for dialogs have a parent class—`Dialog`—for the implementation of a common dialog behavior, Spring has problems in recognizing the correct bean. For this, we also use a `@Qualifier` to explicitly tell the name of the bean we want to reference.

Here is the important part of the Spring application context definition for ICEcube in `/src/main/webapp/icecube/spring-feedback.xml`:

```
<bean id="popupDialog"
  class="com.googlecode.icecube.feedback.
PopupDialog"
  scope="session">
   <aop:scoped-proxy/>
</bean>
```

For ICEfusion, it is in `/src/main/webapp/icefusion/spring-commons.xml`:

```
<bean id="messageDialog"
  class="com.googlecode.icefusion.ui.commons.dialog
    .MessageDialog"
  scope="session">
<aop:scoped-proxy/>
</bean>
```

With `@Autowired`, there are no longer any properties in the Spring `xml` files and the beans no longer need getter and setter for the references.

Download at WoweBook.Com

The `xhtml` file above shows a single `messageDialog` tag used for both contexts. We use a reference to the `messageDialogHandler` attribute that is pointing to a custom event handler and is changed on the fly The `showMessage()` sets the standard handler of the `messageDialog` backing bean, whereas `showMessageCustomHandler()` uses the handler of the page backing bean, namely `messageDialogButtonOk()`.

The `messageDialogButtonOk()` is defined through the `IMessageDialog` interface. The method is called when you click on the **OK** button in the opened message dialog. The implementation of the `messageDialog` backing bean only closes the dialog. In our custom event handler, a text is set for the `buttonClickedMessage` attribute to prepare the output of a status message.

The `IMessageDialog` interface looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/dialog/IMessageDialog.java`):

```
package com.googlecode.icefusion.ui.commons.dialog;
import java.io.Serializable;
public interface IMessageDialog extends Serializable {
    public String messageDialogButtonOk();
}
```

ErrorDialog

The `<icefusion:messageDialog>` and `<icefusion:errorDialog>` tags have a lot in common. This is also true for the event interfaces or backing beans. Both dialog types follow the same principles. So, if you need a more flexible handling of error dialogs, have a look at the description for message dialogs.

The syntax for the error dialog is:

```
<icefusion:errorDialog title="my_title" text="my_text"
  contact="my_support_contact"
  eventBean="#{my_button_event_bean}" />
```

Similar to the `messageDialog`, all the `errorDialog` attributes are optional. It is recommended to set the `text` attribute to get a useful context description.

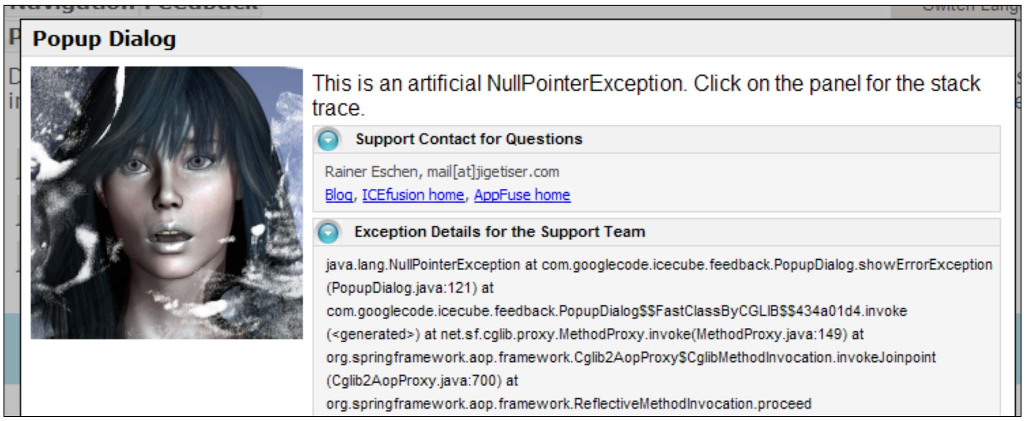
In this section, we will concentrate on the little differences in the presentation of both dialog types:

- Additional contact data for further help
- A stacktrace for error details

Download at WoweBook.Com

Both dialog types are presented as collapsible panels and can be used as standalone, or in combination. The panels are closed by default. The contact data panel helps the user to solve the problem, whereas the stacktrace panel delivers valuable information that can help the support team.

The contact data can be set in HTML. It allows us, for example, to show a detailed description of the steps to get help. Here is a simple example that shows an email address and additional links for home pages with further details. The ICEcube example page can be found in the menu at **Feedback | Popup Dialog**:



Popup Dialog

This is an artificial NullPointerException. Click on the panel for the stack trace.

Support Contact for Questions

Rainer Eschen, mail[at]jigetiser.com
Blog, ICEfusion home, AppFuse home

Exception Details for the Support Team

java.lang.NullPointerException at com.googlecode.icecube.feedback.PopupDialog.showErrorException (PopupDialog.java:121) at com.googlecode.icecube.feedback.PopupDialog$$FastClassByCGLIB$$434a01d4.invoke (<generated>) at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:149) at org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation.invokeJoinpoint (Cglib2AopProxy.java:700) at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed

It would be worth getting the support team to have a look at the stacktrace when an exception is thrown. Although your server log files should deliver the same information, it can be time consuming to find the corresponding part. If the user can copy and paste the text into an email or an instant messenger, the support team can help a lot faster.

The ICEcube example page implements for us the important parts of the `errorDialog` example like this (`/src/main/webapp/icecube/feedback/popupDialog.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
```

Download at WoweBook.Com

```
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.feedback.popupDialog']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.feedback.popupDialog.text']}
    <ice:form>
          <ice:commandButton action=
            "#{popupDialog.showErrorException}" value=
            "#{icecube['application.menu.feedback
            .popupDialog.errorDialog.exception
            .button']}" />
    </ice:form>
    <icefusion:errorDialog title=
      "#{icecube['application.menu.feedback.popupDialog']}"
      text="#{icecube[popupDialog.errorText]}"
      contact="Rainer Eschen,
        mail[at]jigetiser.com&lt;br/>&lt;a href=
        'http://blog.rainer.eschen.name' target='_blank'>
        Blog&lt;/a>, &lt;a href='http://
        icefusion.googlecode.com' target='_blank'>
        ICEfusion home&lt;/a>, &lt;a href='http://
        www.appfuse.org' target='blank'>
        AppFuse home&lt;/a>"
      eventBean="#{popupDialog.errorDialogHandler}"/>
  </ui:define>
</ui:composition>
</body>
</html>
```

To get the contact data that is presented in the screenshot, you have to set the `contact` attribute. The highlighted code above shows the corresponding `HTML` output code.

For a stacktrace, you have to set the `Exception` in the `ErrorDialog` backing bean. The page backing bean, `PopupDialog`, implements an example for this (`/src/main/java/com/googlecode/icecube/feedback/PopupDialog.java`):

```
package com.googlecode.icecube.feedback;
import org.springframework.beans.factory.annotation
  .Autowired;
import org.springframework.beans.factory.annotation
  .Qualifier;
```

Download at WoweBook.Com

```
import com.googlecode.icefusion.ui.commons
  .BackingBeanForm;
import com.googlecode.icefusion.ui.commons.dialog
  .ErrorDialog;
public class PopupDialog extends BackingBeanForm {
  @Autowired
  @Qualifier("errorDialog")
  private ErrorDialog errorDialog;
  public String showErrorException () {
    try {
      // Simulation of an exception for
      // stacktrace presentation
      throw new NullPointerException();
    }
    catch (NullPointerException e) {
      this.errorDialog.setException(e);
      this.errorDialog.setShow(true);
    }
    return null;
  }
}
```

The `showErrorException()` method implements the behavior of the **Error (Exception)** button. The `showErrorException()` is used to initialize the `errorDialog` tag. The method also opens the dialog by setting the `show` attribute.

For an impressive dialog presentation, we need a valid `Exception`. The method artificially creates one. This allows us to hand the exception over to the `errorDialog` backing bean using the `setException()` method. If the `exception` attribute is set, a collapsible panel is rendered in the dialog and the corresponding stacktrace is created as the content for the panel.

QuestionDialog

For everything that needs a decision from the user, you can use a question dialog. It allows you to present a question through the dialog text, and lets the user choose between two answers through the buttons.

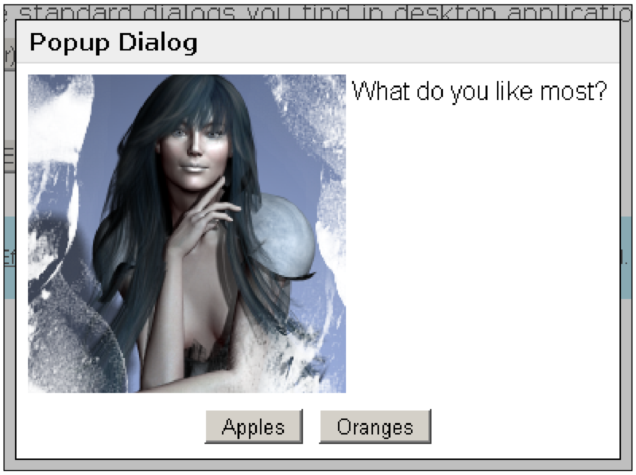
The syntax for a question dialog is:

```
<icefusion:questionDialog title="my_title" text="my_text"
  yes="my_yesButton_text" no="my_noButton_text"
  eventBean="#{my_button_event_bean}" />
```

Download at WoweBook.Com

The `title` and `text` tags follow the ideas of the other dialogs. The `yes` and `no` tags allow the user to exchange the button texts, so that you do not have to phrase questions that only allow a *yes* or *no*. The `eventBean` allows managing the button events outside of the `questionDialog` scope.

Here is an example of how a question dialog looks. The example page can be found in the ICEcube menu at **Feedback | Popup Dialog**:



A simplified parameter set to produce this presentation looks like this:

```
<icefusion:questionDialog title="Popup Dialog"
  text="What do you like most?"
  yes="Apples" no="Oranges" />
```

The ICEcube example page defines the following code to create this presentation (`/src/main/webapp/icecube/feedback/popupDialog.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
```

Download at WoweBook.Com

```
    <ui:define name="title">
      #{icecube['application.menu.feedback.popupDialog']}
    </ui:define>
    <ui:define name="content">
      #{icecube['application.menu.feedback.popupDialog.text']}
      <ice:form>
        <ice:commandButton action=
          "#{popupDialog.showQuestionOtherButtons}" value=
          "#{icecube['application.menu.feedback
          .popupDialog.questionDialog.otherButtons
          .button']}"/>
      </ice:form>
      <ice:outputText value=
        "#{icecube[popupDialog.buttonClickedMessage]}" />
      <icefusion:questionDialog title=
        "#{icecube['application.menu.feedback.popupDialog']}"
        text="#{icecube[popupDialog.questionText]}"
        yes="#{icecube[popupDialog.yesButtonText]}"
        no="#{icecube[popupDialog.noButtonText]}"
        eventBean="#{popupDialog}" />
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

As we use a single `<icefusion:questionDialog>` tag, the setting of the texts is done in the backing bean. Similar to the other discussed dialog example implementations, we have a custom event handler defined that is indeed the backing bean of the example page itself.

A custom event handler has to implement the `IQuestionDialog` interface (`/src/main/java/com/googlecode/icefusion/ui/commons/dialog/IMessageDialog.java`):

```
package com.googlecode.icefusion.ui.commons.dialog;
import java.io.Serializable;
public interface IQuestionDialog extends Serializable {
    public String questionDialogButtonYes();
    public String questionDialogButtonNo();
}
```

To understand how the event handling for a `questionDialog` works, we will take a look at the backing bean of `questionDialog`. As it implements the `IQuestionDialog` interface, we can have a look at the event handling for both:

Download at WoweBook.Com

- The standard behavior
- The implementation of a custom event handling

The `QuestionDialog` backing bean looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/dialog/QuestionDialog.java`):

```
package com.googlecode.icefusion.ui.commons.dialog;
public class QuestionDialog extends Dialog implements
  IQuestionDialog {
  private Boolean yesClicked = false;
  public String questionDialogButtonYes() {
    this.setShow(false);
    this.setYesClicked(true);
    return null;
  }
  public String questionDialogButtonNo() {
    this.setShow(false);
    this.setYesClicked(false);
    return null;
  }
}
```

The standard event handling manages the `show` attribute and the `yesClicked` attribute. The latter can be checked from outside, for example, after the dialog is closed. It is `true` if the first button (normally, the **Yes** button), was clicked. So, you do not need your own event handling implementation to find out which button was clicked.

If we use the Spring reference from the ICEcube example page backing bean (`/src/main/java/com/googlecode/icecube/feedback/PopupDialog.java`):

```
  @Autowired
  @Qualifier("questionDialog")
  private QuestionDialog questionDialog;
```

we can use the following statement after the `questionDialog` was opened:

```
if (questionDialog.getYesClicked()) {
  // Dialog was left with yes click
}
```

The `yes` and `no` attributes are used to manage the button texts. You can also change their texts through changing these attributes. If nothing is set, the buttons show a **Yes** and **No**.

Download at WoweBook.Com

Connection status

The connection status allows you to present the current communication status between a web browser and a web container. It complements the standard status presentation of the web browser. The connection status is very important when the AJAX bridge of ICEfaces manages part updates of web pages. It even allows us to recognize that an update is in preparation. These updates do not follow the classic request/response communication model that the web browser tracks in its own status presentation.

The connection status differs between:

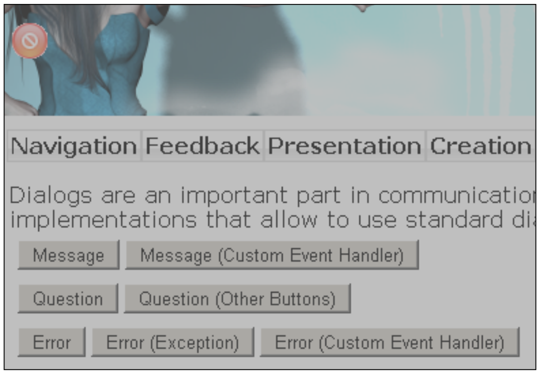
- Idle: An existing connection, but no data exchange
- Active: An existing connection with current data exchange
- Caution: A no longer existing connection because of a communication problem
- Disconnected: A terminated connection because of server timeout

The connection status is the most important feedback component in an ICEfaces application. For this, it should be a part of the page template design so that it is shown on every page. As the communication between a web browser and web container becomes a continuous process through ICEfaces' AJAX bridge, the connection status should be kept visible in any case.

ICEfusion delivers the `<icefusion:connectionStatus>` component, which fulfills this need. Independent from the length or the current visible section of a web page, the connection status is kept visible in the same position. Even when the web browser is smaller than the content, the `connectionStatus` is moved in front of the content.

Here is an example of a connection that was disconnected when the browser showed one of the modal dialog boxes. The visible area of the web page is moved, but the `connectionStatus` retains its position:

Download at WoweBook.Com



Dialogs are an important part in communicatio
implementations that allow to use standard di

The implementation of the `connectionStatus` calculates the position on the fly (`/src/main/webapp/icefusion/taglibs/commons/connectionStatus.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion="http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <div id="divStayTopLeft" style="position:absolute">
    <ice:outputConnectionStatus />
  </div>
  <script type="text/javascript">
    //<![CDATA[
    var verticalpos="fromtop"
    JSFX_FloatTopDiv();
    //]]>
  </script>
</ui:component>
</body>
</html>
```

An `<ice:outputConnectionStatus />` tag, without any extras, is put in a `<div>` tag. This tag is managed by a JavaScript from the Dynamic Drive home page (`http://www.dynamicdrive.com/`). The code can be found in `JSFX_FloatTopDiv()` at path `/src/main/webapp/icefusion/scripts/connectionStatus.js`.

Download at WoweBook.Com

To get the `connectionStatus` into every page of the ICEfusion project, and with it also in every page of the ICEcube web application, the `connectionStatus` tag was added to the `/src/main/webapp/icefusion/taglibs/commons/page.xhtml` page template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
  Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
  xhtml1-transitional.dtd">
<html xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:icefusion="http://icefusion.googlecode.com/
          icefusion">
<f:view locale="#{context.locale}">
<f:loadBundle basename="icefusion.icefusion"
  var="icefusion"/>
<f:loadBundle basename="icecube.icecube" var="icecube"/>
<head>
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/page.css" />
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/icefaces.css" />
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/style.css" />
  <script type="text/javascript"
src=
    "#{iceFusionConsts.contextPath}
    #{iceFusionConsts.scriptBase}/
    connectionStatus.js" >
  /***********************************************
  * Floating Menu script- (c) Dynamic Drive
    (http://www.dynamicdrive.com)
  * This notice MUST stay intact for legal use
  * Visit http://www.dynamicdrive.com/ for this
    script and 100s more.
  ***********************************************/
  </script>
  <script type="text/javascript"
src=
    "#{iceFusionConsts.contextPath}
    #{iceFusionConsts.scriptBase}/icefusion.js" />
  <link rel="shortcut icon"
```

Download at WoweBook.Com

```
href=
    "#{iceFusionConsts.contextPath}
    #{iceFusionConsts.skinBase}/
    #{iceFusionConsts.skin}/images/page.ico"/>
  <title>
    #{iceFusionConsts.application}
    #{iceFusionConsts.release} -
    <ui:insert name="title">
      This page has no title.
    </ui:insert>
  </title>
</head>
<body>
  <icefusion:connectionStatus />
</body>
</f:view>
</html>
```

The highlighted parts show the `connectionStatus` tag and the JavaScript file that is necessary to get the dynamic behavior. It is a part of the license agreement with Dynamic Drive that the page code show this copyright hint.

Tooltip

Tooltips were invented when desktop applications got toolbars full of icons. The icons should help us to use important functions in an application faster. But the icon images were seldom intuitive enough to recognize what function could be found behind an icon. So, the interface designers gave every icon a tooltip that described their functionality.

These days, the users are familiar with a set of icons so that the use of tooltips may only be necessary when they learn a new program. But if you study what images are used for what functionality, these days you rarely find a correlation that lets you stand up and say: "Yeah, that is really intuitive". The magnifying glass icon is a good example for this. Users had to learn that it is used to *search* for something. In reality, a magnifying glass is used to magnify small things.

Nevertheless, the use of tooltips is a good idea in contexts that may need a short description. Often, the ambiguity of terms makes it necessary to have an explanation of which definition is actually meant. If you design forms, this can be a challenge that you can solve with tooltips.

Download at WoweBook.Com

Although a full-blown help page would be a better answer, often the user is not interested in reading an introduction. Instead, he wants to select explanations when they are necessary to him.

The `<icefusion:hint>` component delivers such a behavior for edit field contexts. It can render a special icon, for example, with a question mark that is positioned behind the edit field. The icon delivers the explanation with a `mouseover`. Additionally, it can be used to deliver standardized explanations for every `<ice:panelGroup>` context.

The syntax looks like this:

```
<icefusion:hint title="my_title" text="my_text"
  panel="my_panel_context" />
```

The `title` attribute allows you to define a title for the explanation. It is optional, so that you can get the classical design of tooltips if you do not define it. The `text` attribute shows the written explanation. The `panel` attribute defines the context the `hint` component will be used in. It follows the `<ice:panelTooltip>` behavior that can be studied in the `hint` component definition (`/src/main/webapp/icefusion/taglibs/commons/help/hint.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelTooltip hideOn="mouseout" hoverDelay="200"
    id="#{(not empty panel) ? panel : 'hintIcon'}">
    <c:if test="#{not empty title}">
      <f:facet name="header">
        <ice:outputText value="#{title}"/>
      </f:facet>
    </c:if>
    <f:facet name="body">
      <ice:outputText value="#{(not empty text) ?
        text : icefusion['application.hint.none']}"/>
    </f:facet>
  </ice:panelTooltip>
  <ice:panelGroup panelTooltip="hintIcon"
```

Download at WoweBook.Com

```
        rendered="#{empty panel}">
        <ice:graphicImage url=
          "#{iceFusionConsts.skinBase}/#{context.skin}/
          images/hint.png" />
      </ice:panelGroup>
    </ui:component>
    </body>
    </html>
```

The `panelTooltip` tag has an `id` attribute. This is referenced by a `panelGroup` tag through its `panelTooltip` attribute. If you roll your mouse over the `panelGroup`, the `panelTooltip` definition is shown near the cursor.

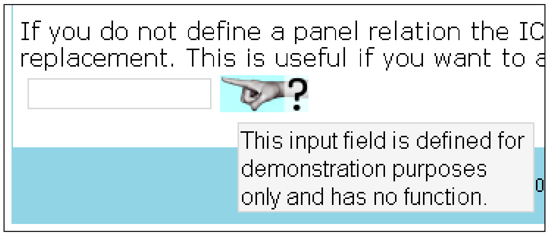
A `panelTooltip` defines two facets:

- Header
- Body

Keep in mind that you need a grouping tag such as `<ice:panelGrid>` if you use more than one tag inside a facet. The code above does not need this because it defines simple text outputs through single tags.

If the `panel` attribute of the `hint` component is not defined, the component renders an additional icon inside a `panelGroup`. This `panelGroup` automatically references the `panelTooltip`. So, you only need to position the `hint` tag behind an edit field to get the correct presentation and behavior.

Here is an example of how it looks if you skip the `panel` attribute. The ICEcube example page can be found in the menu at **Feedback | Tooltip**:


If you do not define a panel relation the IC
replacement. This is useful if you want to a
This input field is defined for demonstration purposes only and has no function.

The ICEcube page example looks like this (`/src/main/webapp/icecube/feedback/tooltip.xhtml`):

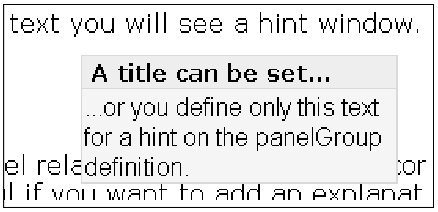
```
    <!DOCTYPE html PUBLIC
      "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns:f="http://java.sun.com/jsf/core"
```

Download at WoweBook.Com

```
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ice="http://www.icesoft.com/icefaces/component"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:t="http://myfaces.apache.org/tomahawk"
    xmlns:icefusion=
      "http://icefusion.googlecode.com/icefusion">
  <body>
  <ui:composition template="#{iceFusionConsts.templatePage}">
    <ui:define name="title">
      #{icecube['application.menu.feedback.tooltip']}
    </ui:define>
    <ui:define name="content">
      <ice:form>
        <ice:panelGroup panelTooltip="hint">
          #{icecube['application.menu.feedback
          .tooltip.text']}
        </ice:panelGroup>
        <icefusion:hint title=
          "#{icecube['application.menu.feedback.tooltip
          .hint.title']}" text=
          "#{icecube['application.menu.feedback
          .tooltip.hint.text']}" panel="hint"/>
          <br/><br/><br/>
        #{icecube['application.menu.feedback
        .tooltip.icon.text']}
        <ice:panelGrid columns="2">
          <ice:inputText readonly="true" />
          <icefusion:hint text=
            "#{icecube['application.menu.feedback.tooltip
            .hint.icon.text']}" />
        </ice:panelGrid>
      </ice:form>
      </ui:define>
  </ui:composition>
  </body>
  </html>
```

Download at WoweBook.Com

The `panelGrid` tag positions the `hint` component behind the edit field. There is no relation to a `panelGroup` or an image tag. The second code example defines a text panel with a tooltip. It uses a panel attribute definition. The tooltip also shows a title:



text you will see a hint window.

A title can be set...

...or you define only this text for a hint on the panelGroup definition.

el rel	or

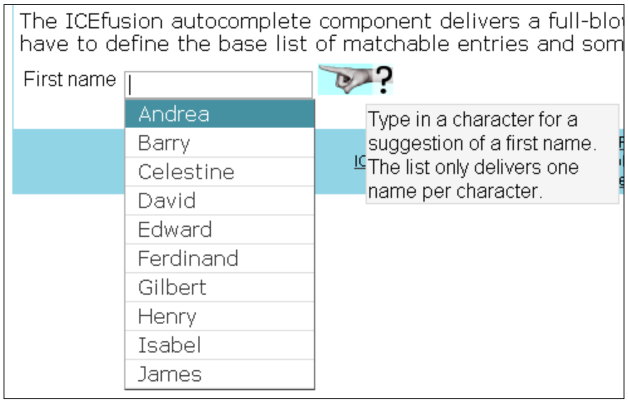
l if you want to add an explanat

Autocomplete

If you have a form and a certain field allows only predefined values, you normally use a combobox that the user can select an entry from. But there are situations where you want to offer more flexibility, or the amount of data is too high to use a combobox. For such situations, an autocomplete add-on to a simple edit field is a good solution.

The edit field is used like a standard edit field, but with the first character you type in, a search function starts its work in the background. It collects all matching entries from a data pool (for example, a database table) and presents these in a list underneath the edit field. If the number of hits exceeds the number of rows to present, only the top-rated entries are shown. If you select an entry from the list, the typed characters are replaced with it in the edit field. If you change the edit field, the search starts again and delivers new suggestions.

ICEfusion delivers the `<icefusion:completer>` component, which allows you to define edit fields with such a behavior. The following screenshot shows how this looks in ICEcube. The example page can be found in the menu at **Feedback** | **Autocomplete**:



The ICEfusion autocomplete component delivers a full-blo
have to define the base list of matchable entries and som

First name

Andrea
Barry
Celestine
David
Edward
Ferdinand
Gilbert
Henry
Isabel
James

Type in a character for a suggestion of a first name. The list only delivers one name per character.

Download at WoweBook.Com

The screenshot presents the suggestion list after the first suggestion is shown and the input is deleted. Additionally, an explanation is shown.

The ICEcube example code for the `completer` component looks like this (`/src/main/webapp/icecube/feedback/autocomplete.xhtml`):

```
<!DOCTYPE html PUBLIC

  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.feedback.autocomplete']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.feedback.autocomplete
    .text']}
    <ice:form>
      <icefusion:completer title=
        "#{icecube['application.menu.feedback
        .autocomplete.completer.title']}" hintText=
        "#{icecube['application.menu.feedback
        .autocomplete.completer.hint.text']}"
        valueBean="#{autocomplete}"/>
    </ice:form>
    </ui:define>
</ui:composition>
</body>
</html>
```

The `completer` component uses the following syntax:

```
<icefusion:completer title="my_title"
  hintTitle="my_hint_title" hintText="my_hint_text"
  valueBean="#{my_value_and_baseList_manager}"
  rows="my_number_for_entries_in_hit_list" />
```

Download at WoweBook.Com

The title attribute is used to define a textual description of the edit field that is rendered through the completer. The title is positioned in front of the edit field. The completer also uses the ICEfusion hint component that is rendered if a hintText attribute is defined. The hintTitle attribute defines an optional title for the hint. The valueBean attribute defines the reference to a bean that delivers the data pool to work with. For this, the bean has to implement the ICompleter interface. The valueBean attribute is the only attribute that is required. The rows attribute defines how many entries the list of suggestions will present. The default is 10.

The interface for the valueBean implementation looks like this (/src/main/java/com/googlecode/icefusion/ui/commons/form/ICompleter.java):

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
import java.util.List;
import javax.faces.model.SelectItem;
public interface ICompleter extends Serializable {
    public List<SelectItem> getCompleterBaseList();
    public String getCompleterValue();
    public void setCompleterValue(String value);
}
```

The interface defines the data pool list and the getter/setter for the value of the edit field. The latter is the result of all previous inputs that can be used for further processing in the form that the input field is a part of.

The completer is based on the <ice:selectInputText> tag. This allows us to define a suggestion list and a valueChangeListener, which is used to process the data pool search and the final update of the suggestion list. The completer component implementation shows the details.

We start with the XHTML code (/src/main/webapp/icefusion/taglibs/commons/form/completer.xhtml):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelGrid columns="3">
```

Download at WoweBook.Com

```
        <ice:outputText value="#{(not empty title) ?
          title : icefusion['application.completer.none']}"/>
        <ice:selectInputText value="#{valueBean.completerValue}"
          valueChangeListener="#{completer.listener}"
          rows="#{(not empty rows) ?
            rows : completer.rows}">
          <f:selectItems value="#{completer.resultList}" />
        </ice:selectInputText>

        <icefusion:hint title="#{hintTitle}"
            text="#{hintText}"/>
          </c:if>
          <c:if test="#{(empty hintTitle)}">
            <!-- Facelets need this empty attribute, else it
              would use title from the completer -->
            <icefusion:hint title="" text="#{hintText}"/>
          </c:if>
        </c:if>
      </ice:panelGrid>
      <!-- Manage valueBean list as parameter for backing
        bean -->
      <ice:selectInputText visible="false" binding=
        "#{completer.baseList}" listValue=
        "#{valueBean.completerBaseList}" />
    </ui:component>
    </body>
    </html>
```

The `completer` manages its output in three columns:

- Title
- Edit field
- Hint

The `selectInputText` references the `completerValue` implementation of the `ICompleter` interface. Additionally, it has a reference to the `completer` backing bean for the `listener` and the `resultList` management. The `listener` uses the second `selectInputText`, which manages the `valueBean` attribute to reference the `completerBaseList` implementation of the `ICompleter` interface. Using the second `selectInputText` is a trick to set the parameters of backing beans from the Facelets programming side through component binding.

Download at WoweBook.Com

The backing bean for the `completer` component looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/form/Completer.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;
import com.googlecode.icefusion.ui.commons
  .BackingBeanForm;
import com.icesoft.faces.component.selectinputtext
  .SelectInputText;
public class Completer extends BackingBeanForm {
  private List<SelectItem> matches =
    new ArrayList<SelectItem>();
  public void listener(ValueChangeEvent event) {
    if (event.getComponent() instanceof SelectInputText) {
      String search = (String) event.getNewValue();
      Long matches_i = 0L;
      matches.clear();
      for (SelectItem entry : (ArrayList<SelectItem>)
        this.getBaseList().getListValue()) {
        if ((matches_i > this.getRows())) {
          break;
        }
        if (entry.getLabel().toString()
          .toUpperCase(this.context.getSettings()
          .getLocale().getLocale()).startsWith(
          search.toUpperCase(this.context.getSettings()
          .getLocale().getLocale()))) {
          matches.add(entry);
          matches_i++;
        }
      }
    }
  }
  public List<SelectItem> getResultList() {
    return matches;
  }
}
```

Download at WoweBook.Com

The event `listener` references the second `SelectInputText` and its `listValue` attribute to get access to the data pool. Using the `baseList` attribute, all entries in this list are compared to the current input value that is delivered via the `event` object. The matching is done via a multilingual, case-insensitive search, whereas the input has to match the beginning of the data pool entries. All matching entries are collected in the `matches` list.

Drag-and-drop

Drag-and-drop is a concept that allows dragging a visual entity on the desktop and dropping it in a different place to trigger an event. The desktop file explorer is doing this when you move a file from one folder to another.

Today's AJAX frameworks also deliver this functionality for the web browser. ICEfaces' drag-and-drop feature is based on the AJAX framework, script.aculo.us. So, you can use the effects that this framework delivers in your web application.

Drag-and-drop is realized through two `<ice:panelGroup>` definitions:

```
<ice:panelGroup
  draggable="true" dragOptions="dragGhost"
  dragMask="dragging,drag_cancel,hover_start,hover_end"
  dropMask="dragging,drag_cancel,hover_start,hover_end"
  dragListener="#{dadSelector.dragListener}"
  dragValue="#{source}">
<ice:panelGroup/>
<ice:panelGroup dropTarget="true">
<ice:panelGroup/>
```

The first panel defines the draggable components that a user can move. You can use images, texts, or whatever can be grouped by a `panelGroup`. The usage of the drag-and-drop feature follows the ideas of similar desktop implementations.

The `draggable` attribute allows a simple `panelGroup` to transform into something ultra modern—a draggable area. The second `panelGroup` defines a droppable area through the `dropTarget` attribute.

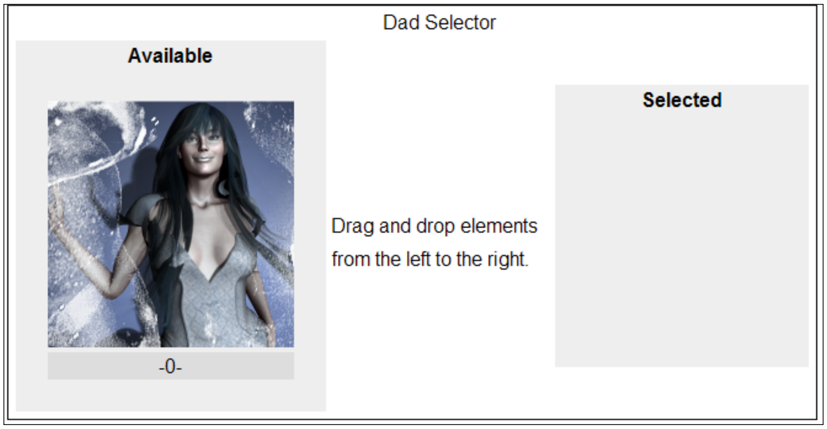
The `dragOptions` attribute allows us to influence the behavior of draggable components during drag-and-drop. The `dragGhost` parameter defines that draggable elements cannot be moved to areas that are outside of the droppable area. If you try to do so, the draggable element returns back to its homebase.

Download at WoweBook.Com

Using drag-and-drop can be very time consuming. The permanent communication with the server side needs a lot of computing power. For optimization purposes, the `dragMask` and `dropMask` attributes allow us to define events that will not be processed by the system. With a typical implementation, you define all of the allowed events here (except for the `dropped` event) that we need in order to follow the user behavior during drag-and-drop.

We track the user behavior through the event handler that is set in the `dragListener` attribute. With the `dragValue` attribute, it is possible to add common objects to the `draggable` element that can be processed by the `dragListener`.

ICEfusion delivers the `<icefusion:dadSelector>` component, which simplifies the usage of drag-and-drop. The following screenshot shows what its presentation looks like. The ICEcube example page can be found in the menu at **Feedback** | **Drag and Drop**:



Drag and drop elements from the left to the right.

The `dadSelector` has the following syntax:

```
<icefusion:dadSelector
  title="my_title"
  text="my_description_of_the_list_entries_use"
  valueBean="#{my_source_and_selected_list_manager}" />
```

Besides the `title` and `text` attributes, we know from the other ICEfusion tags that it also has a `valueBean` attribute. The `valueBean` implements the `IDadSelector` interface (`/src/main/java/com/googlecode/icefusion/ui/commons/form/IDadSelector.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
import java.util.List;
```

Download at WoweBook.Com

```
public interface IDadSelector extends Serializable {
    public List<DadSelectorItem> getDadSelectorSourceList();
    public List<DadSelectorItem>
    getDadSelectorSelectedList();
}
```

The `sourceList` represents the left side in the screenshot and manages all entries you can choose from. The `selectedList` represents the right side of the `dadSelector` and manages all elements you have chosen. The `DadSelectorItem` is an extension to the JSF `SelectorItem` and adds image attributes. This allows you to present images instead of movable text labels.

Here is the corresponding definition from the ICEcube example (`/src/main/webapp/icecube/feedback/dragAndDrop.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.feedback.dragAndDrop']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.feedback.dragAndDrop.text']}
    <ice:form>
      <icefusion:dadSelector title=
        "#{icecube['application.menu.feedback
        .dragAndDrop.dadSelector.title']}" text=
        "#{icecube['application.menu.feedback
        .dragAndDrop.dadSelector.text']}" valueBean=
        "#{dragAndDrop}" />
    </ice:form>
    </ui:define>
</ui:composition>
</body>
</html>
```

Download at WoweBook.Com

The `dragAndDrop` backing bean from the example page looks like this (`/src/main/java/com/googlecode/icecube/feedback/dragAndDrop.java`):

```
package com.googlecode.icecube.feedback;
import com.googlecode.icefusion.ui.commons
  .BackingBeanForm;
import com.googlecode.icefusion.ui.commons.form
  .IDadSelector;
import com.googlecode.icefusion.ui.commons.form
  .DadSelectorItem;
public class DragAndDrop extends BackingBeanForm
  implements IDadSelector {
  private ArrayList<DadSelectorItem> source;
  private ArrayList<DadSelectorItem> selected =
    new ArrayList<DadSelectorItem>();
  protected void init() {
    source = new ArrayList<DadSelectorItem>();
    selected = new ArrayList<DadSelectorItem>();
    DadSelectorItem item0 = new DadSelectorItem();
    item0.setImageUrl(consts.getContextPath() +
      consts.getSkinBase() + "/" + consts.getSkin() +
      "/images/messageDialog.png");
    source.add(item0);
  }
  public List<DadSelectorItem> getDadSelectorSelectedList() {
    return this.getSelected();
  }
  public List<DadSelectorItem> getDadSelectorSourceList() {
    if (this.getSource() == null) {
      this.init();
    }
    return this.getSource();
  }
}
```

The `init()` method prepares the source list that is used on the left side of the `dadSelector` presentation.

Download at WoweBook.Com

Summary

In this chapter, we discussed how we can use a subset of ICEfusion components to implement a desktop-like feedback management. We also had a look at the components that bring a desktop behavior into the web browser, such as the ICEfusion drag-and-drop component.

These ICEfusion components are enhancements to the existing ICEfaces components, and ease development. Their implementation and usage follow the ideas of modern desktop components.

In the following two chapters, we will take a deeper look at components for data management. Data management forms a central part of enterprise development. The next chapter starts with the components for the data presentation.

Download at WoweBook.Com

6
Components for Data Presentation and Multimedia

This chapter will discuss components that are primarily used to present data in different formats. We will start with the classic data table and tree components —surely, the most used components in enterprise programming. Next, we will take a look at the charts and maps that are specialized in graphical data presentation. Multimedia presentations are on the way to becoming the next important standard. So, we will also take a look at video presentation components.

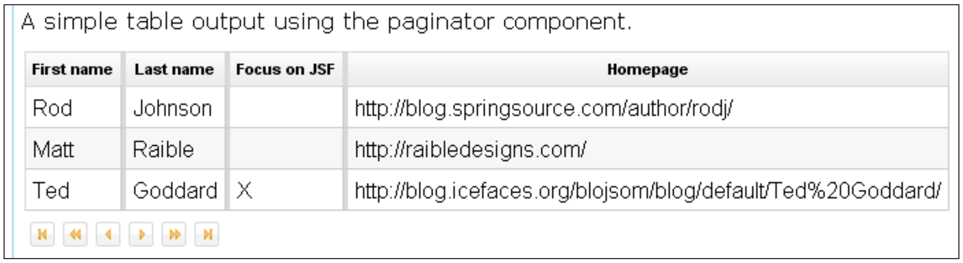
Data table

The ICEfaces data table follows the implementation of the JSF standard data table. It is primarily enhanced with skinning and security features. It also has some additional components that make its use more comfortable.

We will have a look at two variants of the ICEfaces data table in this chapter. Both implementations use the same set of data. First, we will take a look at the static variant and discuss important JSF features. Next, we will take a look at a dynamic implementation that is offered by the `<icefusion:table>` tag.

Download at WoweBook.Com

The static table example can be found in the ICEcube menu at **Presentation | Data Table** and looks like this:



A simple table output using the paginator component.

First name	Last name	Focus on JSF	Homepage
Rod	Johnson		http://blog.springsource.com/author/rodj/
Matt	Raible		http://raibledesigns.com/
Ted	Goddard	X	http://blog.icefaces.org/blojsom/blog/default/Ted%20Goddard/

In ICEfaces, a data table is defined by an `<ice:dataTable>` tag, some `<ice:column>` tags, and a list of objects. The implementation of the ICEcube example page can be found at `/src/main/webapp/icecube/presentation/dataTable.xhtml`:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name=»title»>
    #{icecube['application.menu.presentation.dataTable']}
  </ui:define>
  <ui:define name=»content»>
  #{icecube['application.menu.presentation.dataTable.text']}
    <ice:form>
      <ice:panelGrid>
        <ice:dataTable id="table" var="tableRows"
          value="#{dataTable.rowsList}"
          rows="3" resizable="true">
          <ice:column>
            <f:facet name="header">
              <ice:outputText value="#{icecube[
                'application.menu.presentation.
                dataTable.attribute.firstname']}"/>
```

Download at WoweBook.Com

```
              </f:facet>
                <ice:outputText
                  value="#{tableRows.firstname}"/>
                </ice:column>
          </ice:dataTable>
        </ice:panelGrid>
      </ice:form>
      </ui:define>
</ui:composition>
</body>
</html>
```

The list of objects that is delivered by the backing bean is referenced in the `value` attribute of the `dataTable`. The `dataTable` iterates over the `value` list and sets the `var` attribute for each member. The `column` tags define how the data has to be presented for each member.

You may have recognized the bigger lines between the columns in the screenshot. These are a result of the `resizable` attribute in the `dataTable` tag. If you click on these and move the mouse, the size of the left column is changed. Normally, you have to start with the outer right columns to get a useful result.

In the backing bean, we define a list of Java authors with their names, blog addresses, and a flag if JSF is their main business (`/src/main/java/com/googlecode/icecube/presentation/DataTable.java`):

```
package com.googlecode.icecube.presentation;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.form.ITable;
import com.googlecode.icefusion.ui.commons.form
  .ITableRowSortable;
public class DataTable extends BackingBeanForm
  implements ITable {
  private ArrayList<DataTableRow4JavaAuthors>
    authors = new ArrayList<DataTableRow4JavaAuthors>();
  protected void init() {
    authors.clear();
    authors.add(new DataTableRow4JavaAuthors(
      "Rod","Johnson",
      "http://blog.springsource.com/author/rodj/",false));
    authors.add(new DataTableRow4JavaAuthors(
      "Matt","Raible",
      "http://raibledesigns.com/",false));
    this.initialized = true;
  }
```

Download at WoweBook.Com

```
    public List<ITableRowSortable> getRowsList() {
      if (!this.initialized) {
        init();
      }
      return this.getAuthors();
    }
```

The *list of object* construct behind the `value` attribute allows us to use well-known components from the Java collections, such as `List` or `Map`. Just as they can handle any kind of object, the `dataTable` tag can also do this. In combination with the JSF **Expression Language (EL)** and its internal reflection, we are able to implement highly reusable presentation components for different business contexts.

The EL lets you reference an attribute by its name, which is independent from its type. So, it is possible to use different objects that use the same name for a certain attribute in the same context without any adaptation in the XHTML or the backing bean. This is independent from the list type or the attribute type you use with such an attribute name.

In our example, we have a `DataTableRow4JavaAuthors` class that allows us to define the necessary attributes that we want to present in the columns. The class is a simple POJO that defines attributes with getters and setters.

Pagination

Normally, the `dataTable` lists have so many objects that it is not useful to present these all at once in the web browser. So, we need a `dataTable` extension that allows limiting the number of objects to render for a single page. For this, we use the `<ice:dataPaginator>` tag. It also offers navigation inside the object list. (See the navigation bar in the screenshot above.) The ICEcube example defines the navigation like this (`/src/main/webapp/icecube/presentation/dataTable.xhtml`):

```
  <ice:form>
    <ice:panelGrid>
      <ice:dataTable id="table" var="tableRows"
        value="#{dataTable.rowsList}"
        rows="3" resizable="true">
        <ice:column>
          <f:facet name="header">
            <ice:outputText value="#{icecube[
              'application.menu.presentation.
              dataTable.attribute.firstname']}"/>
          </f:facet>
            <ice:outputText
            value="#{tableRows.firstname}"/>
```

Download at WoweBook.Com

```
      </ice:column>
    </ice:dataTable>
    <ice:dataPaginator for="table">
      <f:facet name="first">
      <ice:graphicImage style="border:none;"
        url="#{iceFusionConsts.skinBase}/ ... /
        arrow-first.gif"></ice:graphicImage>
      </f:facet>
      <f:facet name="last">
      <ice:graphicImage style="border:none;"
        url="#{iceFusionConsts.skinBase}/ ... /
        arrow-last.gif»></ice:graphicImage>
      </f:facet>
    </ice:dataPaginator>
  </ice:panelGrid>
</ice:form>
```

The `dataPaginator` allows to define navigation controls. In our example, we reference icons that are part of the ICEfaces standard skin **Rime**. The ICEfusion skins are based on **Rime** and with it, also the ICEcube skins.

The `dataTable` tag defines an `id` attribute so that the `dataPaginator` can reference to it via its `for` attribute. The `dataPaginator` becomes active when the `rows` attribute of the `dataTable` defines a number of lines that is smaller than the size of the list of objects defined via the `value` attribute.

If you define a `dataPaginator` tag, it is always rendered even if it doesn't need to be. So, it is a good idea to check if the `rows` attribute of `dataTable` is really smaller than the size of the list of objects. You can use the result in the `rendered` attribute of the `dataPaginator`.

Dynamic data table

Static tables are easy to manage. You edit the tag definition and get a new column, or change the sequence of columns through cut and paste. In more complex applications that try to reach a higher level of reuse in the presentation layer, this kind of management is too inflexible. If the column definition varies during runtime, you may have to define a table for each variation. This can be a maintenance hell in the long run.

Alternatively, you use a component that can be configured for how the presentation will look. ICEfusion delivers a `table` tag that can work like this. It integrates the features from the static table above, but decides during runtime how columns are rendered.

Download at WoweBook.Com

The ICEfusion dynamic data table allows defining:

- The columns that are presented. So, a dedicated selection of object attributes can be done.
- The sequence of columns. So, the presentation order of object attributes can be freely defined.

In comparison to the implementation we used for the static data table, the dynamic data table definition is minimalistic. Here is an example from the ICEcube implementation (`/src/main/webapp/icecube/presentation/dataTableDynamic.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
  "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.presentation.dataTable
      .dynamic']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.presentation.dataTable
      .dynamic.text']}
    <ice:form>
      <icefusion:table valueBean="#{dataTable}"
        resizable="true"/>
    </ice:form>
  </ui:define>
</ui:composition>
</body>
</html>
```

Download at WoweBook.Com

The ICEcube backing bean of the page and the backing bean of the `table` tag have to supply the missing functionality instead. The syntax for the `table` tag looks like this:

```
<icefusion:table valueBean="#{bean_that_implements_ITable}"
  resizable="true|false" rows="number_of_rows_to_show"/>
```

The `resizable` attribute and the `rows` attribute follow the description for the static data table. The `valueBean` attribute is the central part of this definition. It defines a bean that implements methods that deliver:

- The list of rows to show: The list has objects that implement the `ITableRowSortable` interface so that a column sorting is possible.
- The list of columns that have to be shown in the table: We use a `LinkedHashMap` here, which defines the attribute names and the resource bundle IDs to use for multilingual presentation of the column headers. This kind of a `Map` allows you to keep the sequence of creation.

The backing bean of the page has to implement the `ITable` interface if it manages what the `table` tag has to show. Here is the definition for `ITable` (`/src/main/java/com/googlecode/icefusion/ui/commons/form/ITable.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
public interface ITable extends Serializable {
    public List<ITableRowSortable> getRowsList();
    public LinkedHashMap<String, String> getColumnsMap();
}
```

The backing bean of the ICEcube page example looks like this (`/src/main/java/com/googlecode/icecube/presentation/DataTable.java`):

```
package com.googlecode.icecube.presentation;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.form.ITable;
import com.googlecode.icefusion.ui.commons.form.ITableRowSortable;
public class DataTable extends BackingBeanForm implements ITable {
  private ArrayList<DataTableRow4JavaAuthors>
    authors = new ArrayList<DataTableRow4JavaAuthors>();
  private LinkedHashMap<String,String>
    attributes = new LinkedHashMap<String,String>();
  protected void init() {
    authors.clear();
    authors.add(new DataTableRow4JavaAuthors(
      "Rod","Johnson",
      "http://blog.springsource.com/author/rodj/",false));
    authors.add(new DataTableRow4JavaAuthors(
```

Download at WoweBook.Com

```
         "Matt","Raible",
         "http://raibledesigns.com/",false));
       // We define the order of attribute-to-column
       attributes.clear();
       attributes.put("lastname","application.menu
         .presentation.dataTable.dynamic.attribute.lastname");
       attributes.put("firstname","application.menu
         .presentation.dataTable.dynamic.attribute
         .firstname");
       this.initialized = true;
     }
     public LinkedHashMap<String, String> getColumnsMap() {
       if (!this.initialized) {
         init();
       }
       return this.getAttributes();
     }
     public List<ITableRowSortable> getRowsList() {
       if (!this.initialized) {
         init();
       }
       return this.getAuthors();
     }
   }
```

The code shows the `ITable` methods—`getColumnsMap()` and `getRowsList()`. These deliver backing bean local definitions of a `Map` and a `List`, both of which are initialized in the `init()` method.

Sortable columns

Sortable columns form an important feature of data tables. The current JSF table implementations initialize a sorting through clicking on a column header. This only works when our list object `DataTableRow4JavaAuthors` implements the `ITableRowSortable` interface (`/src/main/java/com/googlecode/icefusion/ui/commons/form/ITableRowSortable.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
public interface ITableRowSortable extends Serializable {
    public int compareByAttribute(ITableRowSortable object,
    String attribute);
}
```

Download at WoweBook.Com

For a column-specific sorting in the table, we need a special comparator implementation that allows defining which attribute is to be used for sorting. The backing bean of the ICEfusion table uses this implementation to initialize a sorting if the user clicks on a column header. For this, `compareByAttribute()` compares the selected attribute of the object implementing the `Interface` with the same attribute of the given object and returns integer values:

- -1: The given object is bigger
- 0: Both objects are equal
- +1: The given object is smaller

The `DataTableRow4JavaAuthors` list object implements the `ITableRowSortable` interface like this (`/src/main/java/com/googlecode/icecube/presentation/DataTableRow4JavaAuthors.java`):

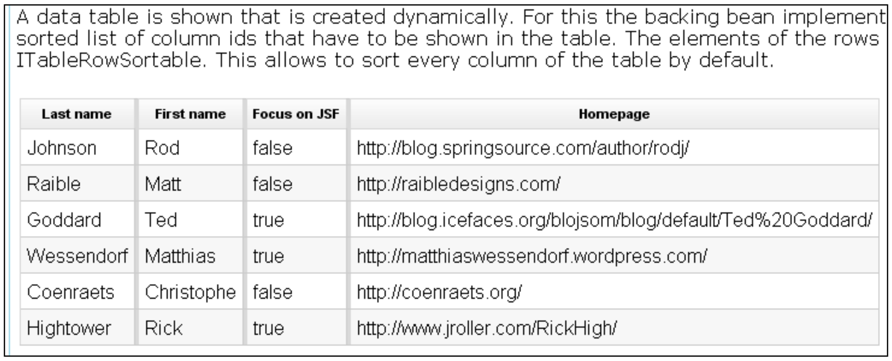
```
package com.googlecode.icecube.presentation;
import com.googlecode.icefusion.ui.commons.form
  .ITableRowSortable;
public class DataTableRow4JavaAuthors implements
  ITableRowSortable {
  public int compareByAttribute(ITableRowSortable object,
    String attribute) {
    if (attribute.equals("firstname")) {
      return this.firstname.compareTo(
        ((DataTableRow4JavaAuthors)object)
        .getFirstname());
    }
    if (attribute.equals("lastname")) {
      return this.lastname.compareTo(
        ((DataTableRow4JavaAuthors)object).getLastname());
    }
    return 0;
  }
}
```

For each attribute of the object data definition, the `compareByAttribute()` method implements a comparison. So, every attribute can be used in a sortable column of the table.

Download at WoweBook.Com

The most important part, and also the reason for the `Interface` definition, is the typecast for the `object` parameter. The `Interface` allows the backing bean of the ICEfusion table tag to keep itself unaware of the class it processes as a row. The knowledge about handling the attributes is kept in the object data class. So, the backing bean can, indeed, handle every data definition. The corresponding data classes only have to implement the `ITableRowSortable` interface.

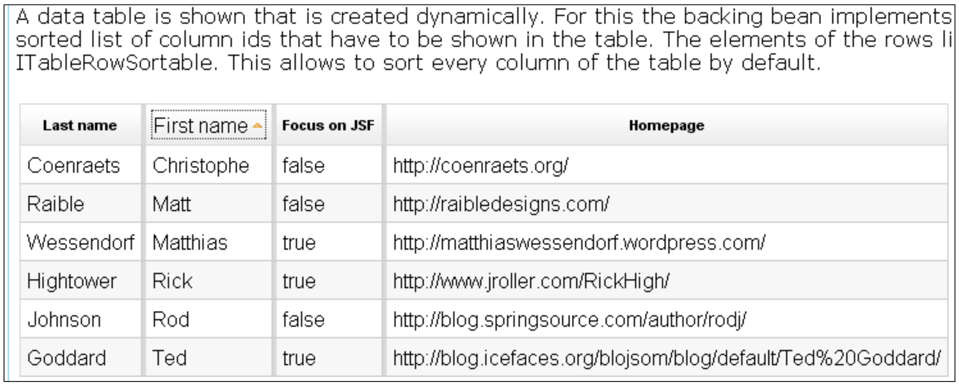
The result of the dynamic data table implementation is nearly the same as the static data table implementation. The `dataPaginator` is not active because the ICEfusion `table` tag defines that this is only shown if the `rows` parameter of the tag defines a number that is smaller than the number of the rows to process. Indeed, we have no `rows` parameter defined in the page so that all of the available rows are shown. Have a look at the ICEcube menu at **Presentation | Dynamic Data Table**:



A data table is shown that is created dynamically. For this the backing bean implement
sorted list of column ids that have to be shown in the table. The elements of the rows
ITableRowSortable. This allows to sort every column of the table by default.

Last name	First name	Focus on JSF	Homepage
Johnson	Rod	false	http://blog.springsource.com/author/rodj/
Raible	Matt	false	http://raibledesigns.com/
Goddard	Ted	true	http://blog.icefaces.org/blojsom/blog/default/Ted%20Goddard/
Wessendorf	Matthias	true	http://matthiaswessendorf.wordpress.com/
Coenraets	Christophe	false	http://coenraets.org/
Hightower	Rick	true	http://www.jroller.com/RickHigh/

If you click on one of the column headers, a sorting is done. We click on the **First name** column for this:

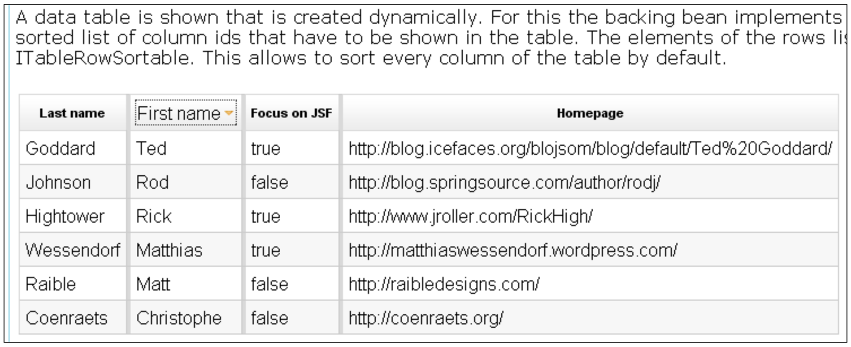


A data table is shown that is created dynamically. For this the backing bean implements
sorted list of column ids that have to be shown in the table. The elements of the rows li
ITableRowSortable. This allows to sort every column of the table by default.

Last name	First name	Focus on JSF	Homepage
Coenraets	Christophe	false	http://coenraets.org/
Raible	Matt	false	http://raibledesigns.com/
Wessendorf	Matthias	true	http://matthiaswessendorf.wordpress.com/
Hightower	Rick	true	http://www.jroller.com/RickHigh/
Johnson	Rod	false	http://blog.springsource.com/author/rodj/
Goddard	Ted	true	http://blog.icefaces.org/blojsom/blog/default/Ted%20Goddard/

Download at WoweBook.Com

A second click on the column header flips the sorting:



A data table is shown that is created dynamically. For this the backing bean implements sorted list of column ids that have to be shown in the table. The elements of the rows li ITableRowSortable. This allows to sort every column of the table by default.

Last name	First name	Focus on JSF	Homepage
Goddard	Ted	true	http://blog.icefaces.org/blojsom/blog/default/Ted%20Goddard/
Johnson	Rod	false	http://blog.springsource.com/author/rodj/
Hightower	Rick	true	http://www.jroller.com/RickHigh/
Wessendorf	Matthias	true	http://matthiaswessendorf.wordpress.com/
Raible	Matt	false	http://raibledesigns.com/
Coenraets	Christophe	false	http://coenraets.org/

The triangle behind the column header label shows how the column is sorted.

Lazy loading

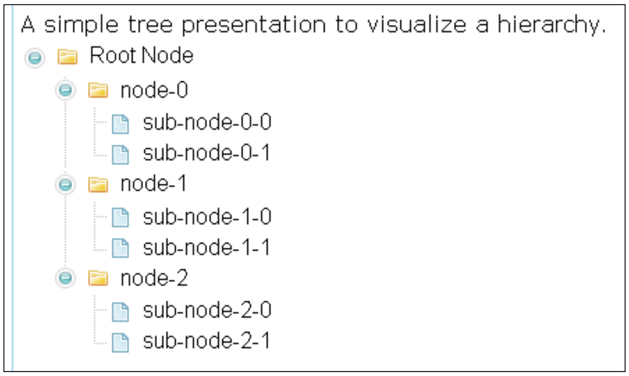
The use of the `dataPaginator` may create the impression that even the database access is done in pages. ICEfaces has no support for this by default. Actually, your backend delivers a complete list of objects and the `dataTable` component manages the paging locally inside the web container. This can become a scaling problem if you have to manage a lot of data and a user has unlimited access to it.

So you may have to implement a better strategy yourself. On the ICEfaces tutorial pages, you can study an example for this (`http://facestutorials.icefaces.org/tutorial/dataTable-JPA-tutorial.html`). If you are interested in a framework that is specialized in the JPA support for lazy loading implementations, have a look at the Crank framework (`http://krank.googlecode.com/`).

Download at WoweBook.Com

Tree

In classic web applications, trees were sometimes used for navigation. In modern web applications that use pulldown menus, trees are primarily used to visualize hierarchical relations of objects. Here is what the ICEcube example is generating. Have a look in the menu at **Presentation | Tree**:



A simple tree presentation to visualize a hierarchy.

```
Root Node
  node-0
    sub-node-0-0
    sub-node-0-1
  node-1
    sub-node-1-0
    sub-node-1-1
  node-2
    sub-node-2-0
    sub-node-2-1
```

The `<ice:tree>` tag shows a simple text output that describes a hierarchy. The icons are from the current skin and can be set individually. Unfortunately, the nodes cannot be set in the XHTML code, but have to be created via the backing bean.

The tree model in the previous image is inspired by the Swing programming model. This may be a help for a formal Swing developer. However, a lot of web developers will recognize an unnecessary complexity in it.

We'll have a look at the tag structure first. The ICEcube example page looks like this (`/src/main/webapp/icecube/presentation/tree.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name=»title»>
```

Download at WoweBook.Com

```
    #{icecube['application.menu.presentation.tree']}
  </ui:define>
  <ui:define name=»content»>
    #{icecube['application.menu.presentation.tree.text']}
    <ice:form>
      <ice:panelGrid>
        <ice:tree value="#{tree.model}" var="item"
          hideRootNode="false"
          hideNavigation="false"
          imageDir="#{iceFusionConsts.skinBase}/
            #{context.skin}/css-images/">
          <ice:treeNode>
            <f:facet name="icon">
              <ice:panelGroup style="display: inline" >
                <ice:graphicImage
                  value="#{item.userObject.icon}"/>
              </ice:panelGroup>
            </f:facet>
            <f:facet name="content">
              <ice:panelGroup style="display: inline" >
                <ice:outputText
                  value="#{item.userObject.text} "/>
              </ice:panelGroup>
            </f:facet>
          </ice:treeNode>
        </ice:tree>
      </ice:panelGrid>
    </ice:form>
  </ui:define>
</ui:composition>
</body>
</html>
```

The tag structure is similar to the data table definition. We have a `value` tag that is managing the model, and a `var` attribute for referencing single objects and attributes for the node presentation.

The model structure is a bit different and defines a tree with the `DefaultMutableTreeNode` nodes. These additionally have the `IceUserObject` objects for managing your data.

Download at WoweBook.Com

The creation of such a tree model looks like this (`/src/main/java/com/googlecode/icecube/presentation/Tree.java`):

```
package com.googlecode.icecube.presentation;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
public class Tree extends BackingBeanForm {
  private DefaultTreeModel model;
  public void init() {
    int ids = 0;
    String skin = consts.getSkinBase() + "/" +
      context.getSkin();
    // create root node with its children expanded
    DefaultMutableTreeNode rootTreeNode =
      new DefaultMutableTreeNode();
    TreeNodeUserObject rootObject =
      new TreeNodeUserObject(rootTreeNode,skin);
    rootObject.setId(ids++);
    rootObject.setText("Root Node");
    rootTreeNode.setUserObject(rootObject);
    model = new DefaultTreeModel(rootTreeNode);
    // add some child nodes
    for (int i = 0; i < 3; i++) {
      DefaultMutableTreeNode branchNode =
        new DefaultMutableTreeNode();
      TreeNodeUserObject branchObject =
        new TreeNodeUserObject(branchNode,skin);
      branchObject.setId(ids++);
      branchObject.setText("node-" + i);
      branchObject.setLeaf(false);
      branchNode.setUserObject(branchObject);
      rootTreeNode.add(branchNode);
      // add some more sub children
      for (int k = 0; k < 2; k++) {
        DefaultMutableTreeNode subBranchNode =
          new DefaultMutableTreeNode();
        TreeNodeUserObject subBranchObject =
          new TreeNodeUserObject(subBranchNode,skin);
        subBranchObject.setId(ids++);
        subBranchObject.setText("sub-node-" + i +
          "-" + k);
        subBranchObject.setLeaf(true);
        subBranchNode.setUserObject(subBranchObject);
```

Download at WoweBook.Com

```
          branchNode.add(subBranchNode);
        }
      }
    }
    public DefaultTreeModel getModel() {
      init();
      return model;
    }
  }
```

The `init()` method creates the structure using the related `DefaultMutableTreeNode` objects. Each of these nodes has a `TreeNodeUserObject` that manages concrete node data that we want to use for presentation. `TreeNodeUserObject` extends `IceUserObject` and can be found at `/src/main/java/com/googlecode/icecube/presentation/TreeNodeUserObject.java`:

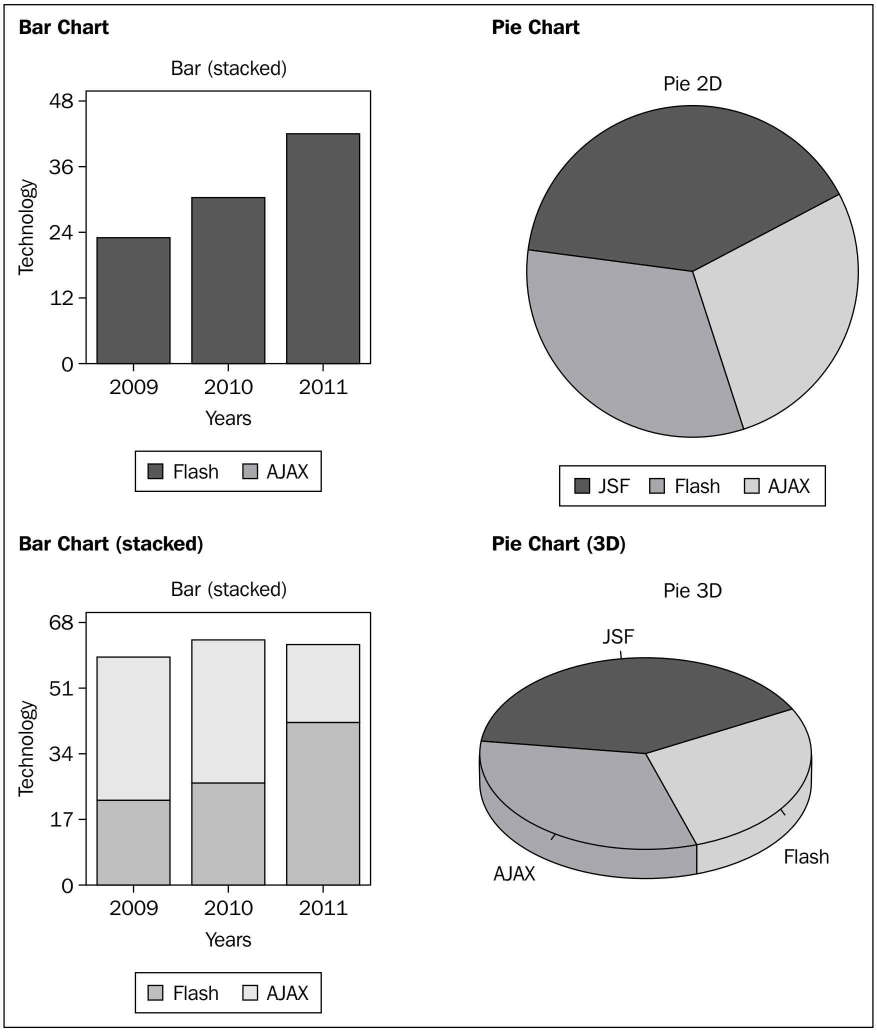
```
package com.googlecode.icecube.presentation;
import javax.swing.tree.DefaultMutableTreeNode;
import com.icesoft.faces.component.tree.IceUserObject;
public class TreeNodeUserObject extends IceUserObject {
  private int id;
  public TreeNodeUserObject(DefaultMutableTreeNode wrapper,
    String skin) {
    super(wrapper);
    this.init(skin);
  }
  protected void init(String skin) {
    this.setBranchExpandedIcon(skin +
      «/css-images/tree_folder_open.gif»);
    this.setBranchContractedIcon(skin +
      «/css-images/tree_folder_closed.gif»);
    this.setLeafIcon(skin +
      «/css-images/tree_document.gif»);
        this.setExpanded(true);
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
}
```

Download at WoweBook.Com

This extension defines an `id` attribute for every node and helps to set the corresponding skin icons automatically. If you need other node data to manage, you would put it into this definition. Such attributes are referenced like this: `#{item.userObject.attribute_name}`.

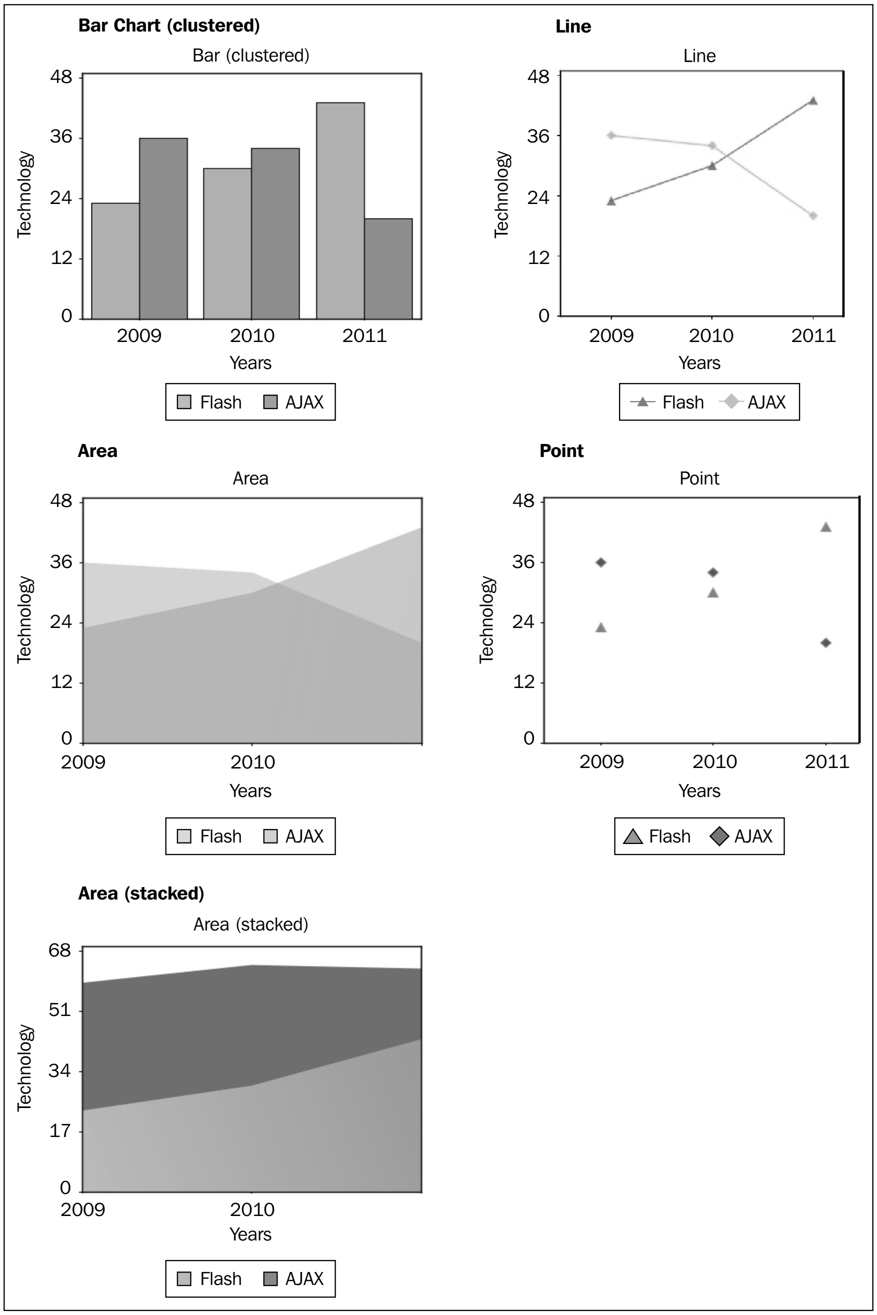
Chart

Charts help to visualize data. The ICEfaces framework offers a JSF-like implementation of the JCharts project (`http://jcharts.sourceforge.net/`). The first part of the ICEcube example page in the menu at **Presentation | Chart** looks like this:



Download at WoweBook.Com

The example page shows an example for every ICEfaces chart. The output is organized in two columns. The following screenshot shows the last part of the example page:



Download at WoweBook.Com

The corresponding code looks like this (`/src/main/webapp/icecube/presentation/chart.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion="http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.presentation.chart']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.presentation.chart.text']}
    <ice:form>
    <ice:panelGrid columns="2">
      <ice:panelGrid>
        <h3>Bar Chart</h3>
        <ice:outputChart type="bar"
          chartTitle="Bar"
          yaxisTitle="Technology"
          xaxisTitle="Years"
          xaxisLabels="2009, 2010, 2011"
          labels="Flash, AJAX"
          data="23, 30, 43: 36, 34, 20"
          colors="blue, green"/>
      </ice:panelGrid>
      <ice:panelGrid>
        <h3>Pie Chart</h3>
        <ice:outputChart type="pie2d"
          chartTitle="Pie 2D"
          labels="JSF, AJAX, Flash"
          data="43, 30, 27"
          colors="blue, green, cyan"/>
      </ice:panelGrid>
      <ice:panelGrid>
        <h3>Line</h3>
        <ice:outputChart type="line"
```

Download at WoweBook.Com

```
          chartTitle="Line"
          yaxisTitle="Technology"
          xaxisTitle="Years"
          xaxisLabels="2009, 2010, 2011"
          labels="Flash, AJAX"
          data="23, 30, 43: 36, 34, 20"/>
      </ice:panelGrid>
      <ice:panelGrid>
        <h3>Area</h3>
        <ice:outputChart type="area"
          chartTitle="Area"
          yaxisTitle="Technology"
          xaxisTitle="Years"
          xaxisLabels="2009, 2010, 2011"
          labels="Flash, AJAX"
          data="23, 30, 43: 36, 34, 20"/>
      </ice:panelGrid>
      <ice:panelGrid>
        <h3>Point</h3>
        <ice:outputChart type="point"
          chartTitle="Point"
          yaxisTitle="Technology"
          xaxisTitle="Years"
          xaxisLabels="2009, 2010, 2011"
          labels="Flash, AJAX"
          data="23, 30, 43: 36, 34, 20"/>
      </ice:panelGrid>
    </ice:panelGrid>
    </ice:form>
    </ui:define>
</ui:composition>
</body>
</html>
```

The code shows the use of attributes for the main chart types. So the attributes for the `bar` also work for the `barstacked` or `barclustered` type. The `<ice:outputChart>` tag is responsible for the rendering of all kinds of charts. It supports:

- Bar
- Bar (stacked)
- Bar (clustered)
- Pie
- Pie 3D

Download at WoweBook.Com

- Area
- Area (stacked)
- Line
- Point

The type attribute defines which kind of chart is rendered. The data to be processed is defined via the data attribute. Values are comma separated and the sets are separated by a colon. The example page uses the same data for all of the charts. The bar chart and all the pie charts use single sets. Although two sets are defined, the bar chart interprets only the first set.

The labels attribute lists the comma-separated names that are used for the sets defined in data. The same is done for defining a color for a set using the colors attribute. The first two charts have a colors attribute set and their colors are kept. All other charts create colors by accident each time the page is rendered. This is for demonstration purposes only. Such color combinations are seldom useful.

For labeling the chart, you can use the chartTitle attribute for a header title, the yaxisTitle attribute for the Y-axis, and the xaxisTitle attribute for the X-axis.

Google Maps

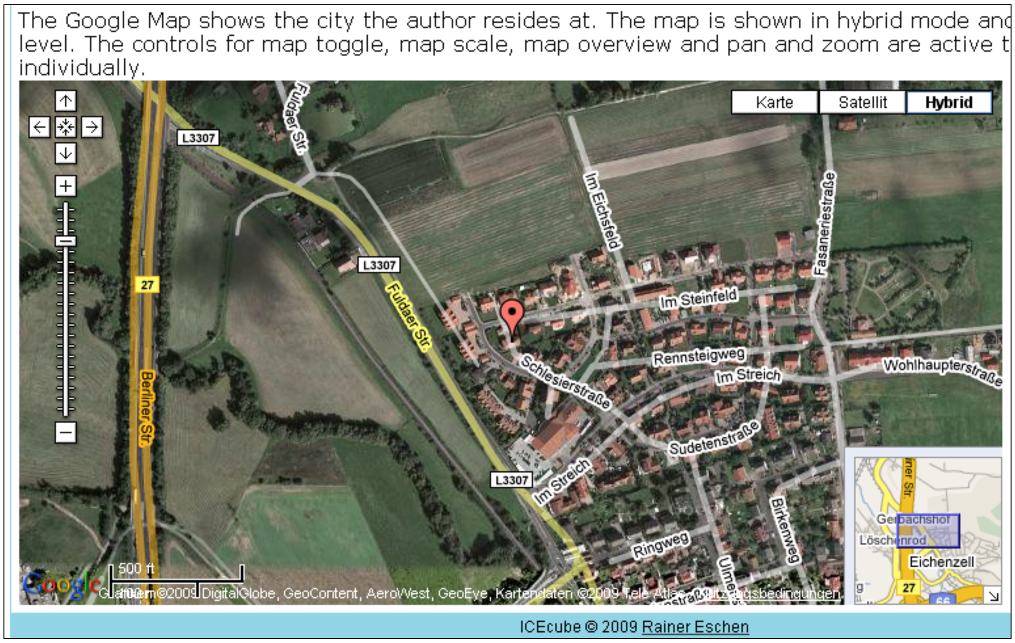
The ICEfaces framework supports the presentation of Google Maps right out of the box. For this, you do not need any JavaScript. There are several tags that help to present a Google Map in your web application. Here is the code for the ICEcube example page (/src/main/webapp/icecube/presentation/googleMap.xhtml):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name=»title»>
    #{icecube['application.menu.presentation.googleMap']}
  </ui:define>
```

Download at WoweBook.Com

```
    <ui:define name=»content»>
      #{icecube['application.menu.presentation.
        googleMap.text']}
      <ice:gMap latitude=»50.5001262» longitude=»9.6912457»
        zoomLevel=»16» type=»Hybrid»>
        <ice:gMapControl name=»GLargeMapControl»/>
        <ice:gMapControl name=»GScaleControl»/>
        <ice:gMapControl name=»GMapTypeControl»/>
        <ice:gMapControl name=»GOverviewMapControl»/>
        <ice:gMapMarker>
          <ice:gMapLatLng latitude=»50.5001262»
            longitude=»9.6912457»/>
        </ice:gMapMarker>
      </ice:gMap>
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

The `<ice:gMap>` tag defines the position, the zoom level, and the kind of map to show. It has four controls for manipulation and a marker to highlight the chosen position. The result looks like this (see ICEcube menu at **Presentation | Google Map**):


The Google Map shows the city the author resides at. The map is shown in hybrid mode and
level. The controls for map toggle, map scale, map overview and pan and zoom are active t
individually.

ICEcube © 2009 Rainer Eschen

Download at WoweBook.Com

The GLargeMapControl definition allows you to pan and zoom the map section. It initiates the control on the top left. The GScaleControl shows a map scale on the bottom left. The GMapTypeControl shows different buttons on the top right. These allow toggling between different map types. The GOverviewMapControl shown on the bottom right allows moving the map section in context to the surrounding area.

The gMapMarker tag allows us to set one or more visual markers. These are shown in red by default. Each of its gMapLatLng elements renders a visual marker through setting the longitude and latitude attributes. We have chosen the same position that the gMap tag uses. This allows you to visually highlight the chosen map section with a corresponding marker.

The ICEfaces framework allows customizing the presentation of Google Maps through a number of CSS stylesheets. The presentation above only uses one individual style definition that can be found in /src/main/webapp/icefusion/styles/icefusion/style.css:

```
.iceGmpMapTd div.gmap {
  width: 750px;
  height: 400px;
}
```

It defines the dimensions of the map to show.

License

If you integrate Google Maps into your application, you actually integrate a service you have to pay for. With your agreement, Google supplies you with a special key that has to be used when you use the service. Our example uses a key that is delivered by ICEsoft for testing purposes only.

The key is managed in the web.xml through the com.icesoft.faces.gmapKey context parameter. For ICEcube, the testing key can be found in /src/main/webapp/WEB-INF/web.xml:

```
<context-param>
  <param-name>com.icesoft.faces.gmapKey</param-name>
  <param-value>ABQIAAAADlu0ZiSTam64EKaCQr9eTRTOTuQNzJNXRlYR
    Lknj4cQ89tFfpxTEqxQnVWL4k55OPICgF5_SOZE06A</param-value>
</context-param>
```

Download at WoweBook.Com

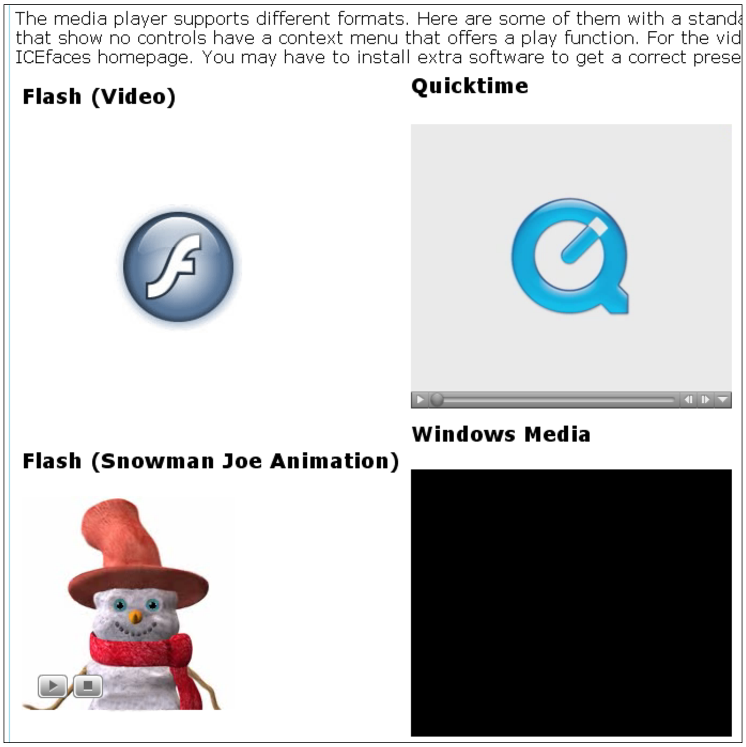
GMaps4JSF

If the use of Google Maps is critical to your project, you may also have a look at GMaps4JSF (`http://code.google.com/p/gmaps4jsf/`). The ICEcube example page for Google Maps also presents a GMaps4JSF example.

GMaps4JSF is more flexible to use. It allows defining events, for example. Chapter 10, *Push Technology*, will show this in more detail when we develop the ICEmapper game.

Media Player

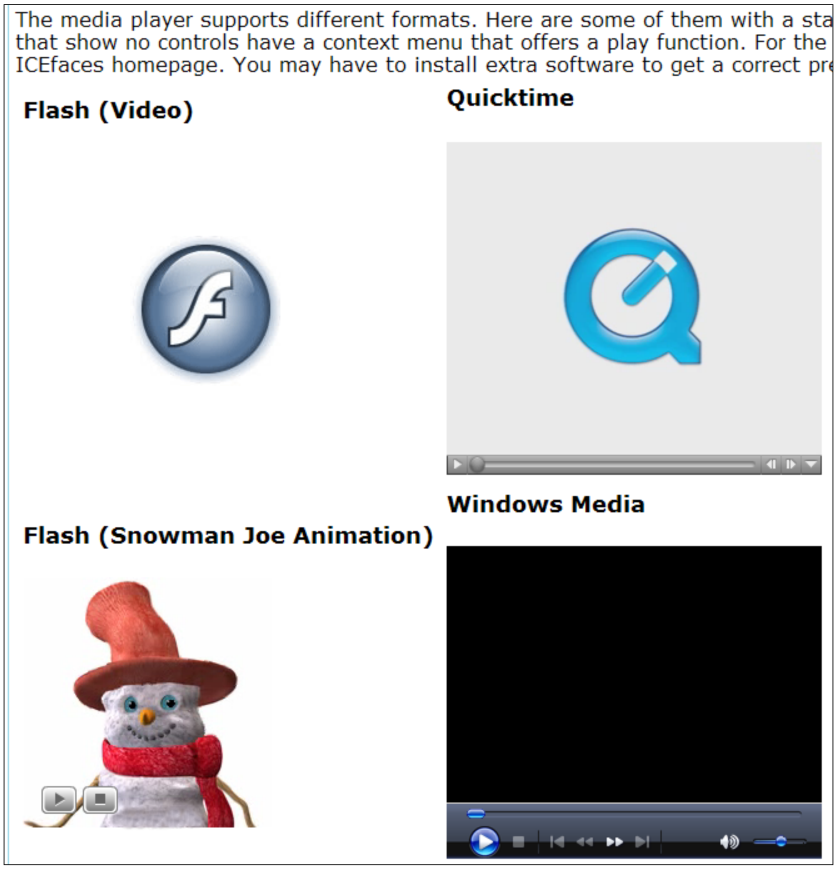
The `<ice:outputMedia>` tag allows you to present different multimedia formats inside a web application. The simplest use case is to play a video, whereas the most complex one is to present a Flash-based application. The initialized ICEcube example page found in the menu at **Presentation** | **Media Player** looks like this (for Firefox on Windows XP):



The media player supports different formats. Here are some of them with a stand
that show no controls have a context menu that offers a play function. For the vid
ICEfaces homepage. You may have to install extra software to get a correct prese

Download at WoweBook.Com

In Microsoft Internet Explorer, it looks like this:



The media player supports different formats. Here are some of them with a sta
that show no controls have a context menu that offers a play function. For the
ICEfaces homepage. You may have to install extra software to get a correct pr

Besides the visual difference after the page initialization, the players work fine. The page defines four players:

- A standard Flash video
- A standard Quicktime video
- A Flash animation
- A standard Windows Media Player

Download at WoweBook.Com

Players that show no controls have a context menu to start their content. All video players use a video from the ICEfaces component showcase that comes in different formats. So, they show the same video in different contexts. The Snowman Joe animation is an example of a more complex Flash presentation. It is based on a 3D animation that was rendered as Flash. If you have a look at it in the web browser, you will recognize that it is a combination of two animations:

- Subtle movement to present a lively character during inactivity
- A lip-sync speaking character during activity, telling something about a Xmas home page context

Such animations can be used in application contexts where the "human factor" can help to transport a message more efficiently. Online tutorials or online help systems are a good example of this.

To create the media players from the previous screenshot, the ICEcube example page looks like this (`/src/main/webapp/icecube/presentation/mediaPlayer.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.presentation.mediaPlayer']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.presentation.
      mediaPlayer.text']}
    <ice:panelGrid columns="2">
      <ice:panelGrid>
        <h3>Flash (Video)</h3>
        <ice:outputMedia player="flash"
          source="... /ICEfaces_Flash.swf"
          style="width:300px;height:250px;">
            <f:param name="play" value="false"/>
```

Download at WoweBook.Com

```
            <f:param name="menu" value="true"/>
          </ice:outputMedia>
        </ice:panelGrid>
        <ice:panelGrid>
          <h3>Quicktime</h3>
          <ice:outputMedia player="quicktime"
            source="... /ICEfaces_Quicktime.mov"
            style="width:300px;height:270px;">
              <f:param name="autoplay" value="false"/>
              <f:param name="controller" value="true"/>
          </ice:outputMedia>
        </ice:panelGrid>
        <ice:panelGrid>
          <h3>Flash (Snowman Joe Animation)</h3>
          <ice:outputMedia player="flash"
            source="... /SnowmanJoeHead_Data/FlashSwf.swf"
            style="width:200px;height:200px;">
              <f:param name="play" value="false"/>
              <f:param name="menu" value="true"/>
              <f:param name="flashvars"
                value="StandByVideo=
                  SnowmanJoeHeadStandby&amp;
                  BrowseVideo=SnowmanJoeHead" />
          </ice:outputMedia>
        </ice:panelGrid>
        <ice:panelGrid>
          <h3>Windows Media</h3>
          <ice:outputMedia player="windows"
            source="... /ICEfaces_Windows_Media.wmv"
            style="width:300px;height:250px;">
              <f:param name="autostart" value="0"/>
              <f:param name="showcontrols" value="1"/>
          </ice:outputMedia>
        </ice:panelGrid>
      </ice:panelGrid>
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

The `player` attribute of the `outputMedia` tag defines which player technology you want to use. The file to play is defined through the `source` attribute. The `style` attribute is used in our example to define the dimensions the player should render for the presentation.

Download at WoweBook.Com

Parameters that you would set with the original HTML embedding code are defined via the `<f:param>` tag. Besides the `play` or `controls` parameters that every player defines, the Snowman Joe animation also shows an example of a `flashvar` parameter definition. What is important with the parameter string is to write it XML-conform (`&amp`, instead of `&`).

Summary

The ICEfaces framework offers a wide range of components to present data. Besides the classic data table and tree presentations, it allows you to integrate state-of-the-art presentation and interaction, such as Google Maps. With the Media Player component, you already hold a tool in your hands that allows building the next generation of web browser applications. These applications will present their information based on multimedia. Video and Flash-based interaction will then be a central part of the data presentation.

We will continue our discussion of data management components in the next chapter. It discusses data creation components, with a focus on forms.

Download at WoweBook.Com

Download at WoweBook.Com

7 Components for Data Creation and Selection

This chapter has a focus on forms. We will take a deeper look at partial submit technology and how it can be used with certain form elements. Additionally, we will take a look at form validation, using dialogs for presentation. At the end of the chapter, we will look at advanced form elements that extend the standard JSF tags.

Forms

The standard implementation of JSF follows the request-response communication model. Fill in a form, submit it to the server, and the server generates a response page. The response may show the same form with extra information, such as validation hints. If you like to have a more desktop-like behavior, you have to keep the form and prevent it from reloading. Additionally, the event handling has to be more fine-grained so that you can react on single-field inputs.

Theoretically, you can surround every field with its own form tag. So, you get a single event handling for fields and the presentation will still show a complete form. But this is not practical because you are still bound to the standard JSF lifecycle. Moreover, it can be pretty complicated to manage these single forms as virtual through extra coding in the backing bean.

Download at WoweBook.Com

AJAX bridge and partial submit

The ICEfaces framework delivers two tools to manage forms in a desktop-like manner:

- AJAX bridge
- Partial submit

The AJAX bridge allows keeping the presentation without a reload of pages in the web browser. ICEfaces manages a server-side **Document Object Model** (**DOM**), that is indeed a copy of the browser's DOM. All changes are made on the server-side DOM first. Next, the AJAX bridge analyzes which parts have to be updated on the client side. Changes of the client-side DOM initiate a presentation update, but not a reload of a web page.

For a fine-grained management of form fields, the partial submit can be used. One example is to implement dependent edit fields to ease the input for the user. You may fill in a form for an address that shows other state selection lists after you have chosen a country. Another example is a language selector that immediately changes the user-interface language.

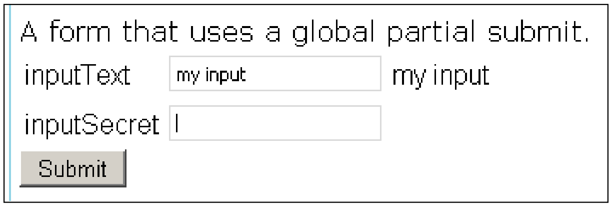
ICEfaces delivers enhanced JSF tags that allow you to set the `partialSubmit` attribute. You can use this for the complete form, so that all of the fields are managed in this way. The ICEcube sample page, `/src/main/webapp/icecube/creation/form.xhtml`, shows this:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.creation.form']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.creation.form.text']}
    <ice:form partialSubmit="true" >
      <ice:panelGrid columns="3">
```

Download at WoweBook.Com

```
          <ice:outputText value="#{icecube[
            'application.menu.creation.form
            .inputText']}" />
          <ice:inputText value=
            "#{form.inputText}" />
          <ice:outputText value=
            "#{form.inputText}" />
          <ice:outputText value="#{icecube[
            'application.menu.creation.form
            .inputSecret']}" />
          <ice:inputSecret value=
            "#{form.inputSecret}" />
          <ice:outputText value=
            "#{form.inputSecret}" />
        </ice:panelGrid>
        <ice:commandButton value="#{icecube[
          'application.menu.creation.form
          .commandButton']}"
          action="#{form.commandButton}" />
      </ice:form>
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

If you write something in the **inputText** field and move to the next field with a *Tab* key, the form Submit is processed in a special manner. The result can be found in the ICEcube menu at **Creation | Form** and looks like this:


A form that uses a global partial submit.

inputText	my input	my input
inputSecret		

After pressing the *Tab* key, the cursor is shown in the **inputSecret** field. You may not have clicked on the **Submit** button, but the form was processed like the **Submit** button was clicked. The proof for this is the text output behind the edit field.

Download at WoweBook.Com

Partial submit and the JSF lifecycle

Technically, we are still working inside the JSF lifecycle. But before you submit, ICEfaces prepares the processing. To circumvent the possible validation constraints, the `required` validation is turned off for all of the fields except the one responsible for submitting the form. So, it is a bit like a simulation of a completely filled out form. For filled fields, the standard JSF validation is still processed. After processing the JSF lifecycle, the validation attributes are restored.

A partial submit is a standard-compliant implementation that manipulates the actual state of a form during runtime. This allows you to have a single event processing per field.

Form field processing

Besides a global `partialSubmit` for a form, you can also have a more fine-grained behavior. For this, you omit the `partialSubmit` attribute for the `form` tag and use it with single fields instead (`/src/main/webapp/icecube/creation/textEntry.xhtml`):

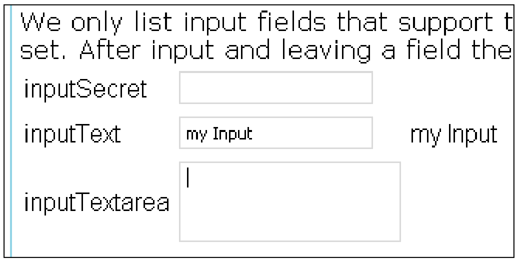
```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.creation.textEntry']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.creation.textEntry
    .text']}
    <ice:form>
      <ice:panelGrid columns="3">
        <ice:outputText value="#{icecube[
          'application.menu.creation.textEntry
          .inputSecret']}" />
```

Download at WoweBook.Com

```
          <ice:inputSecret value=
            "#{textEntry.inputSecret}"
            partialSubmit="true" />
          <ice:outputText value=
            "#{textEntry.inputSecret}" />
          <ice:outputText value=
            "#{icecube['application.menu
            .creation.textEntry.inputText']}" />
          <ice:inputText value=
            "#{textEntry.inputText}"
            partialSubmit="true" />
          <ice:outputText value=
            "#{textEntry.inputText}" />
          <ice:outputText value="#{icecube[
            'application.menu.creation.textEntry
            .inputTextarea']}" />
          <ice:inputTextarea value=
            "#{textEntry.inputTextarea}"
            partialSubmit="true" />
          <ice:outputText value=
            "#{textEntry.inputTextarea}" />
        </ice:panelGrid>
      </ice:form>
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

A submit is processed by leaving a single field that has a `partialSubmit` attribute set. All our fields have one, so we can skip the **Submit** button. Although it is technically possible to omit the button, you may add one so that you do not violate the design principle of *conformity with user expectations*.

The result can be found in the ICEcube menu at **Creation | Text Entry** and looks like this:


We only list input fields that support t
set. After input and leaving a field the

inputSecret		
inputText	my Input	my Input
inputTextarea		

Download at WoweBook.Com

You can click in any of the fields, put in some text, and leave the field to get a partial submitting. Your input is shown behind the field.

Partial submit supporting tags

The ICEfaces tags that offer the `partialSubmit` attribute can be separated into the following categories:

- Text entry
- Selection
- Click

Text entry

The following tags are a part of this category:

- `inputSecret`
- `inputText`
- `inputTextarea`

We have already shown an example of how to use these tags in the sample code of the previous section.

The `InputSecret` and `inputText` tags also offer the `action` and `actionListener` attributes, which can be used similarly to a `commandButton`. The ICEfaces release that was used during writing this book had some tweaks. So, you have to test if it is useful to have a field action or a field action listener in your project.

Selection

The following tags are a part of this category:

- `selectBooleanCheckbox`
- `selectManyCheckbox`
- `selectManyListbox`
- `selectManyMenu`
- `selectOneListbox`
- `selectOneMenu`
- `selectOneRadio`

Download at WoweBook.Com

The example code in the ICEcube sample page, `/src/main/webapp/icecube/creation/selection.xhtml`, looks like this:

```
<ice:panelGrid columns="3">
  <ice:outputText value="#{icecube[
    'application.menu.creation.selection
    .selectBooleanCheckbox']}" />
  <ice:form>
    <ice:selectBooleanCheckbox
      value="#{selection.selectBooleanCheckbox}"
      partialSubmit="true" />
  </ice:form>
  <ice:outputText
    value="#{selection.selectBooleanCheckbox}" />
  <ice:outputText value="#{icecube[
    'application.menu.creation.selection
    .selectManyCheckbox']}" />
  <ice:form>
    <ice:selectManyCheckbox
      value="#{selection.selectManyCheckbox}"
      partialSubmit="true" >
      <f:selectItems value="#{selection.items}"/>
    </ice:selectManyCheckbox>
  </ice:form>
  <ice:outputText value="#{selection.selectManyCheckbox}" />
  <ice:outputText value="#{icecube[
    'application.menu.creation.selection
    .selectManyListbox']}" />
  <ice:form>
    <ice:selectManyListbox
      value="#{selection.selectManyListbox}"
      partialSubmit="true">
      <f:selectItems value="#{selection.items}"/>
    </ice:selectManyListbox>
  </ice:form>
  <ice:outputText value="#{selection.selectManyListbox}" />
  <ice:outputText value="#{icecube[
    'application.menu.creation.selection
    .selectManyMenu']}" />
  <ice:form>
    <ice:selectManyMenu
      value="#{selection.selectManyMenu}"
      partialSubmit="true">
      <f:selectItems value="#{selection.items}"/>
```

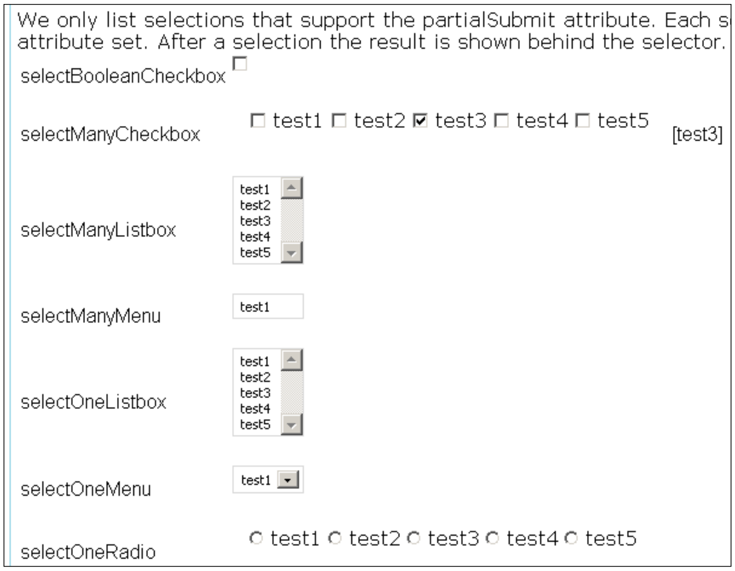
Download at WoweBook.Com

```
        </ice:selectManyMenu>
      </ice:form>
      <ice:outputText value="#{selection.selectManyMenu}" />
      <ice:outputText value="#{icecube[
        'application.menu.creation.selection
        .selectOneListbox']}" />
      <ice:form>
        <ice:selectOneListbox
          value="#{selection.selectOneListbox}"
          partialSubmit="true">
          <f:selectItems value="#{selection.items}"/>
        </ice:selectOneListbox>
      </ice:form>
      <ice:outputText value="#{selection.selectOneListbox}" />
      <ice:outputText value="#{icecube[
        'application.menu.creation.selection
        .selectOneMenu']}" />
      <ice:form>
        <ice:selectOneMenu value="#{selection.selectOneMenu}"
          partialSubmit="true">
          <f:selectItems value="#{selection.items}"/>
        </ice:selectOneMenu>
      </ice:form>
      <ice:outputText value="#{selection.selectOneMenu}" />
      <ice:outputText value="#{icecube[
        'application.menu.creation.selection
        .selectOneRadio']}" />
      <ice:form>
        <ice:selectOneRadio
          value="#{selection.selectOneRadio}"
          partialSubmit="true">
          <f:selectItems value="#{selection.items}"/>
        </ice:selectOneRadio>
      </ice:form>
      <ice:outputText value="#{selection.selectOneRadio}" />
    </ice:panelGrid>
```

You may wonder why we use separate `form` tags per field. This is to get a similar presentation like we have with the text entry example. With this implementation, only the current field is updated in the outputs. With a global `form`, all outputs would be updated even if a field has no explicit selection set. This looks a bit cluttered. In comparison to this, the text entry example always has empty strings. So, the output is always attractive.

Download at WoweBook.Com

The presentation of the code in the ICEcube menu at **Creation** | **Text Entry** looks like this:


We only list selections that support the partialSubmit attribute. Each s
attribute set. After a selection the result is shown behind the selector.

selectBooleanCheckbox ☐

selectManyCheckbox ☐ test1 ☐ test2 ☑ test3 ☐ test4 ☐ test5 [test3]

selectManyListbox test1 test2 test3 test4 test5

selectManyMenu test1

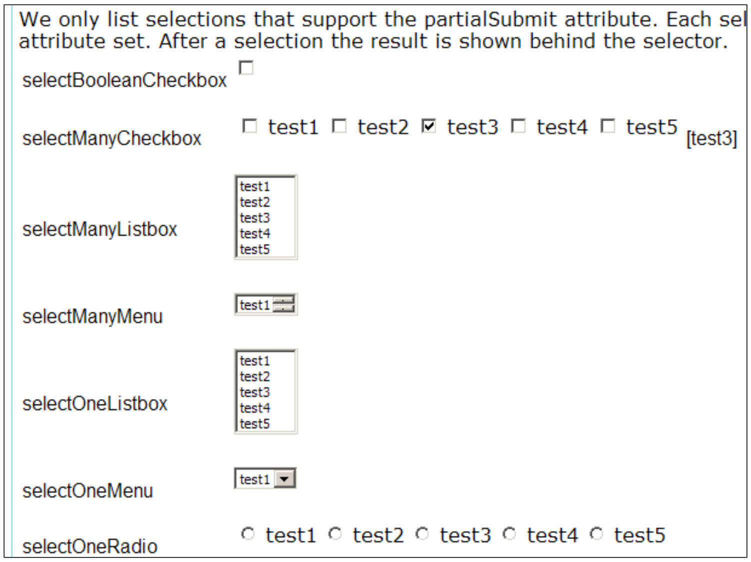
selectOneListbox test1 test2 test3 test4 test5

selectOneMenu test1

selectOneRadio ○ test1 ○ test2 ○ test3 ○ test4 ○ test5

The option **selectManyCheckbox** was selected for **test3**. The result is shown behind the field.

There is one tweak in the screenshot. The option **selectManyMenu** is missing the up and down arrows on the right. The screenshot shows a Firefox rendering. Here is a correct rendering with Internet Explorer:


We only list selections that support the partialSubmit attribute. Each sel
attribute set. After a selection the result is shown behind the selector.

selectBooleanCheckbox ☐

selectManyCheckbox ☐ test1 ☐ test2 ☑ test3 ☐ test4 ☐ test5 [test3]

selectManyListbox test1 test2 test3 test4 test5

selectManyMenu test1

selectOneListbox test1 test2 test3 test4 test5

selectOneMenu test1

selectOneRadio ○ test1 ○ test2 ○ test3 ○ test4 ○ test5

Download at WoweBook.Com

Click

The following tags are a part of this category:

- `commandButton`
- `commandLink`

Besides their behavior of submitting a complete form, it is possible to have a presentation inside a form that works independently. You can offer, for example, a button to explicitly change a state via the backing bean, which is then processed by the `form` submit.

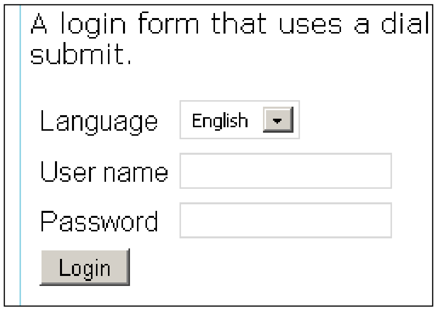
Validation with dialogs

JSF implements a very strong validation concept. But this is only useful if you follow the traditional presentation of forms. If you think about presenting forms in a desktop-like manner (for example, using dialog boxes for the validation messages), you are stuck.

Using our experiences with the ICEfusion dialog concept, we will now have a look at a standardized login form that is using validation dialogs to give a feedback to the user.

Login form component

ICEcube delivers a generic solution for a classic login form through the `<icefusion:login>` tag. Its presentation can be found in the ICEcube menu at **Creation | Validation Form**:


A login form that uses a dial
submit.

The login form allows you to change the current language. The selector uses a partial submit. So, the language change is separated from the login data processing. **User name** and **Password** are a part of a standard form processing that is started with a click on **Login**.

Download at WoweBook.Com

The `login` tag can be used like this:

```
<icefusion:login eventBean="#{action_handler_for_login}" />
```

The `eventBean` is a backing bean that defines the context in which the `login` tag is managed. ICEcube defines a backing bean for the example page that is used for the `eventBean` attribute. The implementation for the `login` tag looks like this (`/src/main/webapp/icefusion/taglibs/commons/form/login.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <!-- Manage eventBean as parameter for backing bean  -->
  <ice:dataTable style="visibility:hidden;"
    binding="#{login.parameters}" value="#{eventBean}"/>
  <ice:form>
    <ice:panelGrid>
      <ice:panelGrid columns="2">
        <ice:outputText value=
          "#{icefusion['application.login
          .language']}" />
        <ice:selectOneMenu value="#{login.locale}"
          partialSubmit="true">
          <f:selectItems
            value="#{login.locales}"/>
        </ice:selectOneMenu>
        <ice:outputText value=
          "#{icefusion['application.login
            .username']}" />
        <ice:inputText value="#{login.username}"/>
        <ice:outputText value=
          "#{icefusion['application.login
            .password']}" />
        <ice:inputSecret
          value="#{login.password}"/>
        <ice:commandButton value=
          "#{icefusion['application.login
            .button']}"
```

Download at WoweBook.Com

```
            action="#{login.login}" />
        </ice:panelGrid>
      </ice:panelGrid>
    </ice:form>
    <icefusion:validationDialog eventBean="#{login}"/>
  </ui:component>
  </body>
  </html>
```

The `login` tag is separated into three parts:

- The parameter handling for the `eventBean` reference that asks for the login input (the `dataTable` tag has no special meaning here)
- The form handling (`language`, `user name`, `password`, and `button`)
- The validation dialog handling

The `eventBean` reference has to implement the `ILogin` interface so that `username` and `password` can be set in the `eventBean`.

Here is the interface definition that can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/form/ILogin.java`:

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
public interface ILogin extends Serializable {
    public void setLoginUsername(String username);
    public void setLoginPassword(String password);
    public String loginAction();
}
```

When the login tag has processed the user input and no validation error is found, it sets both field values via the `setLoginUsername()` and `setLoginPassword()` setters, and finally calls `loginAction()`.

The backing bean of the `login` tag can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/form/Login.java` and looks like this. (The code is simplified and is missing the validation implementation, about which we will talk later.)

```
package com.googlecode.icefusion.ui.commons.form;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import javax.faces.model.SelectItem;
import com.icesoft.faces.component.ext.HtmlDataTable;
public class Login extends BackingBeanForm {
  HtmlDataTable parameters;
  private String username;
```

Download at WoweBook.Com

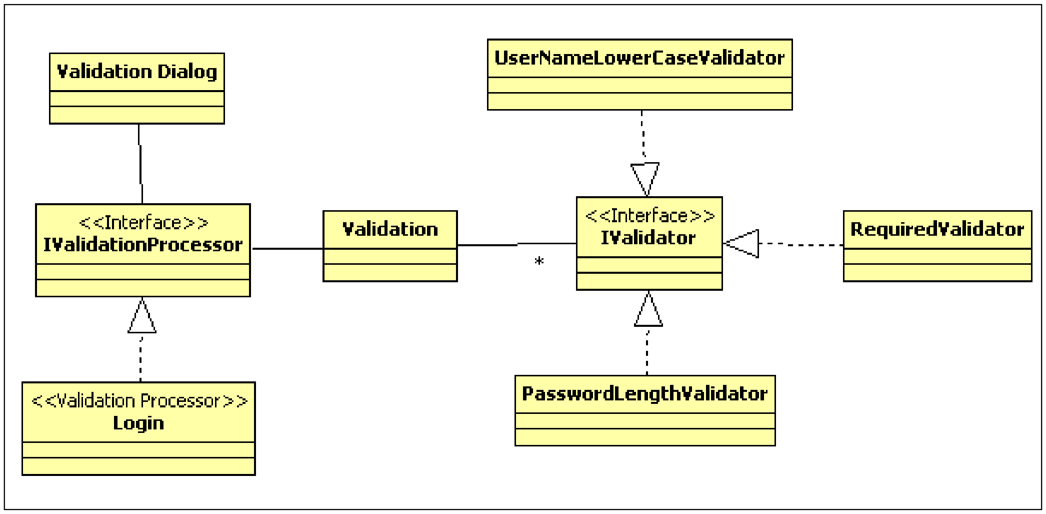
```
    private String password;
    public SelectItem[] getLocales() {
      SelectItem[] locales = new SelectItem[
        this.context.getSettings().getLocales().size()];
      int i = 0;
      for (Locale locale : this.context.getSettings()
        .getLocales()) {
        locales[i++] = new SelectItem(locale.getCode(),
          consts.getLocalized(locale.getLabel(),
          "icefusion"));
      }
      return locales;
    }
    public String login() {
      ((Ilogin)this.parameters.getValue())
        .setLoginUsername(
  this.getUsername());
            ((Ilogin)this.parameters.getValue())
        .setLoginPassword(
  this.getPassword());
            return ((Ilogin)this.parameters.getValue())
        .loginAction();
  }
    public String getLocale() {
      return this.context.getSettings().getLocale()
        .getCode();
    }
    public void setLocale(String locale) {
      this.context.getSettings().setLocale(locale);
    }
  }
```

The communication with the `eventBean` is done through `(ILogin)this.parameters.getValue()`. This is the `dataTable` binding definition in the XHTML code. The getter and setter for the current `locale` selection handle the persistence outside of the backing bean to keep the selection for the web application. ICEcube will change its menu language if you change the language in the login example.

Download at WoweBook.Com

Validation model

The following image shows how the validation for the `login` tag is constructed:



We have a **Validation Processor** that uses the **Validation** in context to configured **Validators**. Each **Validator** is following a certain rule to check a field value state. Which **Validator** is combined with which field value is defined in the **Validation Processor**. The **Validation Processor** gets a list of resource bundle IDs for multilingual output from the **Validation**. The IDs are processed by the **Validation Dialog**.

Login form with validation

To extend the `login` backing bean for validation, it implements the `IValidationProcessor` interface. The interface definition can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/validation/IValidationProcessor.java`:

```
package com.googlecode.icefusion.ui.commons.validation;
import java.io.Serializable;
public interface IValidationProcessor extends Serializable {
    public List<String> getValidationMessages();
    public void setValidationMessages(List<String> messages);
    public Boolean getValidationErrorStatus();
    public void setValidationField(String field);
    public String getValidationField();
    public String validationDialogButtonOk();
}
```

Download at WoweBook.Com

The **Validation** that is managed via the `login` backing bean delivers a list of resource bundle IDs if errors occur. The `<icefusion:validationDialog>` uses the `getValidationMessages()` for processing. Additionally, the dialog uses the `getValidationField()` to set a title that corresponds to the input field. The `validationDialogButtonOk()` manages the button action, and `getValidationErrorStatus()` manages the rendering of the `validationDialog`.

The validation-relevant parts of the `login` backing bean look like this (`/src/main/java/com/googlecode/icefusion/ui/commons/form/Login.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.validation
  .IValidationProcessor;
import com.googlecode.icefusion.ui.commons.validation
  .Validation;
import com.googlecode.icefusion.ui.commons.validation
  .validator.PasswordLengthValidator;
import com.googlecode.icefusion.ui.commons.validation
  .validator.RequiredValidator;
import com.googlecode.icefusion.ui.commons.validation
  .validator.UsernameLowerCaseValidator;
public class Login extends BackingBeanForm implements
  IValidationProcessor {
    private List<String> validationMessages =
      new ArrayList<String>();
    private String validationField;
    private Validation usernameValidation =
      new Validation(this, new RequiredValidator(),
      new UsernameLowerCaseValidator());
    private Validation passwordValidation =
      new Validation(this, new RequiredValidator(),
      new PasswordLengthValidator());
    public String login() {
        this.usernameValidation();
        this.passwordValidation();
        if (!this.getValidationErrorStatus()) {
            ((Ilogin)this.parameters.getValue())
            .setLoginUsername(this.getUsername());
            ((Ilogin)this.parameters.getValue())
            .setLoginPassword(this.getPassword());
            return ((Ilogin)this.parameters.getValue())
            .loginAction();
        }
```

Download at WoweBook.Com

```
            return null;
        }
        public void usernameValidation() {
            if (!this.getValidationErrorStatus()) {
                this.setValidationField("application.login
                .username");
                this.usernameValidation.validateValue(
                this.username);
            }
        }
        public void passwordValidation() {
            if (!this.getValidationErrorStatus()) {
                this.setValidationField("application.login
          .password");
                this.passwordValidation.validateValue(
          this.password);
            }
        }
    }
```

The `login()` method has some additional validation statements. The `eventBean loginAction()` method call is only done if the validation does not deliver errors. We define a validation context for every form field using the `Validation` class. This class can handle different Validators that implement rules for checking a field value. The backing bean defines Validators for `username` and `password` through `Validation` attributes.

The `Validation` constructor allows defining a list of `Validator` variables. Additionally, it gets a reference to a backing bean that implements the `IValidationProcessing` interface. This reference is used to deliver a list of resource bundle IDs for validation errors using the `setValidationMessages()`. The messages are a preparation for the Validation dialog.

Validation dialog

The **Validation dialog** is also using the `IValidationProcessing` interface to communicate with the `login` backing bean. Here is the XHTML code that can be found at `/src/main/webapp/icefusion/taglibs/commons/validation/validationDialog.xhtml`:

```
    <html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
```

Download at WoweBook.Com

```
    xmlns:ice="http://www.icesoft.com/icefaces/component"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
    <ice:form>
        <ice:panelPopup autoCentre="false" draggable="false"
            modal="true"
            rendered="#{eventBean.validationErrorStatus}"
            visible="#{eventBean.validationErrorStatus}">
            <f:facet name="header">
                <ice:panelGrid>
                    <ice:outputText value="#{icefusion[
                        eventBean.validationField]}"/>
                </ice:panelGrid>
            </f:facet>
            <f:facet name="body">
                <ice:panelGrid>
                    <ice:panelGrid columns="2" columnClasses=
                        "icePanelPopupImage
                              icePanelPopupText">
                        <ice:graphicImage url=
                            "#{iceFusionConsts.skinBase}/
                                #{context.skin}/images/
                                validationDialog.png" />
                        <ice:panelGrid>
                            <c:forEach var="message"
                                    items="#{eventBean
                                          .validationMessages}">
                                <ice:outputText value=
                                    "#{icefusion[message]}"/>
                                    <br/>
                            </c:forEach>
                        </ice:panelGrid>
                    </ice:panelGrid>
                    <ice:panelGrid columns="1" styleClass=
                        "icePanelPopupButtons">
                        <ice:commandButton value=
                            "#{icefusion['application
                                .validation
                                .message.button.ok']}"
                               action="#{eventBean
```

Download at WoweBook.Com

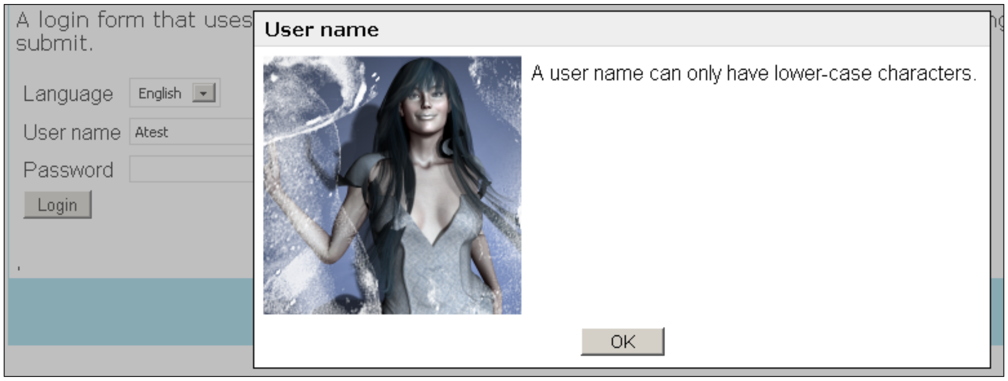
```
                                .validationDialogButtonOk}"
                            styleClass="icePanelPopupButton"/>
                      </ice:panelGrid>
                  </ice:panelGrid>
              </f:facet>
          </ice:panelPopup>
      </ice:form>
  </ui:component>
  </body>
  </html>
```

The `validationDialog` is similar to the other ICEfusion dialogs, but it has no backing bean. Instead it uses a bean reference, in our example the `login` backing bean, which is delivered via the `eventBean` attribute:

```
<icefusion:validationDialog eventBean=
  "#{my_validation_processor_bean}" />
```

The `validationDialog` uses the `getValidationMessages()` and `getValidationField()` methods to process the output. The `getValidationErrorStatus()` method defines if errors exist and if the dialog has to be shown. The `validationDialogButtonOk()` method is doing the action processing. In our example, the `login` backing bean uses it to clear all error messages.

An example of a validation error dialog looks like this (see ICEcube menu at **Creation | Validation Form**):



A user name can only have lower-case characters.

The **User name** input **Atest** has an uppercase character. Uppercase characters are not allowed, due to which the corresponding validation fails. The form also shows an empty **Password** field. The ICEfusion Validation Processor is able to show both errors in one dialog. But, the `login` backing bean checks if a validation error already exists with every validation. In this case, further processing of the validation is suppressed.

Download at WoweBook.Com

Validators

ICEfusion allows extending the number of Validators. All of the existing Validators can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/validation/validator`. If you want to implement your own Validator, you have to implement the `IValidator` interface (`/src/main/java/com/googlecode/icefusion/ui/commons/validation/IValidator.java`):

```
package com.googlecode.icefusion.ui.commons.validation;
import java.io.Serializable;
public interface IValidator extends Serializable {
  public Boolean validate();
  public Object getValue();
  public void setValue(Object value);
  public String getMessage();
  public void setMessage(String key);
}
```

The `UsernameLowerCaseValidator` shows how it is used (`/src/main/java/com/googlecode/icefusion/ui/commons/validation/validator/UsernameLowerCaseValidator.java`):

```
package com.googlecode.icefusion.ui.commons.validation.validator;
import com.googlecode.icefusion.ui.commons.validation.IValidator;
public class UsernameLowerCaseValidator implements IValidator {
    String message =
          "application.validation.validator.usernameLowerCase";
    Object value;
    public Boolean validate() {
        String username = (String)this.value;
        return username.equals(username.toLowerCase());
    }
}
```

Each `Validator` sets a resource bundle ID `message` that is used if the `validate()` method delivers a `false`. The `Validator` processes a `value` object that corresponds to the input of a form field the `Validator` is related to. The `validate()` implementation is pretty simple in this example. But it can also implement a matching with database values or results that are delivered by web service calls.

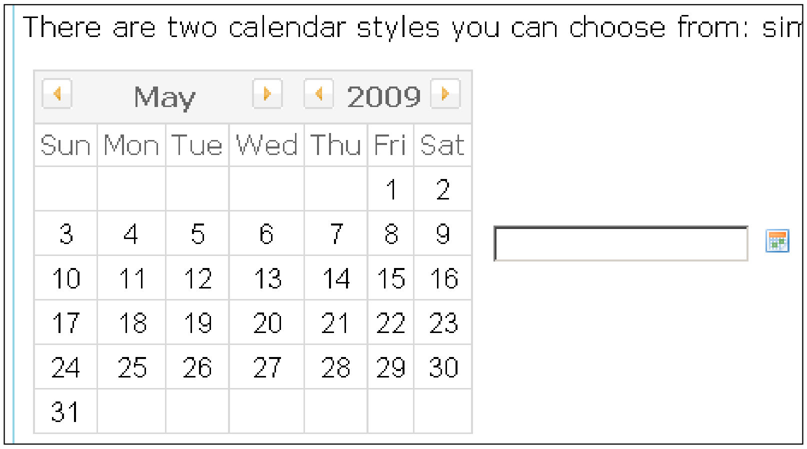
Download at WoweBook.Com

Calendar

Besides the enhanced standard JSF form tags, ICEfaces delivers advanced components that ease the input of data. For the selection of dates, it delivers the calendar component `<ice:selectInputDate>`. The calendar can be used in two modes:

- Simple
- Pop-up

The rendering of the calendar looks like this (see the ICEcube menu at **Creation** | **Calendar**):

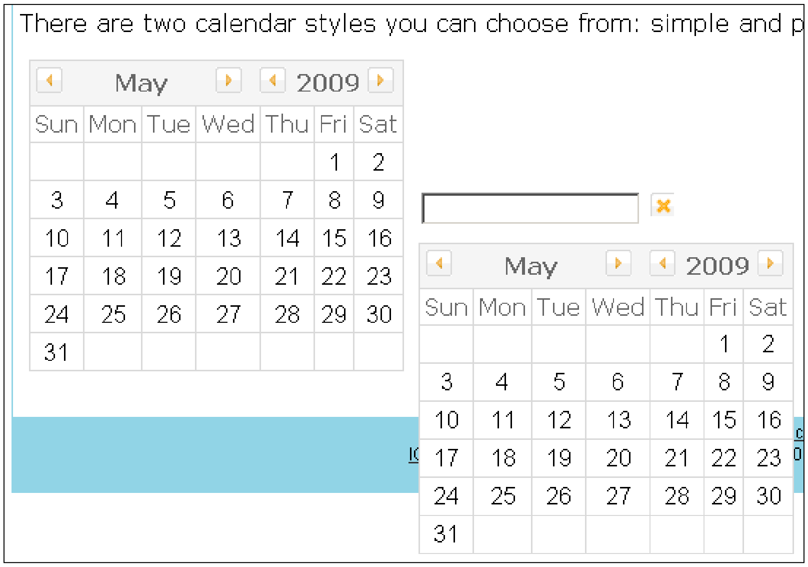


There are two calendar styles you can choose from: sin

May				2009		
Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

The simple mode is shown on the left side, whereas the pop-up mode is shown on the right side. When you click on the icon, you get this:

Download at WoweBook.Com


There are two calendar styles you can choose from: simple and p

Here is the corresponding code from the ICEcube sample page (`/src/main/webapp/icecube/creation/calendar.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.creation.calendar']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.creation.calendar.text']}
    <ice:form>
      <ice:panelGrid columns="2">
        <ice:panelGrid>
          <ice:selectInputDate
            value="#{calendar.date}" />
```

Download at WoweBook.Com

```
              <ice:outputText value="#{calendar.date}" />
            </ice:panelGrid>
            <ice:panelGrid>
              <ice:selectInputDate
                value="#{calendar.datePopup}"
                renderAsPopup="true" />
              <ice:outputText
                value="#{calendar.datePopup}" />
            </ice:panelGrid>
          </ice:panelGrid>
        </ice:form>
        </ui:define>
    </ui:composition>
    </body>
    </html>
```

The different presentations are controlled by the use of the `renderAsPopup` attribute.

Rich text editor

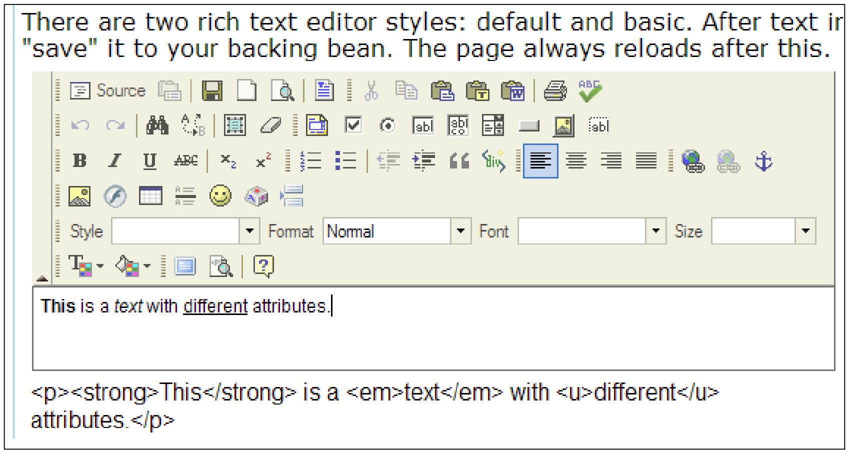
For the web-browser-based input of formatted text, there are different solutions available in the open source community. A lot of them are implemented in JavaScript. ICEfaces reuses the FCKeditor (`http://www.fckeditor.net/`) implementation (FCKeditor has now been replaced by the CKEditor which inherits the quality and strong features that were available with FCKeditor). The implementation is wrapped so that it can be used in the JSF context.

The rich text editor component `<ice:inputRichText>` offers different presentation modes:

- Standard icon set (the `default` mode)
- Reduced icon set (the `basic` mode)

The standard toolbar presentation can be found in the ICEcube menu at **Creation | Rich Text Editor**:

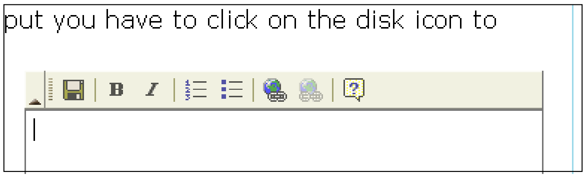
Download at WoweBook.Com


There are two rich text editor styles: default and basic. After text ir
"save" it to your backing bean. The page always reloads after this.

This is a text with different attributes.

<p><strong>This</strong> is a <em>text</em> with <u>different</u> attributes.</p>

The reduced toolbar looks like this:


put you have to click on the disk icon to

The corresponding code of the ICEcube sample page looks like this (`/src/main/webapp/icecube/creation/richTextEditor.xhtml`):

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
```

Download at WoweBook.Com

```
      #{icecube['application.menu.creation
        .richTextEditor']}
    </ui:define>
    <ui:define name="content">
      #{icecube['application.menu.creation.richTextEditor
        .text']}
      <ice:panelGrid columns="2" style="width: 100%">
        <ice:form>
          <ice:panelGrid>
            <ice:inputRichText value=
              "#{richTextEditor.text}"
              toolbar="Default"
              language="#{context.locale}"
              width="550" height="300"/>
            <ice:outputText value=
              "#{richTextEditor.text}" />
          </ice:panelGrid>
        </ice:form>
        <ice:form>
          <ice:panelGrid>
            <ice:inputRichText value=
              "#{richTextEditor.textBasic}"
              toolbar="Basic"
              language="#{context.locale}"
              width="350" height="300"/>
            <ice:outputText value=
              "#{richTextEditor.textBasic}" />
          </ice:panelGrid>
        </ice:form>
      </ice:panelGrid>
    </ui:define>
  </ui:composition>
  </body>
  </html>
```

The main difference between both presentations is the use of the `toolbar` attribute. The `width` and `height` attributes allow a fixed size presentation of a rich text editor.

Each editor shows a **Save** icon that allows setting the editor input in the backing bean. ICEcube has output tags defined that show the result after saving. The first rich text editor screenshot shows that you get an `HTML` notation for text attributes you use in the editor.

Download at WoweBook.Com

Summary

This chapter has shown that it is possible to implement desktop-like behavior for forms. Using the AJAX bridge, partial submit, and a re-implemented validation is a good base to implement even more complex forms. The calendar and the rich text editor are additional tools to implement desktop application behavior in the web browser.

All important ICEfaces components are discussed. The next chapter will add everything missing in ICEcube to get a full-blown web application that follows the desktop metaphor.

Download at WoweBook.Com

Download at WoweBook.Com

8
User Interface Customization

The most important difference between an informational web site and a web application is the behavior. A web site is almost static, whereas a web application is non-sequential and dynamic. An application allows the user to influence or customize the behavior. This chapter will describe a model to show how this can be done for web applications using ICEfaces. We will have a detailed look at the implementation of a multilingual and skinnable presentation. A skin is a set of images, colors, and layouts that can be changed on the fly.

The individualization of a web application in our model can be described as follows:

- **Administration**: All settings an administrator is managing
- **Customization**: A subset of the settings that is changeable by the user

Administration

Complex web applications have their own administration. It allows creating and managing users, user groups, and other entities that are necessary for a user to produce useful working results. In this context, we can consider a web application as a collection of services. The administration defines which services are allowed to be used by which user.

Flexible administration concepts are often designed to be complex to ensure that a standard user does not get in touch with the administration directly. Normally, an administrator predefines all necessary services a user is allowed to use. The standard web application behavior, for example, of the user interface is predefined by the administrator, too.

Download at WoweBook.Com

But not all presets are useful in any case. Normally, the parameters of a preset are chosen so that the user can start working with a web application immediately. We can expect that not all users will be able to follow what an administrator assumes as useful. The users will have a demand for a more individualized behavior of the web application. In short, a user will want to customize our application.

Customization

If a web application allows you to customize its behavior, you have to strike a balance between flexibility and security. On one hand, the user wants to have features that allow him to individualize the application, make it more comfortable, and create optimal working results. On the other hand, you have to keep an eye on its security, reliability, and maintenance. Customization features can influence your service costs because of the additional support that you have to deliver.

There are a lot of possibilities in implementing customization features for a user. Some are common for all web applications, such as a password changer, whereas others strongly depend on the workflow that the web application is a part of.

We will have a look at some of the web application features in this chapter, such as:

- Password
- Username
- Units
- Number format
- Language
- Skinning

Language and skinning will be discussed in detail in this chapter. For this, we will have a look at the ICEcube and ICEfusion implementation. For the other features, you will get some tips that you have to keep in mind when you implement them on your own.

Password

If we take a look at what is common for customization features today, we will find the password changer on top of the list. It allows changing the password whenever the user wants. This is also the most relevant security issue.

Even if your backend has a strong security technology in use, it can suffer from a weak password selection immediately. To keep up the security quality implemented by the administration, you can add a validation that checks, for example:

Download at WoweBook.Com

- The password length
- The usage of numbers, punctuation, and a mixture of upper and lowercase letters
- If the password can be guessed by comparing the hashed value with a library of codes based on natural language and common passwords

Here, it is important to note that there is no administrator with a certain experience to care for security besides the user. Instead, the web application has to do this job on the fly. It has to offer a procedure that guarantees the quality even if the user is a beginner in computer usage.

We will not have a deeper look at the security issues because a concrete implementation heavily depends on the technology you use to secure your web application.

Username

If you take the role of an administrator, you can expect that users will ask you to change their username. Although this can create extra efforts, it improves the usability for a single user if you support this.

To relieve the administrator, it can be useful to allow the user to change the username as a part of the self-administration. This implementation would be similar to the password management. However, your administration model has to be designed for it. If your model does not define any dependency to the concrete username, this should be possible. Normally, this is the case if you use the username only to calculate the login result that tells the system if the user is allowed to use the application.

Email address

You often find systems that use the email address of a user for the username. This helps the administrator to get unique usernames. But the spam problem compels users to change their email address more often these days. It is pretty annoying for a user to be forced to use an old email address after he has changed his preferred one. So, if you follow the idea of using the email address as a username, your administration model has to allow the changing of the username in any case.

Keep in mind that the current email address may already be invalid when the user tries to set a new one for the username. So, it is a good idea to ask which email address should be used for the necessary validation before the username can be changed permanently.

Download at WoweBook.Com

Units

Web applications are often written for target groups with different cultural backgrounds. If your application is handling numbers with certain units, it can be useful to offer a conversion. The most convenient way is to normalize all values to the metric system (`http://en.wikipedia.org/wiki/Metric_system`) before saving them to the database.

The customized target units are used to define the factor that is necessary to calculate the presentation value. If you are not able to normalize the values, you have to save the source unit. The calculation will be more complicated for this.

For a concrete implementation, there are a lot of unit combinations that have to be considered. Although the metric system is the official standard, not all countries use it. Even in countries that use it, you have cases where the unit that was used before starting with the metric system is still preferred. For your own implementation, you may have a look at `http://www.onlineconversion.com/` to get inspired.

Number format

Although a language also defines how the number format is rendered, it can be useful to differ from it. Maybe your web application does not offer the preferred language, or the user explicitly wants a different language and number format.

For a simple implementation, your application may offer an exchange between a dot and a comma for both separators. This works for most Western countries. However, Switzerland uses an apostrophe for the thousand separator (`http://en.wikipedia.org/wiki/Decimal_separator`). If you have a look at India, for example, you need a comma for the thousand separator and the decimal separator. A good idea for a worldwide support is to offer the possible separator characters in two lists—one for the thousand and another for the decimal separator, and let the user choose the combination he/she needs.

Language

Most web applications these days are used in multilingual contexts. It makes no difference from which location of the world an application is used. So, the application has to offer a language selection by default.

This customization feature influences the user acceptance. Besides the possibility of changing the language, the number of offered languages is pretty important. It should correspond to the target group your project has defined for the web application.

Download at WoweBook.Com

Skinning

If your target group has a demand for a more individualized presentation of the user interface to follow a corporate design, we need a skinning feature. ICEfaces has a built-in mechanism to use this by default. Each component supports one or more CSS classes and may also allow individual images that can be changed during runtime.

Language management

The multilingual support we use in ICEcube and ICEfusion is based on the standards that come with the JVM and JSF for locale management. We use resource bundles that can be found at `/src/main/resources/icefusion` (used by the Facelets components) and `/src/main/resources/icecube` (used by the examples). So, the default texts for the configuration page that you can open via **Extra | Settings** in the ICEcube menu can be found in the `/src/main/resources/icefusion/icefusion.properties` file:

```
application.menu.extra=Extra
application.menu.extra.settings=Settings
application.menu.extra.settings.text=Choose a new entry in
  the combobox to change the presentation.
```

For a complete definition of the locale management, JSF contains something like this in the `/src/main/webapp/WEB-INF/face-config.xml` file:

```
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>de</supported-locale>
</locale-config>
```

This defines that **English** is the default language that is used even when the user has a web browser that uses a language that the web application does not know. If you have a look inside the folders, you will also find an `icefusion_de.properties` or `icefusion_en.properties` file. The latter allows you to explicitly define an **English** variant that is used for users who demand **English** language in their web browsers. Although it's a more theoretical example with ICEcube, it can be useful if you use full-blown locale IDs such as `en_US` with a UK-based default implementation. ICEcube uses **US English** by default, though.

Download at WoweBook.Com

Multilingual page template

In addition to the JVM and JSF definitions, contains tags for the support of multilingual presentations. The page template can be found at `/src/main/webapp/icefusion/taglibs/commons/page.xhtml`:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:icefusion=
      "http://icefusion.googlecode.com/icefusion">
<f:view locale="#{context.locale}">
<ice:loadBundle basename="icefusion.icefusion"
  var="icefusion"/>
<ice:loadBundle basename="icecube.icecube" var="icecube"/>
<head>
</head>
<body>
</body>
</f:view>
</html>
```

The `<ice:loadBundle>` tag allows us to define a variable per bundle that can be used inside the XHTML files. ICEcube has its own resource bundle that is used via the `icecube` variable. The ICEfusion components use the `icefusion` variable.

Before ICEfaces 1.8, we had to use `<f:loadBundle>`. The ICEfaces implementation fixes some caching problems so that resource bundles are unloaded when a dynamic locale change is recognized.

For an example of how the `icefusion` variable is used, we have a look at the configuration page, `/src/main/webapp/icefusion/menu/extra/settings.xhtml`. This is delivered by ICEfusion using the `icefusion` variable:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
```

Download at WoweBook.Com

```
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
  "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icefusion['application.menu.extra.settings']}
  </ui:define>
  <ui:define name="content">
    #{icefusion['application.menu.extra.settings.text']}
    <ice:form>
        <ice:panelGrid>
            <ice:panelGrid columns="2">
                <ice:outputText value=
            "#{icefusion[
              'application.settings.language']}" />
                <ice:selectOneMenu value=
              "#{settings.locale}">
                    <f:selectItems value=
                "#{settings.locales}"/>
                </ice:selectOneMenu>
                <ice:outputText value=
              "#{icefusion[
                'application.settings.skin']}" />
                <ice:selectOneMenu value="#{settings.skin}">
                    <f:selectItems value=
                "#{settings.skins}"/>
                </ice:selectOneMenu>
                <ice:commandButton value=
              "#{icefusion[
                'application.settings.button']}"
                    action="#{settings.change}" />
            </ice:panelGrid>
        </ice:panelGrid>
    </ice:form>
   </ui:define>
</ui:composition>
</body>
</html>
```

Download at WoweBook.Com

The Expression Language term to reference a text in a resource bundle uses the bundle variable we have defined in the page template and the resource bundle ID of the text we want rendered:

```
#{bundle_variable['resource_bundle_id']}
```

The page example shows such definitions as well without using an explicit output tag. Facelets allows defining a text output without using a special tag. Take a look at the following code:

```
#{icefusion['application.menu.extra.settings']}
```

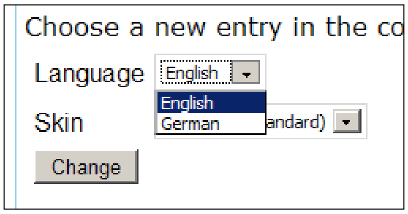
The previous line of code corresponds to the following code:

```
<ice:outputText value=
  "#{icefusion['application.menu.extra.settings']}" />
```

Both deliver the output, **Settings**.

Language selector

The implementation of the language selector looks like this (see **Extra** | **Settings** in the ICEcube menu):



It shows a list of languages that ICEcube and ICEfusion support, namely **English** and **German**. We have a `/src/main/java/com/googlecode/icefusion/ui/commons/extra/Settings.java` backing bean for the settings page that delivers the list entries. Here are the important parts of the locale management:

```
package com.googlecode.icefusion.ui.commons.extra;
import javax.faces.model.SelectItem;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.constant.Context;
import com.googlecode.icefusion.ui.commons.constant
  .ICEfusionConsts;
import com.googlecode.icefusion.ui.commons.constant.Locale;
public class Settings extends BackingBeanForm {
    public SelectItem[] getLocales() {
```

Download at WoweBook.Com

```
            SelectItem[] locales = new SelectItem[
            this.context.getLocales().size()];
            int i = 0;
            for (Locale locale : this.context.getLocales()) {
                locales[i++] =
              new SelectItem(locale.getCode(),
              consts.getLocalized(locale.getLabel(),
                "icefusion"));
            }
            return locales;
        }
        public String getLocale() {
            return this.context.getLocaleCode();
        }
        public void setLocale(String locale) {
            this.context.setLocale(locale);
        }
        public String change() {
            return null;
        }
    }
```

The `getLocale()` function manages the current selection of the languages list, whereas `getLocales()` delivers the list. The backing bean uses the `Context` bean to manage all of the user-specific settings in a session.

Here is the code for `/src/main/java/com/googlecode/icefusion/ui/commons/constant/Context.java` showing the locale management parts:

```
    package com.googlecode.icefusion.ui.commons.constant;
    import com.googlecode.icefusion.ui.commons.BackingBeanForm;
    public class Context extends BackingBeanForm {
        Settings settings = new Settings();
        public String switchToEn() {
            this.getSettings().setLocale("en");
            return null;
        }
        public String switchToDe() {
            this.getSettings().setLocale("de");
            return null;
        }
        public java.util.Locale getLocale() {
            return this.getSettings().getLocale().getLocale();
        }
        public String getLocaleCode() {
```

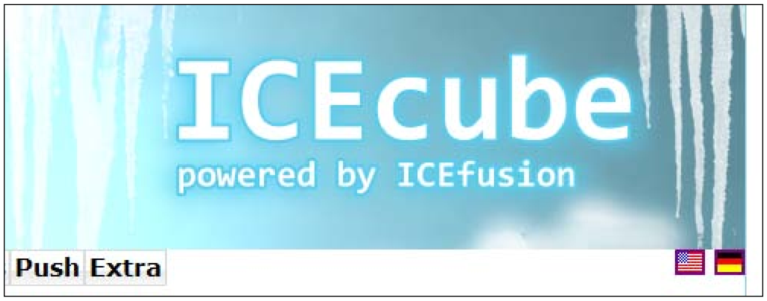
Download at WoweBook.Com

```
            return this.getSettings().getLocale().getCode();
        }
        public List<Locale> getLocales() {
            return this.getSettings().getLocales();
        }
        public void setLocale(String code) {
            this.getSettings().setLocale(code);
        }
    }
```

The `settings` reference is used for the persistence.

Global language switcher

To change the language more comfortably, modern web applications present a component near the pull-down menu. Often, the languages are shown as flags so that a suitable language can be visually chosen even if the current language is unknown, as shown in the next image:



The flags are implemented in a separate ICEfusion component through the `/src/main/webapp/icefusion/taglibs/commons/menuIcons.xhtml` file, as shown in the following code:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <div class="menuIcons">
```

Download at WoweBook.Com

```
      <ice:form>
        <ice:panelGrid columns="2">
                <ice:commandLink action=
                  "#{menuIcons.switchToEn}">
                      <ice:graphicImage url=
                        "#{iceFusionConsts.skinBase}/
                        #{menuIcons.skin}/
                        images/locale/en.png" />
                      </ice:commandLink>
                <ice:commandLink action=
                        "#{menuIcons.switchToDe}">
                      <ice:graphicImage url=
                       "#{iceFusionConsts.skinBase}/
                        #{menuIcons.skin}/
                        images/locale/de.png" />
                </ice:commandLink>
            </ice:panelGrid>
      </ice:form>
    </div>
  </ui:component>
  </body>
  </html>
```

The backing bean for the flag switching can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/navigation/MenuIcons.java`, as shown in the following code:

```
package com.googlecode.icefusion.ui.commons.navigation;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.constant.Context;
public class MenuIcons extends BackingBeanForm {
    public String switchToEn() {
        this.context.setLocale("en");
        return null;
    }
    public String switchToDe() {
        this.context.setLocale("de");
        return null;
    }
    public String getSkin() {
        return this.context.getSkin();
    }
}
```

Download at WoweBook.Com

Each flag has its own method that activates the corresponding language. The flag images are managed via skinning.

Skin management

The skinning we use in ICEcube is based on the CSS support that is delivered with the ICEfaces components. ICEfaces allows influencing the component presentation, for example, by changing the CSS definitions for:

- Images
- Colors
- Fonts
- Positions
- Sizes

Some ICEfaces components also use special tag attributes to influence the presentation.

The ICEfaces sources deliver different skin definitions that you can use as a base for your own implementation. The standard ICEfusion skins are based on the ICEfaces **Rime** skin. You can have a look at it in the component showcase at `http://component-showcase.icefaces.org`. The first combobox on the left selects one of the standard skins, and should show **Rime** by default.

ICEcube reuses the ICEfusion skinning by reusing the ICEfusion page template and its dependent files.

Skin folders

ICEfusion establishes an extended skin management in comparison to the skins you get with the ICEfaces sources. The design allows separating your own extensions, so that you can update the ICEfaces standard skin with future ICEfaces releases. A simple copy and paste of the files is possible without affecting your extensions.

The skins that ICEfusion is using can be found at `/src/main/webapp/icefusion/styles/`. The skin folder names are used as IDs inside the skin management. During the presentation of pages or components, a calculation of file paths is done depending on the current skin ID. The most important part of this is the calculation of CSS files that are referenced through the page template.

Download at WoweBook.Com

Skinnable page template

The global definitions that are used for skinning can be found in the page template `/src/main/webapp/icefusion/taglibs/commons/page.xhtml`. The template looks as shown in the following:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:icefusion=
      "http://icefusion.googlecode.com/icefusion">
<f:view locale="#{context.locale}">
<ice:loadBundle basename="icefusion.icefusion"
  var="icefusion"/>
<f:loadBundle basename="icecube.icecube" var="icecube"/>
<head>
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/page.css" />
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/icefaces.css" />
  <ice:outputStyle href="#{iceFusionConsts.skinBase}/
    #{context.skin}/style.css" />
  <script type="text/javascript"
    src="#{iceFusionConsts.contextPath}
    #{iceFusionConsts.scriptBase}/connectionStatus.js" >
  </script>
  <script type="text/javascript"
    src="#{iceFusionConsts.contextPath}
    #{iceFusionConsts.scriptBase}/icefusion.js" />
  <link rel="shortcut icon"
    href="#{iceFusionConsts.contextPath}
      #{iceFusionConsts.skinBase}/#{context.skin}
      /images/page.ico"/>
  <title>#{iceFusionConsts.application}
    #{iceFusionConsts.release} -
    <ui:insert name="title">
      This page has no title.
    </ui:insert>
  </title>
</head>
<body>
```

Download at WoweBook.Com

```
</body>
</f:view>
</html>
```

There are three CSS files that are used to realize the ICEfusion skinning:

- `icefaces.css`
- `page.css`
- `style.css`

The `icefaces.css` file is an exact copy of the **Rime** skin stylesheet file that is delivered with the ICEfaces sources. It was renamed for a better self-description.

The structure of the ICEfusion skin folder allows you to keep this file in its original format. Any change that is necessary for the original CSS classes is done in the `style.css`. So, if you update the ICEfaces release in the future, you can take the original **Rime** stylesheet file that comes with the release sources and overwrite the existing `icefaces.css`.

If you do not change the files in the `/css-images` folder that delivers the **Rime** standard images, this can also be replaced. But if your skin design needs to adapt to the standard images in order to leave the `icefaces.css` untouched, you have to compare images file-by-file and add only the additional images that come with the new ICEfaces release.

Theoretically, it is possible to define CSS classes in the `style.css` and reference a different folder for the adapted images that these classes use. But the ICEfaces component design is not as homogeneous as you would expect it to be. There is a mixture of direct references to images versus references to CSS classes with image definitions. Some components also need an explicit path to a skin folder set. So, expect tweaks when you think about skinning. If you do not have time for a trial-and-error journey, replace the images in `/css-images` with your changes and make a file-by-file comparison with updates of every ICEfaces release.

The `page.css` file is used to define the structure of the web application. It also defines the basic fonts and colors to use. So, if you want to change the header width or height (for example), you have to take a look here.

Although `page.css` tries to be the global definition file for an ICEfusion skin, it is not in any case. The `icefusion.css` file has its own ideas of how global colors have to be interpreted. So, changes in basic fonts or basic colors need adaptations for certain classes through redefining the classes in `style.css`.

All images that are used besides the standard **Rime** images can be found in `/images`. Here, you can find the images of the ICEfusion dialog components, for example.

Download at WoweBook.Com

Skin context

The page template above shows that we use two beans for the management of skins. The `ICEfusionConsts` file at `/src/main/java/com/googlecode/icefusion/ui/commons/constant/ICEfusionConsts.java` defines all folders that are normally fixed, but can be changed if you think about moving these inside the web application project. It delivers the `skinBase` that is set to `/src/main/webapp/icefusion/styles/` by default:

```
        private String base = "/icefusion";
        private String skinBase = base + "/styles";
```

For the definition of the CSS files in the page template, this information is sufficient. But if you need skin-dependent paths for standard HTML tags, such as a link for a FavIcon, you also need an extra context string that allows you to create a full-blown web container path. For this, `contextPath` from `ICEfusionConsts` is additionally used as follows:

```
        public String getContextPath() {
            return FacesContext.getCurrentInstance().
            getExternalContext().getRequestContextPath();
        }
```

For the skin selection, all paths use `getSkin()` from the `Context` bean at `/src/main/java/com/googlecode/icefusion/ui/commons/constant/Context.java`, as shown in the following code:

```
    package com.googlecode.icefusion.ui.commons.constant;
    import com.googlecode.icefusion.ui.commons.BackingBeanForm;
    public class Context extends BackingBeanForm {
        Settings settings = new Settings();
        public Settings getSettings() {
            return settings;
        }
        public void setSettings(Settings settings) {
            this.settings = settings;
        }
        public List<Skin> getSkins() {
            return this.getSettings().getSkins();
        }
        public String getSkin() {
            return this.getSettings().getSkin();
        }
```

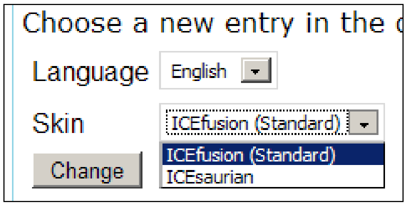
Download at WoweBook.Com

```
        public void setSkin(String code) {
            this.getSettings().setSkin(code);
        }
    }
```

Similar to the locale management, we have a selected `skin` ID and a list of available `skin` that are managed through the `Context` bean. The `Context` bean is used by the backing bean of the settings page in ICEcube to manage the skin selector. The `Context` bean also knows a `settings` reference for persistence.

Skin selector

The implementation of the skin selector looks like this (see ICEcube menu **Extra | Settings**):



Choose a new entry in the

Language English

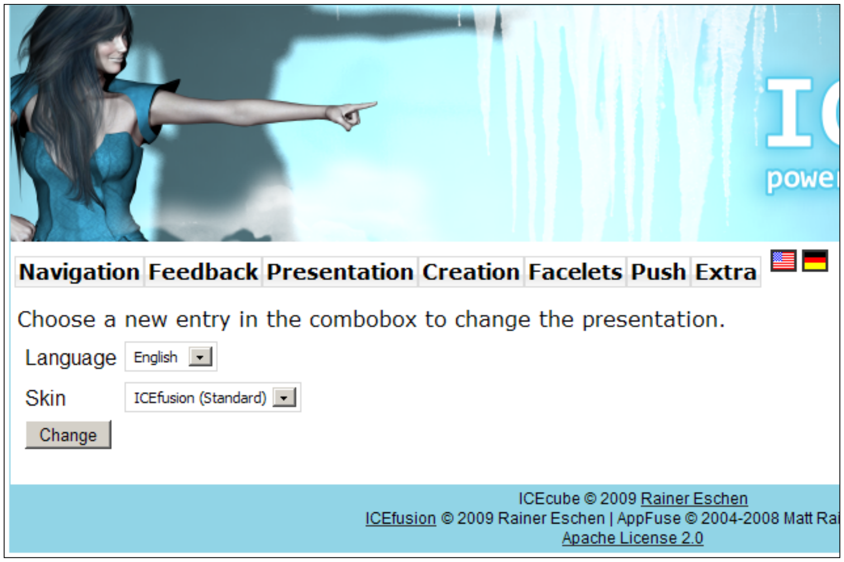
Skin ICEfusion (Standard)

The skin list shows two skin definitions:

- **ICEfusion**: The standard skin almost identical to the ICEfaces Rime skin
- **ICEsaurian**: An experimental skin that uses different images for the logo and the ICEfusion components, and a subtly different color scheme

The **ICEfusion** skin presentation looks like this:

Download at WoweBook.Com


IC
powe
Navigation Feedback Presentation Creation Facelets Push Extra

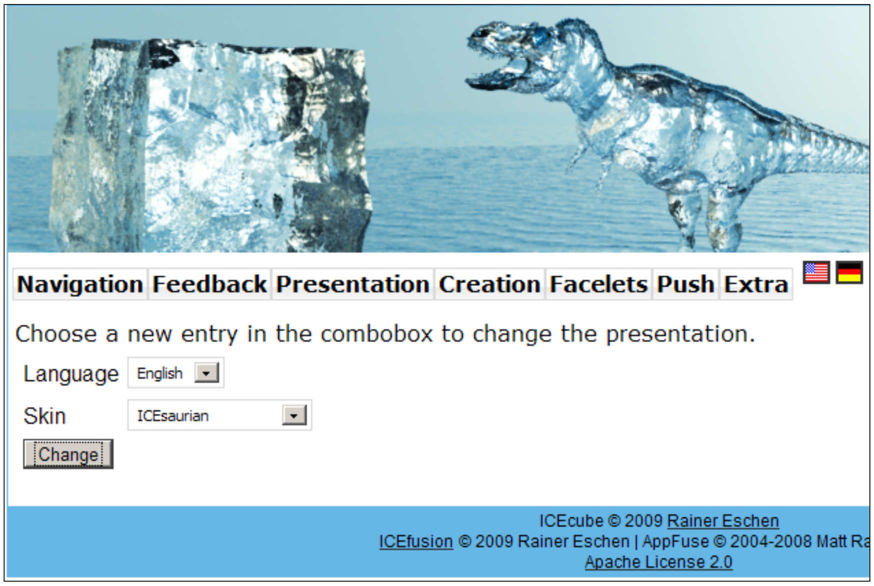
Choose a new entry in the combobox to change the presentation.

Language English

Skin ICEfusion (Standard)

ICEcube © 2009 Rainer Eschen
ICEfusion © 2009 Rainer Eschen | AppFuse © 2004-2008 Matt Rai
Apache License 2.0

The alternative to the pretty woman is an icy dinosaur:


Navigation Feedback Presentation Creation Facelets Push Extra

Choose a new entry in the combobox to change the presentation.

Language English

Skin ICEsaurian

ICEcube © 2009 Rainer Eschen
ICEfusion © 2009 Rainer Eschen | AppFuse © 2004-2008 Matt Ra
Apache License 2.0

Both screenshots show nearly the same presentation of the settings page. The main difference is the logo in this example. But we are not limited to this. Your own skin implementation can change a lot more.

Download at WoweBook.Com

The skin selector is supported by the backing bean of the settings page at `/src/main/java/com/googlecode/icefusion/ui/commons/extra/Settings.java`, as shown in the following code:

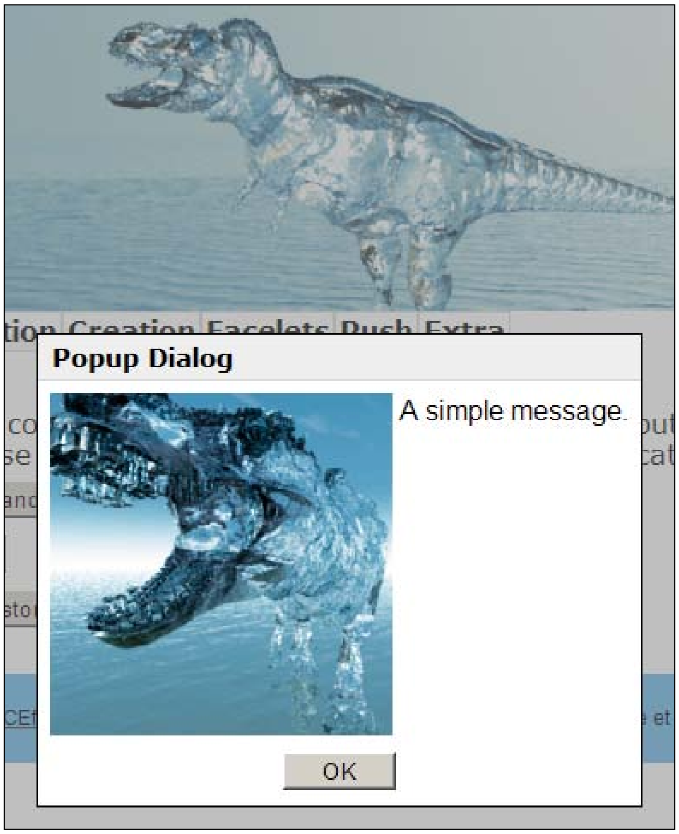
```
package com.googlecode.icefusion.ui.commons.extra;
import javax.faces.model.SelectItem;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.constant.Context;
import com.googlecode.icefusion.ui.commons.constant
  .ICEfusionConsts;
import com.googlecode.icefusion.ui.commons.constant.Skin;
public class Settings extends BackingBeanForm {
    public SelectItem[] getSkins() {
        SelectItem[] skins =
        new SelectItem[this.context.getSkins().size()];
        int i = 0;
        for (Skin skin : this.context.getSkins()) {
            skins[i++] = new SelectItem(skin.getCode(),
            consts.getLocalized(skin.getLabel(),
            "icefusion"));
        }
        return skins;
    }
    public String getSkin() {
        return this.context.getSkin();
    }
    public void setSkin(String skin) {
        this.context.setSkin(skin);
    }
    public String change() {
        return null;
    }
}
```

Its methods use the `Context` bean for the presentation and management of the skin settings.

Download at WoweBook.Com

Skinning in components

For a concrete example of how to use skins in your own components, we will take a look at the ICEfusion `messageDialog` component again. With the **ICEsaurian** skin, it looks like this:



If you have a look at the `XHTML` file at `/src/main/webapp/icefusion/taglibs/dialog/messageDialog.xhtml`, you will recognize that the image reference is dependent on the skin management, as shown in the following code:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
```

Download at WoweBook.Com

```
<body>
      <f:facet name="body">
        <ice:panelGrid>
          <ice:panelGrid columns="2" columnClasses=
            "icePanelPopupImage icePanelPopupText">
            <ice:graphicImage url=
              "#{iceFusionConsts.skinBase}/
              #{context.skin}/images/
              messageDialog.png" />
            <ice:outputText value=
              "#{(not empty text) ? Text :
              ((not empty messageDialog.text) ?
              messageDialog.text : icefusion[
              'application.dialog.message.text'])}"/>
          </ice:panelGrid>
        </ice:panelGrid>
      </f:facet>
</body>
</html>
```

We have a look at the image here because this is the only extension to **Rime** in the **ICEsaurian** skin that is used by the dialog. The code is similar to what we have already seen for the page template. The ICEfusion components reference the `/images` folder that is a part of every skin folder by default. However, you are free to choose a different folder if you want. There is only one thing you have to keep in mind—the skin base folder has to be retained:

```
#{iceFusionConsts.skinBase}/#{context.skin}/
```

Here is an example of a different access to the images by your own dialog component:

```
#{iceFusionConsts.skinBase}/#{context.skin}/images/
  myDialogs/messageDialog.png
```

Designing your own skins

If your web application is highly customizable, you can expect having to implement a lot of customer-specific skins. For such a customer, it is important that the web application shows something familiar. You can achieve this by implementing the skin using an official design guide that is following a corporate design as a part of the corporate identity. If such a guide is not available, you can analyze the customer's web home page and try to imitate it.

Download at WoweBook.Com

When implementing a new skin, we have to take a look at:

- Images that the web application uses, such as a logo, colors, and fonts that have to be used
- ICEfaces components using skin images that influence the color scheme

You may develop your own skin template that helps you to start a new customer skin. We will use the **ICEsaurian** skin like a template skin in the following sections for the adaptation details at which you should take a look.

Images

The most relevant image is the logo image. By default, `/images/logo.png` is used for the header presentation of the web application. You have to follow the dimensions that are defined in the `/page.css`:

```
.layout {
  border: 1px solid;
  border-color: #91D4E6;
  width: 952px;
}
.header {
  height: 180px;
}
```

You are free to change the page layout and define something different from `952x180` for the logo image.

You may need a company logo, which is separated from a second image showing the company's products. If you need to adapt the structure of the header, take a look at the header component at `/src/main/webapp/taglibs/commons/header.xhtml`, as shown in the following code:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
```

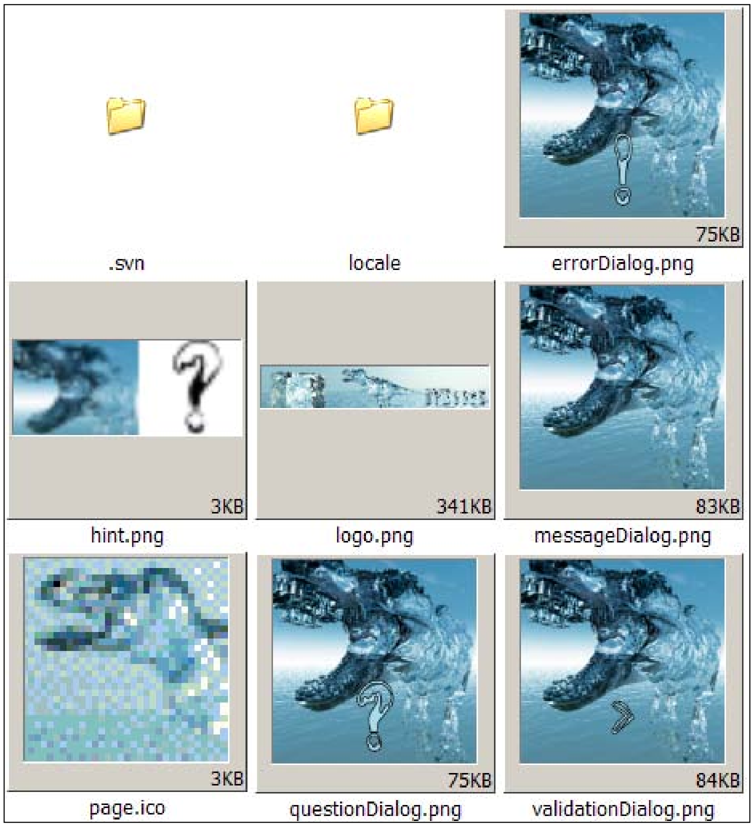
Download at WoweBook.Com

```
      <ice:graphicImage url="#{iceFusionConsts.skinBase}/
        #{context.skin}/images/logo.png" />
    </ui:component>
    </body>
    </html>
```

You can change this to a two-image header like this:

```
    <ui:component>
      <ice:panelGrid columns="2">
       <ice:graphicImage url="#{iceFusionConsts.skinBase}/
          #{context.skin}/images/logo.png" />
        <ice:graphicImage url="#{iceFusionConsts.skinBase}/
          #{context.skin}/images/products.png" />
      </ice:panelGrid>
    </ui:component>
```

There are other images you have to adapt in the `/images` folder. A thumbnail preview helps to get an overview. Here is an example of the **ICEsaurian** skin:



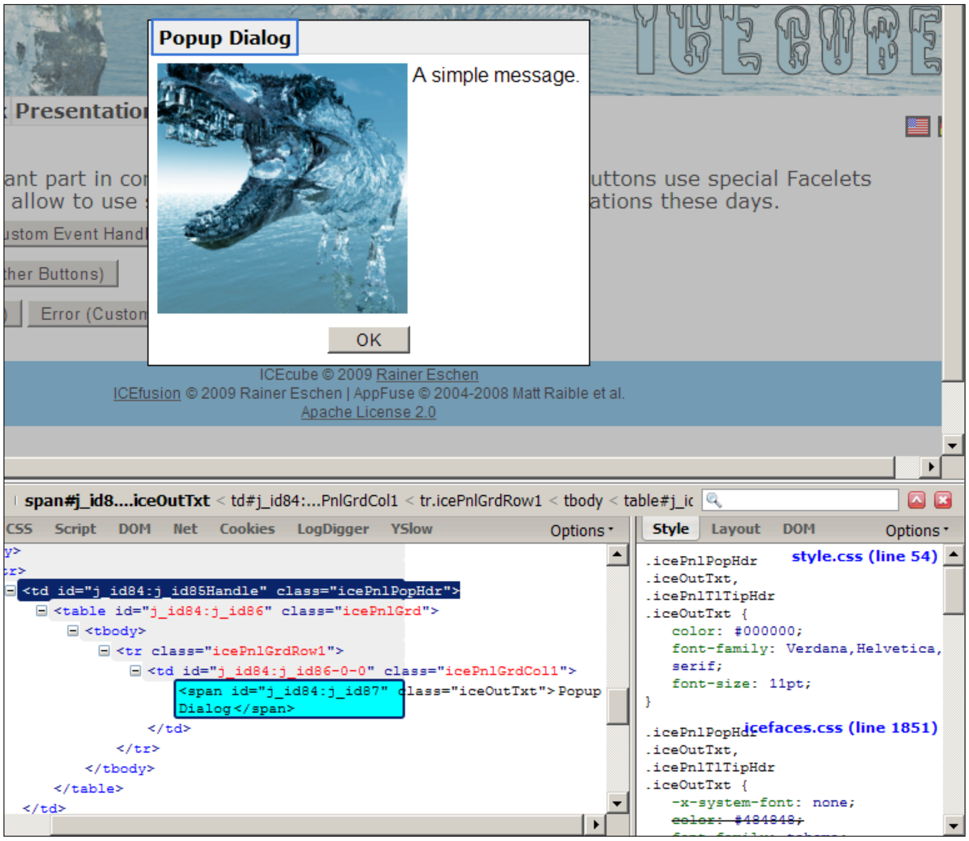
Download at WoweBook.Com

The `/images/locale` folder manages all language flags, which normally do not have to be changed.

Colors and fonts

Basic colors and fonts are managed in the `page.css` file. So, it is a good idea to adapt this file first. A test with the new skin may show some differences in the color schema if you have a look at the different pages of your web application. To recognize which CSS classes from the `icefaces.css` file have to be overwritten through a redefinition in the `style.css` file, you can analyze the rendered HTML code. The most well-known tool for such an analysis is the Firefox plugin, Firebug (`http://getfirebug.com/`).

Firebug allows selecting certain parts of a home page and shows corresponding HTML code. In the following example, the header text of the `messageDialog` is chosen. The HTML code is shown at the bottom left, and the CSS class definition hierarchy is shown on the bottom right:


Popup Dialog
A simple message.
OK
Presentatio
ant part in co
allow to use
ustom Event Hand
her Buttons)
Error (Custon
uttons use special Facelets
ations these days.
ICEcube © 2009 Rainer Eschen
ICEfusion © 2009 Rainer Eschen | AppFuse © 2004-2008 Matt Raible et al.
Apache License 2.0
span#j_id8....iceOutTxt < td#j_id84:...PnlGrdCol1 < tr.icePnlGrdRow1 < tbody < table#j_ic
CSS Script DOM Net Cookies LogDigger YSlow Options
Style Layout DOM Options

```
y>
tr>
<td id="j_id84:j_id85Handle" class="icePnlPopHdr">
  <table id="j_id84:j_id86" class="icePnlGrd">
    <tbody>
      <tr class="icePnlGrdRow1">
        <td id="j_id84:j_id86-0-0" class="icePnlGrdCol1">
          <span id="j_id84:j_id87" class="iceOutTxt">Popup
          Dialog</span>
        </td>
      </tr>
    </tbody>
  </table>
</td>
```

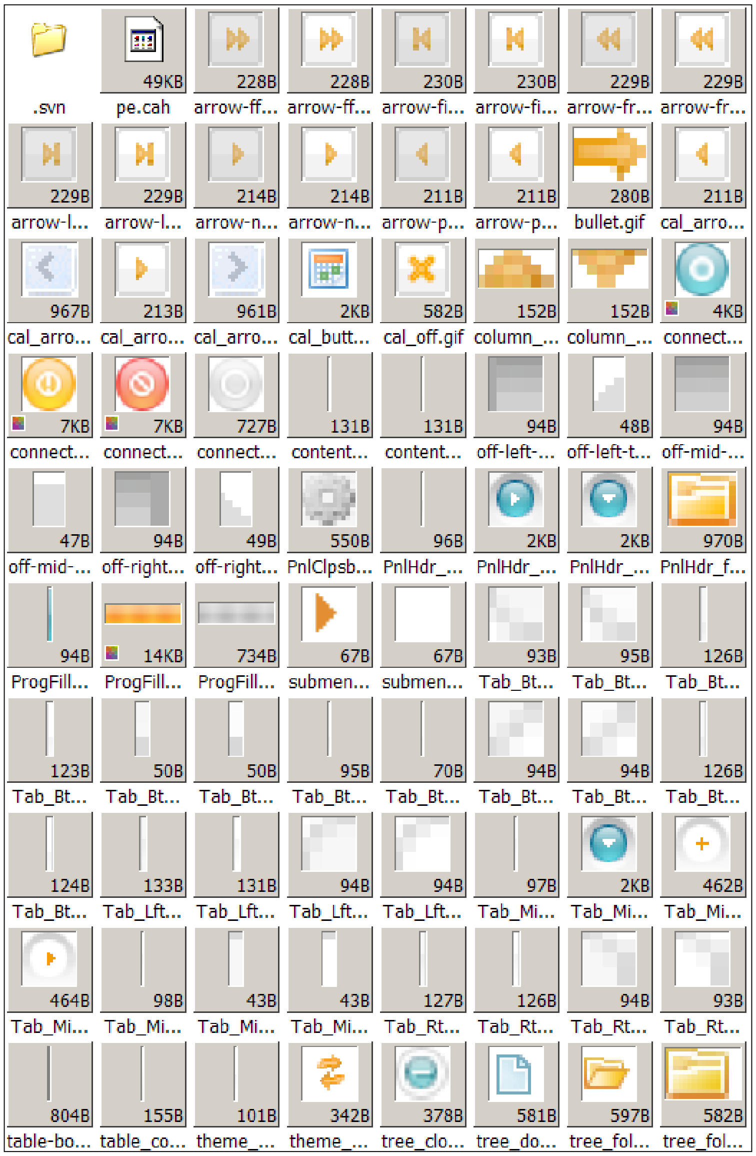
```
.icePnlPopHdr        style.css (line 54)
.iceOutTxt,
.icePnlTlTipHdr
.iceOutTxt {
   color: #000000;
   font-family: Verdana,Helvetica,
   serif;
   font-size: 11pt;
}

.icePnlPopHdr    icefaces.css (line 1851)
.iceOutTxt,
.icePnlTlTipHdr
.iceOutTxt {
   -x-system-font: none;
   ~~color: #484848;~~
```

Download at WoweBook.Com

You can even experiment with the values and change font or background colors to test the results interactively.

ICEfaces components

Firebug can also be used to recognize which images of certain ICEfaces components have to be adapted. This is important because some color effects are based on images, and not on CSS attributes. Here is an extract of the images found in `/css-images`:



Download at WoweBook.Com

All icons can be replaced. But images that are used to realize special presentation effects may have to be re-colored. In the worst case, you have to develop a new presentation effect for a certain ICEfaces component that includes editing all of the involved (single) images.

This kind of adaptation can be very time consuming. If possible, develop a more neutral color presentation for reuse. This allows adapting only the important parts of the skin. So, you can show some corporate design color schema aspects without adapting every component that suffers from a lack of useful skin support.

Summary

This chapter has shown examples for user settings that a modern web application can offer for a better user experience. We took a detailed look at the ICEcube language management, which allows you to add new languages without much effort.

The ICEcube skin management, as a second implementation example, showed you how a customer-specific corporate design can be managed. Following some conventions, the creation of new skins is pretty comfortable.

With the user settings implementation, our web application is now almost complete. For a real-world implementation, we would need to add a context-sensitive help system. But this is a bit out of the scope of this book.

The next chapter will take a detailed look at the implementation aspects of certain ICEfusion components that we used in the previous chapters. This chapter will help you write your own Facelets components that deliver all of the advantages we discussed in the previous chapters using the ICEfusion implementation.

Download at WoweBook.Com

Download at WoweBook.Com

9
Reusable Facelets Components

One of the strengths of ICEfaces is its tight integration with Facelets. This allows you to extend the ICEfaces component set with new components without ever writing JSF custom components from scratch. This chapter takes some of the more advanced ICEfusion components to describe the principles behind such a development, as follows:

- Creating your own taglib and using a new namespace for it
- Adding new tags through the Facelets component definitions
- Managing tag parameters through the JSF Expression Language
- Referencing parameter objects through a backing bean injection
- Reusing the ICEfaces tags by combining them into a new tag

Facelets

Facelets is a view technology for JSF that allows defining pages with less effort. These pages are written in XHTML. One of the advantages of Facelets is that you can update pages without redeployment. A change in a Facelets-based page can be seen immediately in the web browser. You only need to save the change to an XHTML file and reload the page in your web browser. It is even possible to test the page without a servlet container.

Facelets is also a lightweight templating system. It is tightly integrated with JSF and is very flexible to use. Best of all, you can create your own taglib without implementing it in Java.

Download at WoweBook.Com

This chapter cannot describe all the details for using Facelets. So, we will only have a look at how we can define our own tags and how we can use them. If you are new to Facelets, you may also have a look at the developer documentation at `https://facelets.dev.java.net/docs/dev/docbook.html`.

Facelets allows you to create new JSF components. This enables us, for example, to combine ICEfaces tags with reusable Facelets components. For this, we have to define a taglib with a new namespace that lets us reference the new tags.

One of the key features of Facelets is the ability of describing the main functionality of new tags in XHTML instead of Java code. This allows us to have tags written in pure XHTML. We can use this feature to combine the existing ICEfaces tags to establish a reusable and standardized presentation.

For a more complex behavior of our new Facelets components, we can add the backing bean concept. This allows you to use all the means Java delivers for challenging implementations. The underlying Expression Language that combines tags and the backing beans is an additional powerful tool in itself. So, you may have to decide if you want to implement component functionality in a backing bean method, or as a more complex expression of the Expression Language in the XHTML part. A rule of thumb is to use the backing bean implementation if the readability of the Expression Language code suffers.

Taglibs

The ICEfusion components are defined through a single taglib. This allows you to use the same namespace for all tags throughout the web application. The ICEfusion taglib can be found at `/src/main/webapp/icefusion/taglibs/icefusion.taglib.xml` and looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib>
  <namespace>
    http://icefusion.googlecode.com/icefusion
  </namespace>
  <tag>
    <tag-name>header</tag-name>
    <source>commons/header.xhtml</source>
  </tag>
  <tag>
    <tag-name>navigation</tag-name>
    <source>commons/navigation.xhtml</source>
```

Download at WoweBook.Com

```
    </tag>
    <tag>
      <tag-name>connectionStatus</tag-name>
      <source>commons/connectionStatus.xhtml</source>
    </tag>
    <tag>
      <tag-name>messageDialog</tag-name>
      <source>commons/dialog/messageDialog.xhtml</source>
    </tag>
    <tag>
      <tag-name>errorDialog</tag-name>
      <source>commons/dialog/errorDialog.xhtml</source>
    </tag>
    <tag>
      <tag-name>questionDialog</tag-name>
      <source>commons/dialog/questionDialog.xhtml</source>
    </tag>
  </facelet-taglib>
```

The taglib lists tag names and corresponding references to XHTML definition files. These files define how the component implementation looks, for example, if backing beans are used or not.

It is possible to choose an individual folder structure to manage the component implementations.

For ICEfusion, the components are organized next to the `/src/main/webapp/icefusion/taglibs/icefusion.taglib.xml` using the `commons` subfolder:

- `dialog`: Components for dialog presentation
- `form` : Components for more complex form handling
- `help`: Components for support of self-descriptiveness and learning
- `validation`: Components to support input validation

This structure is extensible. So we could have, for example, more specific components that are useful in different industries. Each industry would have its own folder besides the `commons` folder then.

The second important part in the taglib is the definition of the namespace that is used with the new tags. It can be found on top of the taglib file. For ICEfusion, we use:

```
    <namespace>
      http://icefusion.googlecode.com/icefusion
    </namespace>
```

Download at WoweBook.Com

The string is following a convention. But the URL never delivers something useful, or even the taglib itself. This string is only relevant for the Facelets template files or page files you want to use the new tags for.

An ICEfusion definition for such files looks like this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
  <icefusion:connectionStatus />
</body>
```

If we use the namespace identifier `icefusion` with a tag name, such as `connectionStatus`, inside a page, Facelets recognizes that the taglib definition for `http://icefusion.googlecode.com/icefusion` has to be used to find the corresponding component implementation.

Tags

Facelets allows combining JSF tags. For this, you have to define a Facelets component structure like we did for the `<icefusion:connectionStatus>`:

Download at WoweBook.Com

You can find the component definition at `/src/main/webapp/icefusion/taglibs/commons/connectionStatus.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <div id="divStayTopLeft" style="position:absolute">
    <ice:outputConnectionStatus />
  </div>
  <script type="text/javascript">
    //<![CDATA[
    var verticalpos="fromtop"
    JSFX_FloatTopDiv();
    //]]>
  </script>
</ui:component>
</body>
</html>
```

Only JSF tags between the `<ui:component>` tags are relevant to Facelets. Every page or template you use the `connectionStatus` tag for gets a replacement with the JSF tags from the component during runtime.

The `connectionStatus` is a good example of a pure XHTML component definition. It uses the corresponding ICEfaces `outputConnectionStatus` component that shows a status icon for the connection between web browser and web container. Additionally, the component has a JavaScript definition that allows a free-floating behavior of the icon inside the web browser. So, if you scroll down the browser's content, the icon stays on the top left.

Although `connectionStatus` is pretty simple, it shows all advantages of the Facelets component concept:

- We do not have to code in Java in any case, or do not have to implement something complex.
- The implementation can be done very quickly and the maintenance is kept easy.
- Page and template code becomes simpler and more readable.
- The existing ICEfaces component model can be easily extended according to the needs of your project.

Download at WoweBook.Com

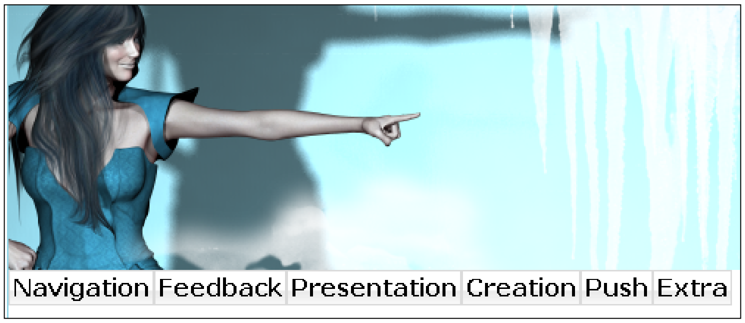
- We can standardize more complex components, so that it is possible to change their presentation or behavior at any time in a central place. This is a powerful tool in bigger projects to reduce complexity.
- With an efficient component design, you get a better reusability in your project.

Component logic

For more complex components, we need additional logic. We have three possibilities to implement this for a Facelets component:

- JSTL-conform statements, such as `<c:if>`
- Backing bean methods
- A combination of both

Here is an example of the third possibility:



The component implementation looks like this (`/src/main/webapp/icefusion/taglibs/commons/navigation.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <c:if test="#{context.dynamicMenu}">
    <icefusion:dynamicMenu/>
```

Download at WoweBook.Com

```
    </c:if>
    <c:if test="#{!context.dynamicMenu}">
      <icefusion:menu/>
    </c:if>
    <icefusion:menuIcons />
</ui:component>
</body>
</html>
```

The `navigation` component manages the navigation area of the page template. ICEfusion allows managing a dynamic menu and a static menu. This component checks which menu type has to be rendered through the `context` bean (`/src/main/java/com/googlecode/icefusion/ui/commons/constant/Context.java`). The `<c:if>` statement helps to suppress the unnecessary menu definition. Additionally, the navigation renders the menu icons that are shown on the right side of the pull-down menu.

We have a reference to a backing bean in this code and also to a JSTL logic element, the `<c:if>` statement. As the `context` bean is a more global management construct, and not a corresponding backing bean of the `navigation` component, it makes sense to put in some JSTL logic instead.

This logic should not be a part of the `context` bean because we would violate the *separation of concerns* rule. However, this kind of implementation is only useful for simple implementations. With a more complex component, we would have a backing bean anyway and would put the logic into the corresponding backing bean, of course.

This example also shows the limitations of the JSTL logic elements. In our case, a missing `else` statement negatively influences the readability of the code. The code gets worse with the number of conditions that have to be processed.

Experience shows that it is a good idea to have one backing bean for every Facelets component. This corresponds to the rule of having one backing bean for every page in your web application.

Here is an example of a dedicated backing bean used by the `dynamicMenu` component defined at `/src/main/webapp/icefusion/taglibs/commons/dynamicMenu.xhtml` (The presentation result is the same as the screenshot above.):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
```

Download at WoweBook.Com

```
  xmlns:icefusion="http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelGrid columns="2">
    <ice:form>
      <ice:menuBar noIcons="true">
      <ice:menuItems value="#{dynamicMenu.menuModel}" />
      </ice:menuBar>
    </ice:form>
  </ice:panelGrid>
</ui:component>
</body>
</html>
```

The `dynamicMenu` backing bean defines the menu structure. This is a simple example of such a usage. For more complex components, we would use more methods of the backing bean.

You may recognize that the naming convention uses the same label in both contexts: backing bean and the XHTML file. This helps to recognize relations if you have a look into different folders of the web application. Experience shows that it is even useful to use this principle for the folder names. For this, ICEfusion and ICEcube have similar structures in the `/src/main/java` and `/src/main/webapp` folder branches.

Attribute parameters

If you have a look at the JSF components, you will find attributes you can use to influence their behavior. We can define such attributes with Facelets components too. For this, we use the principles of the JSF Expression Language we already followed in the previous chapters.

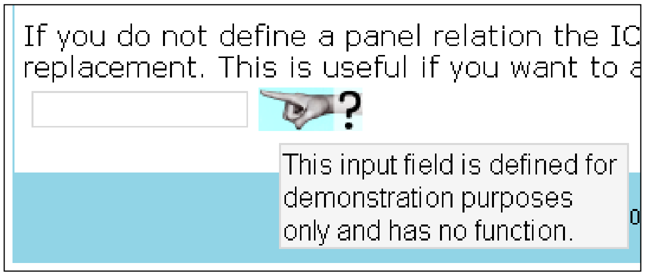
Attributes are defined indirectly through the use inside an expression by a component's tag. So, if you reference a `#{my_attribute}` attribute in the component, and the component tag has an attribute set with the same name:

```
<icefusion:my_component my_attribute="test"/>
```

the `#{my_attribute}` expression delivers the `test` value.

Download at WoweBook.Com

We start with the ICEfusion `hint` component for the first example:



If you do not define a panel relation the IC
replacement. This is useful if you want to a

This input field is defined for demonstration purposes only and has no function.

The syntax for the `hint` component use looks like this:

```
<icefusion:hint title="my_title" text="my_text"
  panel="my_panel_context"/>
```

Here is the code for the component implementation (`/src/main/webapp/icefusion/taglibs/commons/help/hint.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelTooltip hideOn="mouseout" hoverDelay="200"
    id="#{(not empty panel) ? panel : 'hintIcon'}">
    <c:if test="#{not empty title}">
      <f:facet name="header">
        <ice:outputText value="#{title}"/>
      </f:facet>
    </c:if>
    <f:facet name="body">
      <ice:outputText value="#{(not empty text) ?
        text : icefusion['application.hint.none']}"/>
    </f:facet>
  </ice:panelTooltip>
  <ice:panelGroup panelTooltip="hintIcon"
    rendered="#{empty panel}">
    <ice:graphicImage url="#{iceFusionConsts.skinBase}/
      #{context.skin}/images/hint.png" />
```

Download at WoweBook.Com

```
    </ice:panelGroup>
  </ui:component>
  </body>
  </html>
```

This implementation shows the JSTL logic elements that are controlled by a component attribute. The `title` attribute defines if the title is rendered or not. If this attribute's value is `not empty`, the `facet` header is shown. Something similar is done with the `text` to show. If the `text` is not set, a standard text is shown instead. It informs about the missing text.

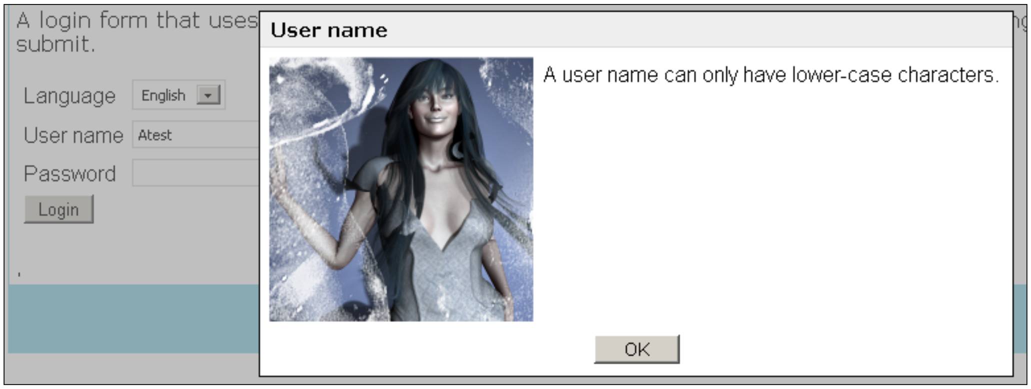
The most interesting feature with this implementation is the handling of the panel reference. If the `panel` attribute is set, the ICEfaces component `panelTooltip` takes this object for reference. If the `panel` attribute is not set, it takes the `hintIcon` reference to a panel the component has defined itself.

The correct use of interfaces and classes is important with the Facelets component attributes. Although XHTML processes everything as text, the JSF components process interfaces and classes as is.

Attribute references

The `panel` attribute of the `hint` component already showed that it is possible to work with references. We now have a look at self-defined object references. These allow working with an external context inside a Facelets component.

We will have a look at the `validationDialog` component, for example:


A login form that uses
submit.

User name

A user name can only have lower-case characters.

The syntax looks like this:

```
  <icefusion:validationDialog eventBean=
    "#{my_validation_processor_bean}" />
```

Download at WoweBook.Com

We set an `eventBean` that defines the reference to work with. With self-defined object references, it is important that the developer of such an object knows what is to be delivered so that the Facelets component can work correctly. For this, we define an `Interface`.

The `validationDialog` references the `IValidationProcessor` interface that can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/validation/IValidationProcessor.java`:

```
package com.googlecode.icefusion.ui.commons.validation;
import java.io.Serializable;
public interface IValidationProcessor extends Serializable {
    public List<String> getValidationMessages();
    public void setValidationMessages(List<String> messages);
    public Boolean getValidationErrorStatus();
    public void setValidationField(String field);
    public String getValidationField();
    public String validationDialogButtonOk();
}
```

Here is the code for the `validationDialog` component implementation (`/src/main/webapp/icefusion/taglibs/commons/validation/validationDialog.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:form>
    <ice:panelPopup autoCentre="false" draggable="false"
      modal="true"
      rendered="#{eventBean.validationErrorStatus}"
      visible="#{eventBean.validationErrorStatus}">
      <f:facet name="header">
        <ice:panelGrid>
          <ice:outputText value=
            "#{icefusion[eventBean.validationField]}"/>
        </ice:panelGrid>
      </f:facet>
```

Download at WoweBook.Com

```
        <f:facet name="body">
          <ice:panelGrid>
            <ice:panelGrid columns="2" columnClasses=
              "icePanelPopupImage icePanelPopupText">
              <ice:graphicImage url=
                "#{iceFusionConsts.skinBase}/
                #{context.skin}/images/
                validationDialog.png" />
              <ice:panelGrid>
                <c:forEach var=»message» items=
                  "#{eventBean.validationMessages}">
                  <ice:outputText value=
                    "#{icefusion[message]}"/>
                </c:forEach>
              </ice:panelGrid>
            </ice:panelGrid>
            <ice:panelGrid columns="1" styleClass=
              "icePanelPopupButtons">
              <ice:commandButton value="#{icefusion[
                'application.validation.
                message.button.ok']}"
                action="
                  #{eventBean.validationDialogButtonOk}"
                styleClass="icePanelPopupButton"/>
            </ice:panelGrid>
          </ice:panelGrid>
        </f:facet>
      </ice:panelPopup>
    </ice:form>
  </ui:component>
  </body>
  </html>
```

The `eventBean` is used in two modes:

- Get information for processing
- Call external event handler

The information is delivered by

```
public Boolean getValidationErrorStatus();
public String getValidationField();
public List<String> getValidationMessages();
```

Download at WoweBook.Com

to decide if the dialog is rendered, which field is currently validated, and which validation errors have to be shown.

The external event handler for the **OK** button is delivered by:

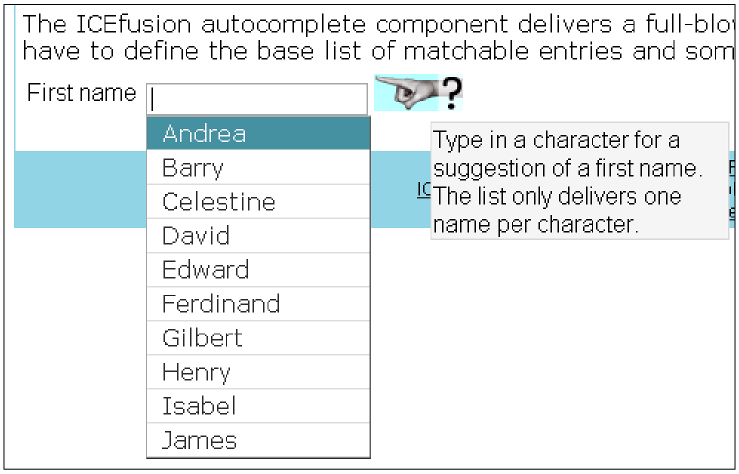
```
public String validationDialogButtonOk();
```

Backing bean injection through the Facelets attribute

The next step in referencing is to use the referenced object in the backing bean of the Facelets component. Although you can set such injections with the help of the Spring configuration, it is more intuitive and also more flexible to do so through an attribute.

Using interfaces for parameter passing

Let's have a look at the `completer` component:



The ICEfusion autocomplete component delivers a full-blo
have to define the base list of matchable entries and som

First name

Andrea
Barry
Celestine
David
Edward
Ferdinand
Gilbert
Henry
Isabel
James

Type in a character for a suggestion of a first name. The list only delivers one name per character.

The syntax looks like this:

```
<icefusion:completer title="my_title"
  hintTitle="my_hint_title" hintText="my_hint_text"
  valueBean="#{my_value_and_baseList_manager}"
  rows="my_number_for_entries_in_hit_list" />
```

Download at WoweBook.Com

The backing bean injection is done through the `valueBean` attribute. Here is the example page (`/src/main/webapp/icecube/feedback/autocomplete.xhtml`) that shows how it is used:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
  "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name=»title»>
    #{icecube['application.menu.feedback.autocomplete']}
  </ui:define>
  <ui:define name=»content»>
    #{icecube['application.menu.feedback.autocomplete
      .text']}
    <ice:form>
      <icefusion:completer title=»#{icecube[
        'application.menu.feedback.autocomplete
        .completer.title']}» hintText=»#{icecube[
        'application.menu.feedback.autocomplete
        .completer.hint.text']}»
        valueBean="#{autocomplete}"/>
    </ice:form>
  </ui:define>
</ui:composition>
</body>
</html>
```

The `ICompleter` interface exists for objects that are set via the `valueBean` attribute (`/src/main/java/com/googlecode/icefusion/ui/commons/form/ICompleter.java`), as shown:

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
import javax.faces.model.SelectItem;
public interface ICompleter extends Serializable {
    public List<SelectItem> getCompleterBaseList();
    public String getCompleterValue();
```

Download at WoweBook.Com

```
        public void setCompleterValue(String value);
    }
```

The implementation of the `completer` component looks like this (`/src/main/webapp/icefusion/taglibs/commons/form/completer.xhtml`):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:panelGrid columns="3">
    <ice:outputText value="#{(not empty title) ? title :
      icefusion['application.completer.none']}"/>
    <ice:selectInputText value="#{valueBean.completerValue}"
      valueChangeListener="#{completer.listener}"
      rows="#{(not empty rows) ? rows : completer.rows}">
        <f:selectItems value="#{completer.resultList}" />
    </ice:selectInputText>
          <icefusion:hint title="" text="#{hintText}"/>
  </ice:panelGrid>
  <!-- Manage valueBean list as parameter for backing
    bean -->
  <ice:selectInputText visible="false" binding=
    "#{completer.baseList}" listValue=
      "#{valueBean.completerBaseList}" />
</ui:component>
</body>
</html>
```

To inject the object reference into the backing bean, we use a trick. We define an artificial JSF component in our example, the ICEfaces `selectInputText`, and use a component binding through its `binding` attribute. This allows us to access the object that is injected inside the backing bean.

Download at WoweBook.Com

Hiding the ICEfaces components used for parameter passing

When you use ICEfaces components for a parameter passing, you have to take care of the rendering. These components will not be a part of the visual presentation. We can:

- Set an attribute that an ICEfaces component offers for visibility management
- Manage the visibility ourselves using cascading stylesheet definitions

Here is an example of a visibility attribute:

```
<ice:selectInputText visible="false"
  binding="#{completer.baseList}"
  listValue="#{valueBean.completerBaseList}" />
```

The `visible` attribute allows hiding the `selectInputText` we use in the `completer` component.

If an ICEfaces component does not deliver a visibility attribute, you can use the `style` attribute instead:

```
<ice:dataTable style="visibility:hidden;"
  binding="#{login.parameters}"
  value="#{eventBean}"/>
```

This is an explicit management of visibility using a standard stylesheet attribute. You have to keep in mind that this will be more expensive to use because the ICEfaces renderer has no possibility of optimization.

Accessing the Facelets attribute references

Which JSF component we use for the parameter passing depends on the object you want to inject because we have a strict type checking here. For example, for a generic list of objects, the `dataTable` component is a good choice.

Download at WoweBook.Com

In the `completer` example above, we use similar structures for injection (input) and presentation (output). So, we use the same component type here for both. But, this is only done to make the code more readable. As we have to typecast all parameters in the backing bean, and the parameter passing concept using the ICEfaces components is generic, we have a very flexible choice of the component to use.

Our parameter passing component in the `completer` example is referenced via the `baseList` attribute in the backing bean for component binding, and gets a special kind of list for the `listValue` attribute. The list type is defined in the `ICompleter` interface description:

```
public List<SelectItem> getCompleterBaseList();
```

`SelectItem` is a list element definition of the JSF standard implementation. It describes list elements that are used in JSF presentation components (for example, comboboxes).

The backing bean of the `completer` component looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/form/Completer.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.icesoft.faces.component.selectinputtext
  .SelectInputText;
public class Completer extends BackingBeanForm {
  private SelectInputText baseList;
  private List<SelectItem> matches =
    new ArrayList<SelectItem>();
  public void listener(ValueChangeEvent event) {
    if (event.getComponent() instanceof SelectInputText) {
      String search = (String) event.getNewValue();
      Long matches_i = 0L;
      matches.clear();
      for (SelectItem entry : (ArrayList<SelectItem>)this
        .getBaseList().getListValue()) {
        if ((matches_i > this.getRows())) {
          break;
        }
        if (entry.getLabel().toString().toUpperCase(
          this.context.getSettings().getLocale()
          .getLocale()).startsWith(search.toUpperCase(
          this.context.getSettings().getLocale()
          .getLocale()))) {
```

Download at WoweBook.Com

```
                  matches.add(entry);
                  matches_i++;
                 }
          }
        }
      }
    }
```

The `listener()` method uses the `baseList` attribute to get access to the list of searchable items via the `listValue` getter. We need an explicit typecast to `SelectItem` to get back the original object type defined in the interface description.

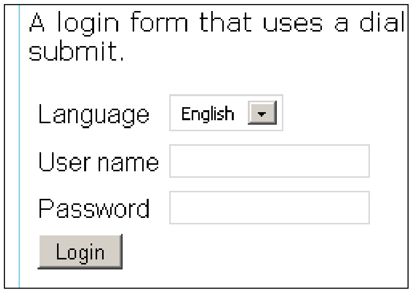
Facelets component reuse

The `completer` component shows another interesting aspect with the Facelets components design: reuse of the Facelets components inside other Facelets components. In our example, we have a ICEfusion `hint` component added to the `completer`:

```
<ice:panelGrid columns="3">
  <ice:outputText value="#{(not empty title) ? title :
    icefusion['application.completer.none']}"/>
  <ice:selectInputText value="#{valueBean.completerValue}"
    valueChangeListener="#{completer.listener}"
    rows="#{(not empty rows) ? rows : completer.rows}">
      <f:selectItems value="#{completer.resultList}" />
  </ice:selectInputText>
  <c:if test="#{(not empty hintText)}">
          <icefusion:hint title="" text="#{hintText}"/>
</ice:panelGrid>
```

The `hint` component is used transparently because the corresponding parameters are not processed directly.

Another example for reuse of Facelets components is the `login` component:


A login form that uses a dial
submit.

Language English
User name
Password
Login

Download at WoweBook.Com

It uses the following syntax:

```
<icefusion:login eventBean="#{action_handler_for_login}" />
```

The `login` component processes **User name** and **Password** through an external reference that is set via the `eventBean` attribute. The interface for the bean implementation looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/form/ILogin.java`):

```
package com.googlecode.icefusion.ui.commons.form;
import java.io.Serializable;
public interface ILogin extends Serializable {
    public void setLoginUsername(String username);
    public void setLoginPassword(String password);
    public String loginAction();
}
```

The management of `username` and `password` is externalized. The processing is initialized by the `loginAction` event that is called by the `login` component when the button is clicked in the form.

The component implementation can be found at `/src/main/webapp/icefusion/taglibs/commons/form/login.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <!-- Manage eventBean as parameter for backing bean  -->
    <ice:dataTable style="visibility:hidden;" binding=
    "#{login.parameters}" value="#{eventBean}"/>
  <ice:form>
    <ice:panelGrid>
      <ice:panelGrid columns="2">
        <ice:outputText value="#{icefusion[
          'application.login.language']}" />
        <ice:selectOneMenu value="#{login.locale}"
          partialSubmit="true">
          <f:selectItems value="#{login.locales}"/>
        </ice:selectOneMenu>
```

Download at WoweBook.Com

```
          <ice:outputText value="#{icefusion[
            'application.login.username']}" />
          <ice:inputText value="#{login.username}"/>
          <ice:outputText value="#{icefusion['
            application.login.password']}" />
          <ice:inputSecret value="#{login.password}"/>
          <ice:commandButton value="#{icefusion[
            'application.login.button']}"
            action="#{login.login}" />
        </ice:panelGrid>
      </ice:panelGrid>
    </ice:form>
    <icefusion:validationDialog eventBean="#{login}"/>
  </ui:component>
  </body>
  </html>
```

The `eventBean` attribute is managed by the backing bean of the `login` component. For the validation results, it uses an ICEfusion `validationDialog` component that we discussed previously. The `eventBean` attribute of the `ValidationDialog` is implemented by the backing bean of the `login` component (`/src/main/java/com/googlecode/icefusion/ui/commons/form/Login.java`):

```
  package com.googlecode.icefusion.ui.commons.form;
  import javax.faces.model.SelectItem;
  import com.googlecode.icefusion.ui.commons.BackingBeanForm;
  import com.googlecode.icefusion.ui.commons.validation
    .IValidationProcessor;
  public class Login extends BackingBeanForm implements
  IValidationProcessor {
    private List<String> validationMessages =
      new ArrayList<String>();
    private String validationField;
    public String validationDialogButtonOk() {
      this.validationMessages.clear();
      return null;
    }
    public Boolean getValidationErrorStatus() {
      return this.validationMessages.size() > 0;
    }
    public List<String> getValidationMessages() {
      return this.validationMessages;
    }
    public void setValidationMessages(List<String> messages) {
      this.validationMessages = messages;
```

Download at WoweBook.Com

```
    }
    public String getValidationField() {
      return this.validationField;
    }
    public void setValidationField(String field) {
      this.validationField = field;
    }
  }
```

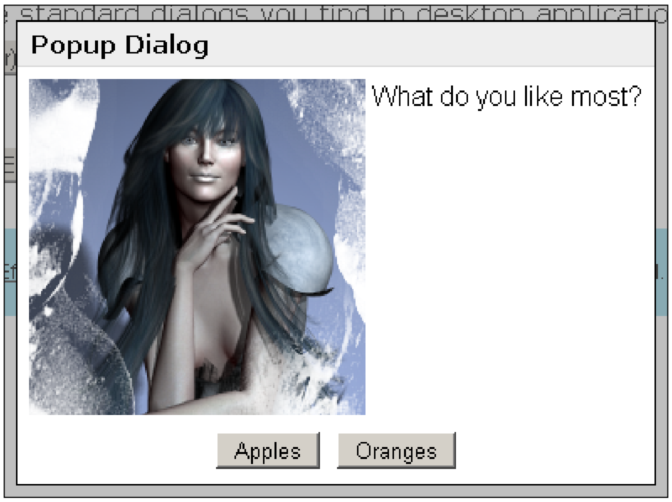
If the **Login** button is clicked, the `login` event initializes the validation for the **User name** and **Password** fields. Before the validation is done for each field, its label is set for the `validationDialog`. If the validation recognizes errors, it delivers the corresponding error messages back to the `login` backing bean. Then, these are referenced by the `validationDialog` because the `errorStatus` is true when more than one message exists.

Initialization through Facelets or backing bean attributes

All previous examples used the Facelets component attributes to initialize the component. But a component can also be initialized through the attributes of a backing bean. We can use the Spring context for this if the web application allows a component initialization during the web application initialization. Alternatively, we use injected objects in the backing bean and set attributes during runtime via processed events. But we do not have to decide between the Facelets attributes and the backing bean attributes initialization. Indeed, it is possible to mix those so that the developer can decide when to initialize what and how to do this.

Download at WoweBook.Com

For an example, we will take a look at the `questionDialog` component:



Here is the syntax:

```
<icefusion:questionDialog
  title="my_title" text="my_text"
  yes="my_yesButton_text" no="my_noButton_text"
  rendered="true|false"
  eventBean="#{my_button_event_bean}"/>
```

All presentation parts of the dialog can be set through the Facelets attributes. The event handling is managed externally via the `eventBean` attribute. The corresponding `IQuestionDialog` interface looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/dialog/IQuestionDialog.java`):

```
package com.googlecode.icefusion.ui.commons.dialog;
import java.io.Serializable;
public interface IQuestionDialog extends Serializable {
    public String questionDialogButtonYes();
    public String questionDialogButtonNo();
}
```

Depending on the button that is clicked in the dialog, the `yes` or `no` variant is processed.

Download at WoweBook.Com

The implementation of the `questionDialog` component can be found at `/src/main/webapp/icefusion/taglibs/commons/dialog/questionsDialog.xhtml`, as shown:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:form>
    <ice:panelPopup autoCentre="false" draggable="false"
      modal="true" rendered="#{(not empty rendered) ?
        rendered : questionDialog.show}"
      visible="#{(not empty rendered) ? Rendered :
        questionDialog.show}">
      <f:facet name="body">
            <c:if test="#{empty eventBean}">
              <ice:commandButton value=
                "#{(not empty yes) ? yes : ((not empty
                  questionDialog.yes) ?
                  questionDialog.yes : icefusion[
                  'application.dialog.question.button
                  .yes'])}" action="#{questionDialog
                    .questionDialogButtonYes}»
                styleClass=»icePanelPopupButton»/>
              <ice:commandButton value=
                "#{(not empty no) ? no : ((not empty
                questionDialog.no) ?
                questionDialog.no : icefusion[
                'application.dialog.question.button
                .no'])}" action="#{questionDialog
                  .questionDialogButtonNo}»
                styleClass=»icePanelPopupButton»/>
            </c:if>
      </f:facet>
    </ice:panelPopup>
  </ice:form>
</ui:component>
</body>
</html>
```

Download at WoweBook.Com

If we take a look at the rendered attribute of the ICEfaces panelPopup component, we find this expression:

```
<ice:panelPopup rendered=
  "#{(not empty rendered) ? rendered : questionDialog.show}"
/>
```

We have a rendered attribute from the questionDialog component that is checked first. So, the Facelets component attribute gets a higher priority. If it is not set, the value is taken from the show attribute of the backing bean. It is important that the backing bean delivers a valid value in any case. In this example, the boolean fulfills this. But in cases where you use other attribute types, you have to prevent a null value delivery inside the backing bean.

When we need pure strings for the output, we can manage defaults inside the XHTML file:

```
<ice:commandButton value=
  "#{(not empty yes) ? Yes : (
    (not empty questionDialog.yes) ? questionDialog.yes :
      icefusion[
        'application.dialog.question.button.yes'])}"
/>
```

First, the questionDialog attribute yes is processed. If this is not set, the backing bean attribute yes is checked. If this is also not defined, we take a resource bundle ID for processing.

The backing bean for the questionDialog component looks like this (/src/main/java/com/googlecode/icefusion/ui/commons/dialog/QuestionDialog.java):

```
package com.googlecode.icefusion.ui.commons.dialog;
public class QuestionDialog extends Dialog implements
  IQuestionDialog {
  private String yes;
  private String no;
  private Boolean yesClicked;
  public String getYes() {
    return yes;
  }
  public void setYes(String yes) {
    this.yes = yes;
  }
  public String getNo() {
    return no;
  }
  public void setNo(String no) {
    this.no = no;
```

Download at WoweBook.Com

```
      }
      public Boolean getYesClicked() {
        return yesClicked;
      }
      public void setYesClicked(Boolean yesClicked) {
        this.yesClicked = yesClicked;
      }
      public String questionDialogButtonYes() {
        this.setShow(false);
        this.setYesClicked(true);
        return null;
      }
      public String questionDialogButtonNo() {
        this.setShow(false);
        this.setYesClicked(false);
        return null;
      }
    }
```

The backing bean implements the `IQuestionDialog` interface by default. So, there is no real difference to an external bean implementation you would set via a Facelets attribute. The expressions in the XHTML file allow keeping a "Backing Bean Attribute Only" usage simple. You can use the setter for the presentation strings and leave the `questionDialog` Facelets attributes unset.

An example of a mix with a Facelets attribute initialization is setting an external reference bean that processes the button clicks, but uses the backing bean to set the presentation strings. This can be useful if you always use the same presentation, but want to manage different contexts for the dialog answer buttons.

Summary

Facelets allow you to extend ICEfaces so that you can build even more powerful components. In contrast to the custom JSF component development, this can be done much more easily. But easier does not mean that you get only simple results. The Facelets concept is flexible enough to enable you to implement even more complex components. This helps in designing reusable components which, in turn, help to keep the maintenance efforts minimal.

Until now, we primarily had a look at the standard JSF programming model that follows the request-response pattern. This needs interaction in the web browser to get a different presentation.

The last chapter will discuss the AJAX Push technology that allows you to initiate a change in the browser presentation from the web container side.

Download at WoweBook.Com

Download at WoweBook.Com

10
Push Technology

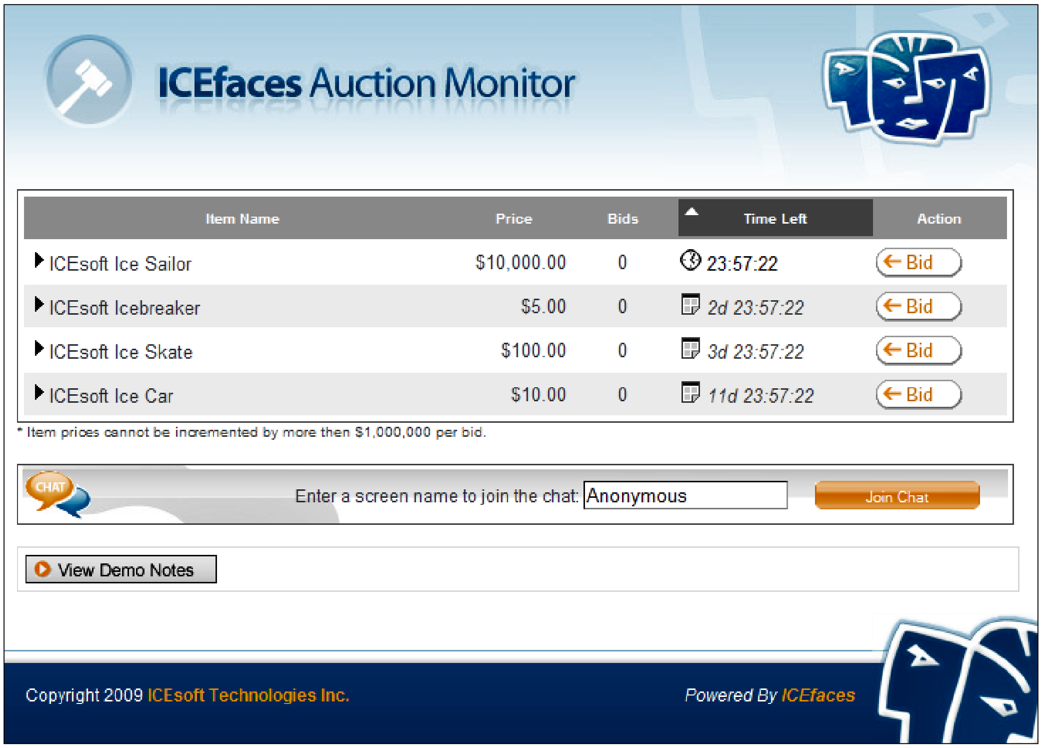
This chapter will discuss one of the most exciting features of ICEfaces: AJAX-based push technology, or AJAX Push for short. First, we will take a look at how to configure its use for our project. The implementation of a progress dialog will show how it works concretely. Finally, we will take a look at ICEmapper, an AJAX Push game that shows how to implement multiuser concepts.

AJAX Push

The concept behind AJAX Push is quite simple. Instead of letting the web browser initiate the rendering according to the request-response pattern of classic web containers, the server does it on its own. Additionally, this is done in a multicast so that a one-to-many communication is established. With this, the server communicates with a predefined group of users, or web browser instances, that are updated in parallel.

Download at WoweBook.Com

The use of AJAX Push is interesting in contexts where users share information and need near real-time updates about changes. Auctions or chats are such examples. You can find a live system from the examples of the ICEfaces sources distribution at `http://auctionmonitor.icefaces.org`. For a test, open two web browsers or more and use them for different bids. You can even start a chat conversation. Here is a screenshot that shows how it looks:



Item Name	Price	Bids	Time Left	Action
ICEsoft Ice Sailor	$10,000.00	0	23:57:22	Bid
ICEsoft Icebreaker	$5.00	0	2d 23:57:22	Bid
ICEsoft Ice Skate	$100.00	0	3d 23:57:22	Bid
ICEsoft Ice Car	$10.00	0	11d 23:57:22	Bid

* Item prices cannot be incremented by more then $1,000,000 per bid.

CHAT
Enter a screen name to join the chat: Anonymous
Join Chat
View Demo Notes
Copyright 2009 ICEsoft Technologies Inc.
Powered By ICEfaces

Programming model

If you take a look at the programming model of the ICEfaces AJAX Push, you will find it quite simple to use. Here are the steps:

1. Define a group label that is used to describe the group of users and the informational context respectively.
2. Add your session to the `SessionRenderer` using this group label.
3. Initialize a rendering with the `SessionRenderer` using the group label if the informational context changes.

Download at WoweBook.Com

A simplified code example looks like this:

```
package com.domain.push;
import com.icesoft.faces.async.render.SessionRenderer;
public class MyAJAXPush {
  private String renderGroup = "myAJAXPushGroup";
  public void init() {
    // do some initialization
    // ...
    SessionRenderer.addCurrentSession(this.renderGroup);
  }
  public void update() {
    // update the informational context
    // ...
    SessionRenderer.render(this.renderGroup);
  }
}
```

The `SessionRenderer` is used globally in your web application. So, it is also possible to initialize the update for a certain user group in another bean context. For this, we would define the user group label in our global constants.

Architecture

The use of the `SessionRenderer` is quite simple. This is possible because of the strong standardization and transparent behavior of the multicast rendering in the background. You are able to implement AJAX Push differently so that you get more control over rendering. For this, ICEfaces offers other renderers and interfaces that you can use for your backing bean implementations.

But the architecture behind AJAX Push in the lower layers is quite complex. This is because it not only inverts the JSF communication between a web browser and a web container using a different technology, but also circumvents a lot of limitations. We will not take a look at these details here. If you are interested in this, have a look at `http://www.icefaces.org/main/ajax-java/ajaxpush.iface`.

Download at WoweBook.Com

Configuration

Using AJAX Push in our project needs some special configurations. We have to take a look at the following:

- Communication mode between a web browser and a web container
- Scope for backing beans
- Push server infrastructure

Deployment descriptor

The `/src/main/webapp/WEB-INF/web.xml` file allows setting the communication mode between a web browser and a web container. We have to allow asynchronous communication and with it, the use of AJAX Push, as shown here:

```
<context-param>
  <param-name>
    com.icesoft.faces.synchronousUpdate
  </param-name>
  <param-value>false</param-value>
</context-param>
```

As AJAX Push is active by default, it is not really necessary to set this parameter explicitly. But it is a good idea to document that you use it this way for later maintenance purposes.

Spring scopes

We manage our backing beans via the Spring framework. Similar to the `faces-config.xml` definition for managed beans in JSF, Spring has a `scope` attribute. This is set with the bean definition in the Spring application context. For AJAX Push, we have to use the following scopes:

- Session
- Application

Here is an example of a `session` scoped bean:

```
<bean id="myAJAXPush" class="com.domain.push.MyAJAXPush"
  scope="session">
  <aop:scoped-proxy/>
</bean>
```

Download at WoweBook.Com

To avoid exceptions because of lost dependencies, we add a `scoped-proxy` definition. This allows us to have dependencies between different scopes, such as between session scope and application scope, without any disadvantages.

To realize an `application` scope, we omit the `scope` attribute definition. Spring is working in the application scope mode by default when deployed to a web container.

Meanwhile, there are additional frameworks for Spring environments that deliver a more fine-grained scope model (for example, the edoras framework). These allow a more efficient management of beans in the web container. If you use ICEfaces without Spring, you can use the ICEfaces extended request scope.

Push server

The communication between a web browser and a web container is very restrictive. Normally, you are allowed to use two communication channels in parallel per web browser. For a modern AJAX communication, this is limited in a lot of cases.

This limitation in channels is protocol dependent and also restricted on the client side through the web browser implementation. So, it is only possible to keep all communication within the existing limits, even if you actually need more channels.

With the ICEfaces push server—a separate WAR file—you get a tool that allows you to get rid of these limitations. It implements a global channeling on the server side. After the deployment of the push server, all other ICEfaces deployments recognize it automatically and use its channel management for their communication. This virtualized communication only needs a single permanent "physical" connection.

Production

In previous ICEfaces releases (before 1.8.x), there was no push server. This kind of implementation still works. But if you skip the push server, you may recognize inconsistencies in the web browser renderings from time to time. If you use multiple push applications in a single web container, it is recommended to deploy the push server in any case.

Download at WoweBook.Com

Deployment

The push server can be seen as an extra deployment for your projects. You deploy its WAR only once to the web container that you will use for your web application deployments. After that, your multiuser AJAX Push is ready to use.

You can find the WAR at `/icefaces/push-server` in the ICEfaces binary distribution.

ICEcube/ICEfusion

ICEcube and ICEfusion are Maven 2 projects. These allow deploying the project to a Jetty web container on the fly, without any extra installation. For the extra deployment of the push server, we use a trick. We:

- Define a Jetty configuration file for the push server WAR
- Let the WAR be a part of the project
- Let the Maven POM process this configuration

The WAR can be found at `/target-push/push-server-1.8.1.war`. The filename has an explicit release number to document which release is in use. Expect a strong dependency with the next releases because AJAX Push is under development, even if it is production ready. It is important that you exchange this file if you change the ICEfaces dependencies in the POM later.

Here is the Jetty configuration file " (`/jetty.xml`):

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC
  "-//Mort Bay Consulting//DTD Configure//EN"
  "http://jetty.mortbay.org/configure.dtd">
<Configure id="Server" class="org.mortbay.jetty.Server">
  <Set name="handler">
    <New id="Handlers"
      class="org.mortbay.jetty.handler.HandlerCollection">
      <Set name="handlers">
        <Array type="org.mortbay.jetty.Handler">
          <Item>
            <New id="WebHandler"
              class="org.mortbay
                .jetty.webapp.WebAppContext" />
          </Item>
        </Array>
      </Set>
    </New>
```

Download at WoweBook.Com

```
    </Set>
    <Ref id="WebHandler">
      <Set name="contextPath">/push-server</Set>
      <Set name="war">target-push/push-server-1.8.1.war</Set>
    </Ref>
  </Configure>
```

You may also have to adapt the `WebHandler` definition if you exchange the push server WAR.

The POM is using the Jetty configuration like this (`/pom.xml`):

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.9</version>
  <configuration>
    <jettyConfig>jetty.xml</jettyConfig>
    <contextPath>/</contextPath>
    <scanIntervalSeconds>3</scanIntervalSeconds>
    <scanTargetPatterns>
      <scanTargetPattern>
        <directory>src/main/webapp/WEB-INF</directory>
        <excludes>
          <exclude>**/*.jsp</exclude>
          <exclude>**/*.xhtml</exclude>
        </excludes>
        <includes>
          <include>**/*.properties</include>
          <include>**/*.xml</include>
        </includes>
      </scanTargetPattern>
    </scanTargetPatterns>
  </configuration>
</plugin>
```

This is also the original Jetty plugin definition of the AppFuse code ICEfusion is based on. It was only extended with a single line.

ICEcube creates a log file (`/icecube.log`) that allows checking if the push server is deployed. Each web application logs the probe to connect to the push server. If the push server is not deployed, you get a message like this:

```
[icecube] 2009-06-24 08:29:48,765 DEBUG [http-8080-2] HttpAdapter.
publish(239) | Outgoing message:
source_servletContextPath: /icecube
```

Download at WoweBook.Com

```
message_type: Presence
destination_servletContextPath: push-server
source_nodeAddress: 192.168.56.1
Hello
[icecube] 2009-06-24 08:29:48,796 WARN [http-8080-2] MainServlet.
setUpMessageServiceClient(285) | Push Server not found - the Push
Server must be deployed to support multiple asynchronous applications.
[icecube] 2009-06-24 08:29:48,796 INFO [http-8080-2] MainServlet.
setUpMessageServiceClient(291) | Adapting to Push environment.
```

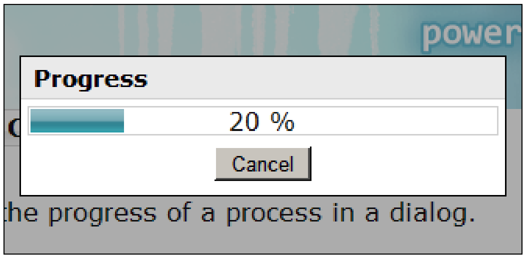
The ProgressDialog tag

The `ProgressDialog` tag follows the principles of the ICEfusion dialog tags implementation. But there is a difference in the kind of information delivery during the runtime. A progress component describes the percentage of work that is done in the context of an ongoing process. For this, the progress component uses numbers and/or a graphics bar.

The real-time presentation of the actual percentage needs direct communication between the process and the visual presentation panel, namely, the web browser. This is a classic Observer-Observable pattern, where the browser is the Observer. So, the web browser needs a direct link to the Observable that allows it to recognize when to update its presentation.

One challenge with the classic request-response model is that we do not have a direct and active link between Observer (a web browser) and the Observable (the process in a web container). Here, AJAX Push helps to provide such a link. So, we can use a `SessionRenderer` to push the updates to the progress component that is shown inside the `ProgressDialog`.

Here is an example of how this looks (see Push | Progress Dialog in the ICEcube menu):



Download at WoweBook.Com

The tag can be used like this:

```
<icefusion:progressDialog title="my_title" />
```

The `title` attribute is optional. The screenshot shows a non-set `title`.

The corresponding ICEcube demo looks like this (`/src/main/webapp/icecube/push/popupProgressDialog.xhtml`):

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/
DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.push.popupProgressDialog']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.push.popupProgressDialog
    .text']}
    <ice:form>
        <ice:commandButton value="#{icecube[
          'application.menu.push.popupProgressDialog
            .button']}"
          action="#{popupProgressDialog.progress}"/>
    </ice:form>

    <icefusion:progressDialog />
    </ui:define>
</ui:composition>
</body>
</html>
```

If you click on the **Start Process** button, the backing bean prepares the process that controls the updates in the progress bar. The backing bean also initiates the rendering of the `progressDialog`.

Download at WoweBook.Com

The backing bean looks like this (`/src/main/java/com/googlecode/icecube/push/PopupProgressDialog.java`):

```
package com.googlecode.icecube.push;
import org.springframework.beans.factory.annotation
  .Autowired;
import org.springframework.beans.factory.annotation
  .Qualifier;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.push
  .IProgressDialog;
import com.googlecode.icefusion.ui.commons.push
  .ProgressDialog;
public class PopupProgressDialog extends BackingBeanForm
  implements IProgressDialog {
    @Autowired
    @Qualifier("progressDialog")
    private ProgressDialog progressDialog;
    Long progress = 0L;
    Boolean cancel = false;
    public Long getProgress() {
      //The progress dialog is waiting one second before
      //the next update is processed. So, with this
      //increment we have a 10 seconds presentation.
        this.setProgress(this.progress + 10);
        return this.progress;
    }
    public String progress() {
        this.setProgress(0L);
        this.progressDialog.startProcess(this);
        return null;
    }
}
```

Each process has to implement the `IProgressDialog` interface that:

- Defines the progress exchange from the process to the dialog
- Defines how to react on a **Cancel** button event in the dialog

For this, we have a dependency injection, `@Autowired`, between the backing bean of the `ProgressDialog` component and our process implementation. Additionally, the `progress()` event to open the dialog sets a reference of the process back to the backing bean of the dialog. Finally, we can communicate in both directions.

Download at WoweBook.Com

The `IProgressDialog` interface can be found at `/src/main/java/com/googlecode/icefusion/ui/commons/push/IProgressDialog.java`:

```
package com.googlecode.icefusion.ui.commons.push;
import java.io.Serializable;
public interface IProgressDialog extends Serializable {
    public Long getProgress();
    public void setProgress(Long progress);
    public Boolean getCancel();
    public void setCancel(Boolean cancel);
}
```

The backing bean of the `ProgressDialog` looks like this (`/src/main/java/com/googlecode/icefusion/ui/commons/push/ProgressDialog.java`):

```
package com.googlecode.icefusion.ui.commons.push;
import java.io.Serializable;
import com.googlecode.icefusion.ui.commons.dialog.Dialog;
import com.icesoft.faces.async.render.SessionRenderer;
public class ProgressDialog extends Dialog implements Serializable {
    private Long progress = 0L;
    private IProgressDialog process;
    private String renderGroup = "progressbar";
    private Boolean cancel = false;
    private Boolean ready = false;
    public void startProcess(IProgressDialog process) {
        this.process = process;
        this.setReady(false);
        this.setCancel(false);
        this.setProgress(0L);
        this.setShow(true);
        SessionRenderer.addCurrentSession(this.renderGroup);
    }
    public String buttonOk() {
        this.setShow(false);
        return null;
    }
}
```

Download at WoweBook.Com

The backing bean shows the basic structure of the `SessionRenderer` usage we discussed at the beginning of this chapter. Its initialization is done by the `startProcess()` method. This method is called by the backing bean of the ICEcube demo. The `buttonOK()` method manages the button event of the dialog.

The backing bean has an inner class that works as a thread, as shown here:

```
import edu.emory.mathcs.backport.java.util.concurrent
  .LinkedBlockingQueue;
import edu.emory.mathcs.backport.java.util.concurrent
  .ThreadPoolExecutor;
import edu.emory.mathcs.backport.java.util.concurrent
  .TimeUnit;
public class ProgressDialog extends Dialog implements
  Serializable {
  protected static ThreadPoolExecutor threadPool =
    new ThreadPoolExecutor(5, 15, 30, TimeUnit.SECONDS,
    new LinkedBlockingQueue(20));
    public void startProcess(IProgressDialog process) {
        threadPool.execute(new updateThread());
    }
    public String buttonCancel() {
        this.setCancel(true);
        this.setShow(false);
        return null;
    }
    // Inner class
    protected class updateThread implements Runnable {
        private Long last = 0L;
        private Long current = 0L;
        public void run() {
            while ((last < 100) && !cancel) {
                this.current = process.getProgress();
                if (this.current > this.last) {
                    last = current;
                    // prepare output
                    progress = current;
                    SessionRenderer.render(renderGroup);
                }
                try {
                    Thread.sleep(500);
```

Download at WoweBook.Com

```
                        if (cancel) {
                            break;
                        }
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        // ignore
                    }
                }
                if (!cancel) {
                    ready = true;
                } else {
                    process.setCancel(cancel);
                }
                SessionRenderer.render(renderGroup);
            }
        }
    }
```

The thread is initialized by `startProcess()` and is stopped early by `buttonCancel()` by changing the `cancel` attribute. Inside the thread, the current progress (that is set by the demo backing bean in our example) is tracked on a regular basis. If it is increased, an AJAX Push rendering is initiated. If the progress has reached 100%, the **OK** button is rendered by setting the `ready` attribute for the presentation . The code for this is shown as follows (`/src/main/webapp/icefusion/taglibs/commons/push/progressDialog.xhtml`).

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:icefusion=
    "http://icefusion.googlecode.com/icefusion">
<body>
<ui:component>
  <ice:form>
    <ice:panelPopup autoCentre="false" draggable="false"
      modal="true" rendered="#{progressDialog.show}"
      visible="#{progressDialog.show}">
      <f:facet name="header">
        <ice:panelGrid>
          <ice:outputText value="#{(not empty title) ?
            title : ((not empty progressDialog.title) ?
            progressDialog.title : icefusion[
```

Download at WoweBook.Com

```
                'application.dialog.progress.title'])}"/>
            </ice:panelGrid>
          </f:facet>
          <f:facet name="body">
            <ice:panelGrid>
              <ice:outputProgress value=
                "#{progressDialog.progress}"
                indeterminate="false"/>
              <ice:panelGrid columns="1"
                styleClass="icePanelPopupButtons">
                <ice:commandButton value=
                  "#{icefusion['application.dialog.
                    progress.button.cancel']}"
                  action="#{progressDialog.buttonCancel}"
                  styleClass="icePanelPopupButton"
                  rendered="#{!progressDialog.ready}"/>
                <ice:commandButton value="#{icefusion[
                  'application.dialog.progress.button.ok']}"
                  action="#{progressDialog.buttonOk}"
                  styleClass="icePanelPopupButton"
                  rendered="#{progressDialog.ready}"/>
              </ice:panelGrid>
            </ice:panelGrid>
          </f:facet>
        </ice:panelPopup>
      </ice:form>
    </ui:component>
    </body>
    </html>
```

The separate thread in the backing bean allows having a separation of concerns. The underlying process increases the progress, but it does not have to worry about the presentation. The `progressDialog` is indeed independent from the implementation of the process and can be kept reusable.

The `progressDialog` is an example of a kind of local or single-user usage of the AJAX Push. Next, we will have a look at the ICEmapper game, which supports multiuser AJAX Push.

Download at WoweBook.Com

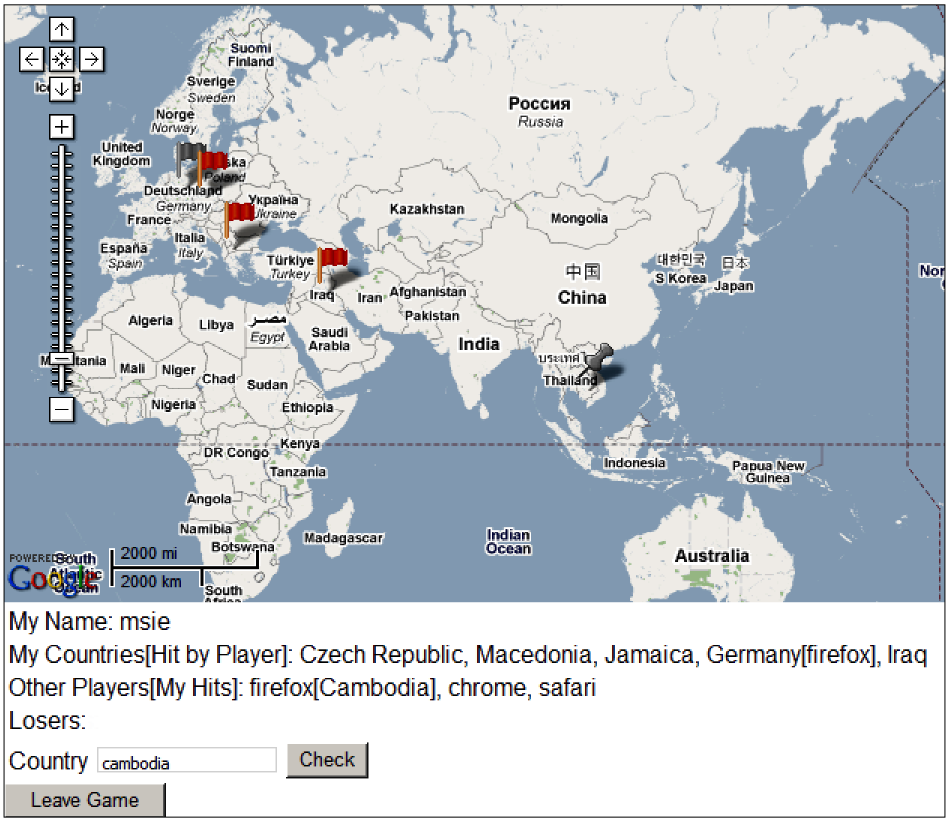
ICEmapper

You've already had a look at the auction and chat implementation in the ICEfaces examples. These examples are pretty cool and also present realistic use cases. The book example should follow this idea. But the aim was to create something fresh and new, and not just a poor copy of the existing stuff.

The first idea for such an implementation came up when I had a look at the concrete usage of Google Maps while studying its JavaScript APIs. To keep things simple, it finally became a game project: ICEmapper.

The basic idea is a mixture between Battleship® and Risk®. But instead of "Ships," we have invisibly-allocated countries to hit. And the "Army" of a country can be beaten immediately by naming the country.

Each player has five allocated countries, which are only shown in his own map using red markers (using a flag). If a country was hit, its marker changes to gray. All countries of other players that are hit are marked with a special gray marker, as shown here:


My Name: msie

My Countries[Hit by Player]: Czech Republic, Macedonia, Jamaica, Germany[firefox], Iraq

Other Players[My Hits]: firefox[Cambodia], chrome, safari

Losers:

Country cambodia Check

Leave Game

Download at WoweBook.Com

Dependent on the resolution of the map, you can recognize what was or wasn't hit. For a better overview, a list of the current status is shown beneath the map. Besides choosing a name before you start the game, you get a list of the following:

- The allocated countries you have (and the name of the player who hit a single country already)
- The list of other players (and their countries you already hit)
- The list of losers (players whose countries were all hit)

You can use the input field to type a country's name, or you can click on a country inside the map. However, there is one limitation with international users. Google Maps automatically uses the locale of the web browser to define the writing of the countries. So, a German browser, for example, uses the German writing of a country when you click on the map. But ICEmapper can only use English names for comparisons.

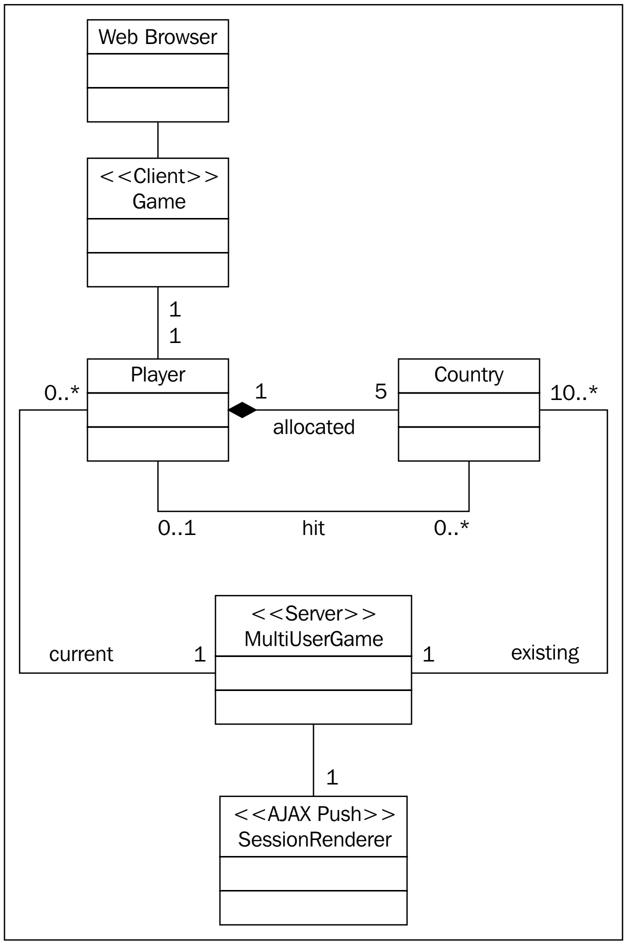
Google Maps only allows hardcoding of the locale it will use as a part of the `<script>` tag URL. But it does not support a change of the locale during runtime via JavaScript. The URL parameter does not work like expected. So, you may have to change your default web browser locale to English, or type the countries' names by hand. The input accepts and matches partial country names.

The application checks your country suggestion after clicking on the **Check** button. If you lose the game, the **Check** button is hidden. So, you have to click on **Leave Game** to become a part of another game.

Download at WoweBook.Com

Object model

ICEmapper implements a classic client-server model:



Each **Player** who corresponds to a **Web Browser** instance is managed by a **Game** client. It is implemented as a backing bean for the example page. The client primarily manages the **Country** relations, which we have seen beneath the map in the screenshot above.

To get a multiuser context, the **MultiUserGame** exists. It helps **Game** to describe contexts that exist between players. **MultiUserGame** is also used to initiate new players, or to describe the total of all countries.

Download at WoweBook.Com

Client side

We start by taking a look at the ICEcube example page that can be found at `/src/main/webapp/icecube/push/game.xhtml`. The code is separated into sections, each with its own description. Look at the following:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ice="http://www.icesoft.com/icefaces/component"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk"
  xmlns:icefusion="http://icefusion.googlecode.com/icefusion"
  xmlns:m="http://code.google.com/p/gmaps4jsf/">
<body>
<ui:composition template="#{iceFusionConsts.templatePage}">
  <ui:define name="title">
    #{icecube['application.menu.push.game']}
  </ui:define>
  <ui:define name="content">
    #{icecube['application.menu.push.game.text']}
    <!-- Use ICEfaces config for gmaps4jsf -->
    <style type="text/css">
      .iceGmpMapTd div.gmap {width: 0px; height: 0px}
          .iceGmpMapTd {visibility: hidden}
    </style>
    <ice:gMap/>
```

Although we do not use the ICEfaces Google Map component for presentation purposes, we added it to the code to use its initialization feature.

```
        <script>
            function clickedMap(marker, point) {
                if(!marker) {
                    var pointString=point.lat()+ "," +
                      point.lng();
                    var geocoder = new GClientGeocoder();
                    geocoder.getLocations(pointString,
                      calculateCountry);
                }
            }
            function calculateCountry(response) {
```

Download at WoweBook.Com

```
                var country;
                if (!response ||
                  response.Status.code != 200) {
                    country="None";
                }
                else {
                    if (response.Placemark[0]
                      .AddressDetails == undefined) {
                        country="None";
                    }
                    else {
                        country=response.Placemark[0]
                         .AddressDetails.Country
                           .CountryName;
                    }
                }
                document.getElementById(
                   "gmapGameForm:country").value=country;
            }
        </script>
```

The `clickedMap()` JavaScript method takes the position of a mouse click in the map and calculates a country name. The name is set in the **Country** edit field so that it can be used for the game to **Check**.

```
        <ice:form id="playerGameForm"
        rendered="#{game.player eq null}">
            <ice:panelGrid columns="3">
                <ice:outputText value="#{icecube[
                  'application.menu.push.game.player
                    .text']}" />
                <ice:inputText id="name" value="#{game.name}"
                  required="true"/>
                <ice:commandButton value="#{icecube[
                  'application.menu.push.game.player
                    .button']}"
                  action="#{game.addPlayer}" />
            </ice:panelGrid>
        </ice:form>
```

Download at WoweBook.Com

If the **Game** client is in its initialization phase, it shows a form that allows setting the player name in the **Your Name in the Game** edit field.

```
<ice:form id="gmapGameForm" rendered=
"#{game.player ne null and
  !game.playerNotAddedMessage.rendered}">
  <m:map id="gmapGame" jsVariable="gmapGameJS"
      width="930" height="400" zoom="2"
      autoReshape="true" type="G_NORMAL_MAP"
      renderOnWindowLoad="false">
      <ui:repeat var="country" value=
      "#{game.countriesMap}" >
            <m:marker address="#{country.key}">
                <m:icon imageURL="#{country.value}"
            width="28" height="28"/>
            </m:marker>
        </ui:repeat>
      <m:mapControl name="GLargeMapControl"
      position="G_ANCHOR_TOP_LEFT"/>
      <m:mapControl name="GScaleControl"
      position="G_ANCHOR_BOTTOM_LEFT"
      offsetWidth="70"/>
      <m:mapControl name="GMapTypeControl"
      position="G_ANCHOR_TOP_RIGHT"/>
      <m:eventListener eventName="click"
          jsFunction="clickedMap"/>
  </m:map>
```

We use an alternative Google Maps component here that is based on the GMaps4JSF implementation (`http://code.google.com/p/gmaps4jsf/`). Although the ICEfaces component delivers enough comfort to present a map, it lacks support for interactivity. ICEmapper uses the map to select countries. This cannot be implemented with the current ICEfaces component in a short time.

Interesting with the `<m:map>` tag is the support for event handling. The `<m:eventListener>` allows us to define the JavaScript function, `clickedMap()`, which handles the country selection and delivery to the edit field. It would have been possible to have an automatic check after a click on a country in the map. However, with the current implementation, we are also able to use a classic string input for a country check.

Download at WoweBook.Com

The output for the markers is given through `<ui:repeat>` and `<m:marker>`.
The backing bean delivers a `Map<Country_name,Image_url>` structure that handles all contexts. Which kind of image is shown for a certain country is calculated through `MultiUserGame`.

```
               <ice:panelGrid columns="1">
                   <ice:outputText value="#{icecube[
                 'application.menu.push.game.player.status
                     .name']}: #{game.name}" />
                   <ice:outputText value="#{icecube[
                     'application.menu.push.game.player
                       .status.countries']}:
                     #{game.countries}" />
                   <ice:outputText value="#{icecube[
                      'application.menu.push.game.player.status
                        .countries.hit']}:
                 #{game.countriesHit}" />
                   <ice:outputText value="#{icecube[
                       'application.menu.push.game.player.status
                       .losers']}: #{game.losers}" />
               </ice:panelGrid>
             <ice:panelGrid columns="3">
                   <ice:outputText value="#{icecube[
                     'application.menu.push.game.country
                        .text']}" />
                   <ice:inputText id="country" value=
                     "#{game.country}" />
                   <ice:commandButton value="#{icecube[
                     'application.menu.push.game.country
                       .button']}" action="#{game.checkCountry}"
                       rendered="#{!game.loser}"/>
             </ice:panelGrid>
             <ice:commandButton value="#{icecube[
               'application.menu.push.game.leave.button']}"
                   action="#{game.deletePlayer}" />
           </ice:form>
```

The status information shows the state of the game from the player's perspective.
A player can leave the game at any time using the **Leave Game** button.

```
           <icefusion:messageDialog title="#{icecube[
             'application.menu.push.game.player.message
             .notAdded.title']}" text="#{icecube[
             'application.menu.push.game.player.message
             .notAdded.text']}" rendered="#{game
```

Download at WoweBook.Com

```
              .playerNotAddedMessage.rendered}"
            eventBean="#{game.playerNotAddedMessage}"/>
          <icefusion:messageDialog title="#{icecube[
            'application.menu.push.game.player.message.hit
             .title']}" text="#{icecube['application
                .menu.push.game.player.message.hit.text']}"
          rendered="#{game.playerHitMessage.rendered}"
          eventBean="#{game.playerHitMessage}">
          </icefusion:messageDialog>
      </ui:define>
  </ui:composition>
  </body>
  </html>
```

The game uses two `messageDialogs` to inform the player about the following:

- The problem in letting him be a part of the game because of the absence of free countries
- A hit of a country of another player

Message handlers

The corresponding handler beans are implemented as inner classes of the backing bean for the example page. We will have a look at their implementation first. The code can be found at `/src/main/java/com/googlecode/icecube/push/game/Game.java`, as shown:

```
package com.googlecode.icecube.push.game;
import com.googlecode.icefusion.ui.commons.BackingBeanForm;
import com.googlecode.icefusion.ui.commons.dialog
  .IMessageDialog;
public class Game extends BackingBeanForm {
    private String name;
    // inner classes for messageDialogs
    public class PlayerNotAddedMessage implements
      IMessageDialog {
        private Boolean rendered = false;
        public String messageDialogButtonOk() {
            this.setRendered(false);
            setName(null);
            return null;
        }
    }
```

Download at WoweBook.Com

```
        public class PlayerHitMessage implements IMessageDialog {
            private Boolean rendered = false;
            public String messageDialogButtonOk() {
                this.setRendered(false);
                multiUserGame.push();
                return null;
            }
        }
    }
```

The `PlayerNotAddedMessage` inner class manages the situation when the `Player` cannot be added to an ongoing game. In this situation, the `Player` already has set a user `name` to use. This is reset in the `messageDialogButtonOK()` event handler.

The `PlayerHitMessage` manages a hit of a `Country` of another `Player`. In this situation, we have to initiate an AJAX Push rendering, so that all status messages in the current clients are updated. Although the rendering definitions are managed by `MultiUserGame`, we indeed leverage the global character of the AJAX Push and initiate the rendering outside of the context holder. For a better abstraction, `MultiUserGame` offers the `push()` method.

Button handlers

The example page offers different buttons to initiate a behavior (`/src/main/java/com/googlecode/icecube/push/game/Game.java`):

```
    package com.googlecode.icecube.push.game;
    import com.googlecode.icefusion.ui.commons.BackingBeanForm;
    public class Game extends BackingBeanForm {
        private String name;
        private Player player;
        private String country;
        public String addPlayer() {
            try {
                this.setPlayer(this.multiUserGame.addPlayer(
                this.name));
                // Maybe the name was adapted
                this.setName(this.getPlayer().getName());
                this.multiUserGame.push();
            } catch (NotEnoughCountriesException e) {
                this.getPlayerNotAddedMessage().setRendered(
                  true);
            }
            return null;
        }
```

Download at WoweBook.Com

```
        public String deletePlayer() {
            this.multiUserGame.deletePlayer(this.getPlayer());
            this.setPlayer(null);
            this.setName(null);
            this.multiUserGame.push();
            return null;
        }
        public String checkCountry() {
            Player playerHit = this.multiUserGame.checkCountry(
            this.getPlayer(), this.getCountry());
            if (playerHit != null) {
                this.playerHitMessage.setRendered(true);
                this.multiUserGame.push();
            }
            return null;
        }
    }
```

The `addPlayer()` method tries to add the `Player` to an ongoing game.
If this initialization works, an update of all clients has to be done. If the `NotEnoughCountriesException` is thrown, the corresponding `messageDialog` is activated.

`DeletePlayer()` manages the **Leave Game** button. It resets all settings of the `Player` and updates the status information of all the clients.

`CheckCountry()` takes the input of the current edit field for the `Country` and lets `MultiUserGame` check if there are any matches. If the `Player` realizes a hit, the corresponding `messageDialog` is activated and all of the clients get an update.

Status information

The status information that is shown beneath the map is calculated by `MultiUserGame`. So, `Game` only outputs the results (`/src/main/java/com/googlecode/icecube/push/game/Game.java`).

```
    package com.googlecode.icecube.push.game;
    import com.googlecode.icefusion.ui.commons.BackingBeanForm;
    public class Game extends BackingBeanForm {
        private Player player;
        public List<Map.Entry<String, String>>
          getCountriesMap() {
```

Download at WoweBook.Com

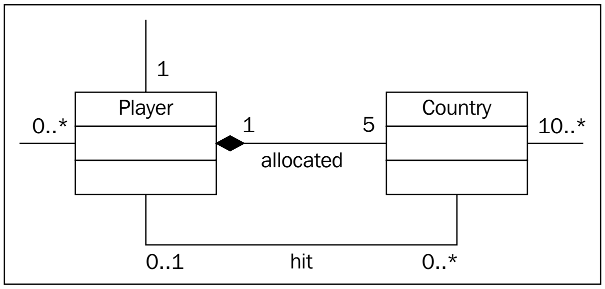
```
            return this.multiUserGame.getCountriesMap(
              this.getPlayer());
        }
        public String getCountries() {
            return this.multiUserGame.getCountries(
            this.getPlayer());
        }
        public String getCountriesHit() {
            return this.multiUserGame.getCountriesHit(
            this.getPlayer());
        }
        public String getLosers() {
            return this.multiUserGame.getLosers(
            this.getPlayer());
        }
    }
```

Server side

The server-side part of the implementation is used as player independent as well as session independent. The corresponding Spring bean does not define any scope (what corresponds to the `application` scope). This is necessary to keep and manage the status of the current game globally.

The implementation works with a dynamic list of players. So, it is possible to add or delete players during runtime without an influence on the general game behavior.

`MultiUserGame` manages calculations that are necessary to evaluate the relations between each `Player`. The data model describes these as shown in the next image:



Download at WoweBook.Com

The **allocated** composition defines the countries the **Player** possesses. A **Player** starts with five countries when he becomes a part of the game. The **hit** relation is added later and describes which of those countries get a relation with another **Player**. If all of his countries have such a relation, the **Player** has lost.

If we have a look at the status information again, the calculation looks like this:

- The list of allocated countries you have (and the name of the player who hit a single country already)
    - Take the list of my countries. Generate a list of names and add a player's name to it if a relation to another player exists.
- The list of other players (and their countries you already hit)
    - Have a look at the global players list and filter **Me**. Generate a list of names and a list of country names for each player when you find **Me** as a reference set for a country.
- The list of losers (players whose countries were all hit)
    - Take the list of global players. Generate a name if all countries of a player have a relation to another player set.

With this description, the loops in the corresponding methods should be easy to understand. The code for `MultiUserGame.java` is as shown (`/src/main/java/com/googlecode/icecube/push/game/MultiUserGame.java`).

```
public String getCountries(Player player) {
  String countriesList = "";
  int index = players.indexOf(player);
  if (index == -1) {
    return countriesList;
  }
  Map<String, Player> countries =
    players.get(index).getCountries();
  for (Entry<String, Player>country : countries.entrySet()) {
    countriesList += country.getKey();
    if (country.getValue() != null) {
      countriesList += "[" +
        country.getValue().getName() + "]";
    }
    countriesList += ", ";
  }
  String returnString = (countriesList.length() < 2) ?
    countriesList : countriesList.substring(0,
      countriesList.length() - 2);
  return returnString;
```

Download at WoweBook.Com

```
}
public String getCountriesHit(Player currentPlayer) {
  String hitList = "";
  for (Player player : players) {
    if (!player.equals(currentPlayer)) {
     hitList += player.getName();
      if (player.getCountries().values().contains(
        currentPlayer)) {
        hitList += "[";
        for (Entry<String, Player> country :
          player.getCountries().entrySet()) {
          if (country.getValue() != null &&
            currentPlayer.equals(country.getValue())) {
            hitList += country.getKey() + ", ";
          }
        }
        hitList = hitList.substring(0,
          hitList.length() - 2);
        hitList += "]";
      }
      hitList += ", ";
    }
  }
  return (hitList.length() < 2) ? hitList :
    hitList.substring(0, hitList.length() - 2);
}
public String getLosers(Player currentPlayer) {
  String losersList = "";
  for (Player player : players) {
    if (!player.equals(currentPlayer) &&
      !player.getCountries().values().contains(null)) {
      losersList += player.getName() + ", ";
    }
  }
  return (losersList.length() < 2) ? losersList :
    losersList.substring(0, losersList.length() - 2);
}
```

Download at WoweBook.Com

Summary

AJAX Push is a flexible and powerful tool to establish a server-side rendering infrastructure. It makes no difference if we use it in a single-user context such as the `progressDialog`, or in a multiuser context such as the ICEmapper game.

The use of the `SessionRenderer` is pretty easy. You can use it inside a single backing bean, or in a more global context when different beans initiate a rendering update alternately. In any case, the usage of the `SessionRenderer` is transparent.

With the end of this chapter, we have discussed all relevant aspects of the ICEfaces framework to implement desktop-like web applications. To use this know-how successfully in your project, have a deeper look at the implementation and configuration details of ICEcube/ICEfusion. The implementation allows you to use it both as a blueprint for a first prototype and for a concrete productive implementation.

Good luck!

Download at WoweBook.Com

Index

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Download at WoweBook.Com

Thank you for buying

ICEfaces 1.8: Next Generation Enterprise Web Development

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing ICEfaces 1.8: Next Generation Enterprise Web Development, Packt will have given some of the money received to the ICEfaces project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

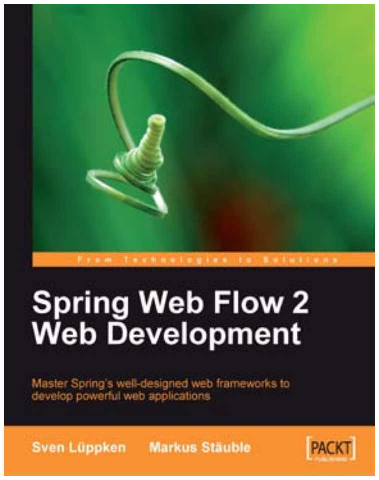
About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.PacktPub.com.

Download at WoweBook.Com


From Technologies to Solutions
Spring Web Flow 2 Web Development
Master Spring's well-designed web frameworks to develop powerful web applications
Sven Lüppken Markus Stäuble
PACKT

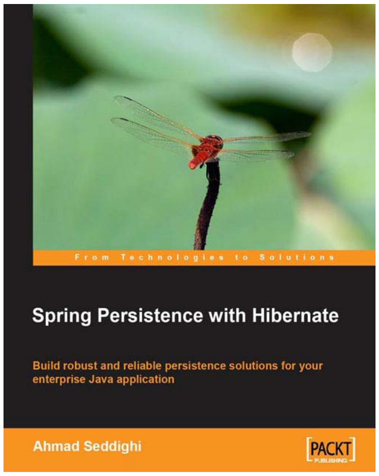
Spring Web Flow 2 Web Development

ISBN: 978-1-847195-42-5 Paperback: 272 pages

Master Spring's well-designed web frameworks to develop powerful web applications

1. Design, develop, and test your web applications using the Spring Web Flow 2 framework
2. Enhance your web applications with progressive AJAX, Spring security integration, and Spring Faces
3. Stay up-to-date with the latest version of Spring Web Flow

---


From Technologies to Solutions
Spring Persistence with Hibernate
Build robust and reliable persistence solutions for your enterprise Java application
Ahmad Seddighi
PACKT

Spring Persistence with Hibernate

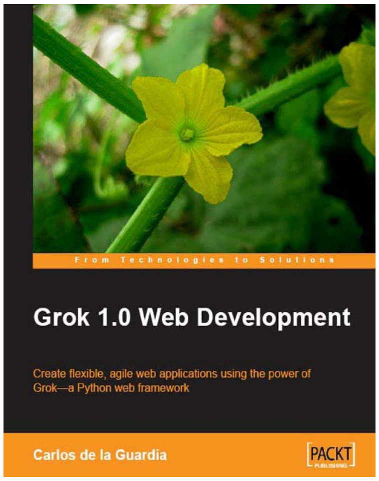
ISBN: 978-1-849510-56-1 Paperback: 340 pages

Build robust and reliable persistence solutions for your enterprise Java application

1. Get to grips with Hibernate and its configuration manager, mappings, types, session APIs, queries, and much more
2. Integrate Hibernate and Spring as part of your enterprise Java stack development
3. Work with Spring IoC (Inversion of Control), Spring AOP, transaction management, web development, and unit testing considerations and features

Please check **www.PacktPub.com** for information on our titles

Download at WoweBook.Com

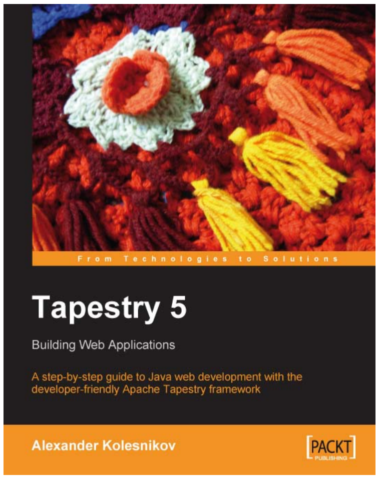


Grok 1.0 Web Development

ISBN: 978-1-847197-48-1   Paperback: 250 pages

Create flexible, agile web applications using the power of Grok—a Python web framework

1. Develop efficient and powerful web applications and web sites from start to finish using Grok, which is based on Zope 3
2. Integrate your applications or web sites with relational databases easily
3. Extend your applications using the power of the Zope Toolkit



Tapestry 5: Building Web Applications

ISBN: 978-1-847193-07-0   Paperback: 280 pages

A step-by-step guide to Java Web development with the developer-friendly Apache Tapestry framework

1. Latest version of Tapestry web development framework
2. Get working with Tapestry components
3. Gain hands-on experience developing an example site
4. Practical step-by-step tutorial

Please check **www.PacktPub.com** for information on our titles

Download at WoweBook.Com