



Community Experience Distilled

Learning Vaadin 7

Second Edition

Master the full range of web development features powered by Vaadin-built rich Internet applications

Foreword by Dr. Joonas Lehtinen, CEO and Founder, Vaadin

Nicolas Fränkel

[PACKT] open source*
PUBLISHING community experience distilled

Learning Vaadin 7

Second Edition

Master the full range of web development features
powered by Vaadin-built rich Internet applications

Nicolas Fränkel



open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Vaadin 7

Second Edition

Copyright © 2013 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, nor Packt Publishing, and its dealers and 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 of 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: October 2011

Second edition: September 2013

Production Reference: 1050913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-977-2

www.packtpub.com

Cover Image courtesy of www.public-domain-image.com

Credits

Author

Nicolas Fränkel

Project Coordinator

Wendell Palmer

Reviewers

Martin Cremer

Jonatan Kronqvist

Jouni Lehto

Tomek Lipski

Michael Vogt

Proofreader

Samantha Lyon

Indexer

Hemangini Bari

Graphics

Valentina Dsilva

Sheetal Aute

Disha Haria

Abhinash Sahu

Ronak Dhruv

Acquisition Editor

Kartikey Pandey

Lead Technical Editor

Madhuja Chaudhari

Production Coordinator

Nitesh Thakur

Technical Editors

Dipika Gaonkar

Dennis John

Mrunmayee Patil

Shali Sasidharan

Sonali Vernekar

Cover Work

Nitesh Thakur

Foreword

When we started designing Vaadin Framework in the year 2000 – then called Millstone Framework – we had a clear vision of creating a platform that would make building web applications fast, easy, and modular; something that we wanted to use by ourselves in the process of building business-oriented web applications. We envisioned a full stack of technologies starting from a web server, an object relationship mapping tool, a rich set of user interface components, and an extensible theme system. Everything built from scratch with a tiny team with no funding and little experience. Fortunately we did not have a clue about the size and complexity of the task or the lack of our experience – otherwise we would have never dared to start working on such a huge task. Finally, it took two years and three complete rewrites to understand the value of focusing solely on the user interface layer and being able to release something solid that has outgrown all the expectations we had.

Now when I look back at the design principles we chose for Vaadin, three principles in particular seem to have contributed to the longevity of the framework. First, we reasoned that the diversity and incompatibility of the web browsers we experienced back in the year 2000 was not going away – quite the contrary. While the Web is today the de facto platform for building all kinds of user interfaces, the capabilities of web browsers seem almost unlimited today and the number of web browsers has grown to include smartphones and tablets in addition to the handful of desktop browsers that should be supported by most applications. So we chose to embrace this complexity and abstract away from the browser to make it easier for developers to support "all" browsers at once. Secondly, we set our optimization target to be developer efficient, which in most cases can be roughly measured by the number of code lines in the user interface layer of the program. This has been a good choice as developers continue to be a more expensive resource in business application projects than servers. Finally, we recognized the need to support the heterogeneous teams where some developers might be more experienced than others. Some of the mechanisms to support the teams include theme packaging, multiple levels of abstraction, support for data bindings side-by-side with internal data in components, and deep inheritance hierarchies for user interface components to name a few.

Vaadin 7 has been the holy grail to our team for as long as I can remember. Somewhere after releasing IT Mill Toolkit 4 in 2007 we started dreaming of all the changes we should be making to the core of the framework to remove some of the shortcomings we felt it had. We started building up a huge backlog of things we should fix sometime soon when we have the time to do so and the courage to tear down parts of the old design. Then came IT Mill Toolkit 5 with the GWT-based client side and later on Vaadin 6. While they both included important features, they skipped many of the hard design choices in our "someday when we have the time" backlog. And we still believed that the time to do these would come really soon after the release of Vaadin 6. What happened was that we underestimated the success of Vaadin 6 and ended up releasing eight minor releases to it during 2009 to 2012. This took all of our focus and we pushed Vaadin 7 even further. Finally near the end of 2011, we decided that we cannot push Vaadin 7 any further and this would be the right time to make all the important changes we have been dreaming of.

While we thought we succeeded in cutting down the number of features in the release to an amount that it would be doable and fully finished in nine months, we ended up using almost twice that time and even then had to leave big things out from the release. But oh boy! We managed to get in over 60 new features, including a huge deal of merging Google Web Toolkit directly inside the Vaadin Framework. With this release, we also shifted the underlying philosophy of the framework by recognizing that all the three layers of abstraction that the framework implements should be equally accessible to the software developers: server-side Java, client-side compiled Java, and client-side JavaScript. This effectively changed Vaadin from being a server-side only framework that abstracts away from the Web to being a full stack framework that bridges Java and HTML5 platforms in a coherent way that supports the developers on all of its levels.

I have always been a huge fan of open source since being introduced to it by starting to play around with Linux kernel 0.3 and early Linux distributions. Working on, living in, and breathing open source did make it natural to choose to release Vaadin with an open source license and to build a community around it. After years of trying and failing to build an impactful community, all pieces finally clicked together in 2009 with the release of Vaadin 6. Seeing how people all over the world started to use Vaadin for building applications their businesses depended on for years to come had been great. What has been even more amazing is how people have started to contribute back to Vaadin—in terms of add-on components, helping each other on the forums, and promoting the framework to their peers. At the end of the day, a lively and friendly community and an ecosystem around Vaadin has been the key to the rapid growth of adoption.

I think that I first heard of Nicolas Fränkel by reading one of his many insightful blog posts some years back. I also remember him being one of the more active Vaadin community members helping others on the forum. At one time Nicolas invited me to a really nice dinner in Geneva where I was visiting the SoftShake conference to discuss about Vaadin and overeat the excellent Swiss fondue. During the dinner, we ended up talking about the need for a book that would tutor beginners through Vaadin and would introduce them to common patterns for Vaadin development. I remember getting contacted by Packt Publishing about getting in touch with potential authors for such a book. Nicolas had quite a lot of Vaadin experience and I asked if he would be interested in considering writing the book. To my surprise he agreed and the first edition of this book was born. Later on when Vaadin 7 was published, Nicolas decided to update the book to cover this newly released Version 7. I am sure that Nicolas underestimated the effort needed in writing about Vaadin 7 the same way as our team did while developing it. Maybe even for the same reason – the list of new things in Vaadin 7 is huge.

You might be familiar with *Book of Vaadin* – a free book about Vaadin. While being a very complete reference to Vaadin and anything related to it, the amount of content and the reference-like approach can make it overwhelming for a beginner. This book takes another approach. Instead of trying to be a reference, it teaches Vaadin concepts by introducing them one-by-one in an order natural for learning. It is written as a journey of building a simple Twitter client while learning the most important aspects of Vaadin – one-by-one.

In conclusion, I'd like to give my deep thanks to Nicolas for taking the challenge of writing this book, which I am sure, will help many people to get a quick start for writing Vaadin-based applications. I hope that these applications will benefit the companies investing in them as well as save a lot of time and frustration from the end users. But at the end of the day, it is most important to me – and I am sure that Nicolas shares this too – that you as a developer of those applications will save your time and frustration and be able to accomplish something that would not be possible otherwise.

Dr. Joonas Lehtinen
CEO and Founder, Vaadin

About the Author

Nicolas Fränkel operates as a successful Java/Java EE architect with more than 10 years' experience in consulting for different clients.

Based in France, he also practices (or has practiced) as a WebSphere Application Server administrator, a certified Valtech trainer, and a part-time lecturer in different French universities, so as to broaden his understanding of software craftsmanship.

His interests in IT are diversified, ranging from Rich Client Application, to Quality Processes through open source software and build automation. When not tinkering with new products or writing blog posts, he may be found practicing sports: squash, kickboxing, and skiing at the moment. Other leisure activities include reading novels, motorcycles, photography, and drawing, not necessarily in that order.

I'd like to thank the Vaadin team for all its glory, the product is awesome, guys! Keep up the good work. I would also like to thank my wonderful wife Corinne for letting me sin once again, this time in full understanding of the time it takes. I love you more after each passing year. My son Dorian, this is only the beginning.

About the Reviewers

Martin Cremer is working as an architect for a company in the financial sector. His work focuses on maintaining and developing reference architecture for web-based enterprise applications with Vaadin as well as supporting developers in their daily work.

Jonatan Kronqvist completed his M.Sc. and has been working with Vaadin Ltd., the company behind the Vaadin framework, since 2006. During this time, he has been a Vaadin consultant, a project manager, and a core developer of the Vaadin framework. Currently he spends his time focusing on add-ons and tools for easing development with Vaadin.

Before going full time on Vaadin, he has worked on many different projects ranging from advanced 3D graphics at a CAD software company to leading the development of a popular computer game for children.

Jouni Lehto has over 10 years' experience on different kind of web technologies and has been involved in a few projects where Vaadin has been the choice.

Tomek Lipski is an open source enthusiast and evangelist. He has over 16 years' commercial experience in IT, and 10 years' experience working in portal, enterprise integration, VAS, and traditional IT areas for the biggest companies in Central Europe.

In 2011, Lipski designed and coordinated an implementation and launch of Aperte workflow – an open source BPMS. Aperte workflow utilizes the OSGi plugin management system to provide flexible solutions combining several popular open source Java-based technologies, such as Vaadin, Liferay, and Activiti.

You can follow [@tomekripski](https://twitter.com/tomekripski) on Twitter or check out his blog at <http://blog.tomekripski.com>.

Michael Vogt started his career in 2000 at Apple, Germany as a WebObjects developer. Since then he has worked in many different companies and countries, mostly as a freelancer on GWT projects. Currently he works in the service department of Vaadin.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Vaadin and its Context	7
Rich applications	8
Application tiers	8
Tier migration	9
Limitations of the thin-client applications approach	11
Poor choice of controls	11
Many unrelated technologies	12
Browser compatibility	14
Page flow paradigm	14
Beyond the limits	15
What are rich clients?	15
Some rich client approaches	16
Why Vaadin?	20
State of the market	20
Importance of Vaadin	20
Vaadin integration	21
Integrated frameworks	21
Integration platforms	22
Using Vaadin in the real world	23
Concerns about using a new technology	23
Summary	25
Chapter 2: Environment Setup	27
Vaadin in Eclipse	27
Setting up Eclipse	28
When Eclipse is not installed	28
Installing the Vaadin plugin	30
Creating a server runtime	32
Creating our first Eclipse Vaadin project	32
Testing our application	35

Table of Contents

When Eclipse is already installed	36
Checking if WTP is present	36
Adding WTP to Eclipse	37
Vaadin in IntelliJ IDEA	39
Setting up IntelliJ	40
Adding the Vaadin 7 plugin	43
Creating our first IntelliJ IDEA Vaadin project	44
Adjusting the result	45
Adding framework support	46
Deploying the application automatically	46
Testing the application	47
Final touches	47
Changing the Vaadin version	47
Context-root	48
Servlet mapping	49
Vaadin and other IDEs	49
Adding Vaadin libraries	49
Creating the application	49
Adding the servlet mapping	50
Declaring the servlet class	51
Declaring Vaadin's entry point	51
Declaring the servlet mapping	51
Summary	52
Chapter 3: Hello Vaadin!	53
Understanding Vaadin	53
Vaadin's philosophy	54
Vaadin's architecture	55
Client-server communication	56
The client part	57
The server part	59
Client-server synchronization	60
Deploying a Vaadin application	60
Inside the IDE	61
Creating an IDE-managed server	61
Adding the application	63
Launching the server	63
Outside the IDE	65
Creating the WAR	65
Launching the server	65
Using Vaadin applications	66
Browsing Vaadin	66
Out-of-the-box helpers	66
The debug mode	67
Restart the application, not the server	69

Behind the surface	69
Stream redirection to a Vaadin servlet	69
Vaadin request handling	70
What does a UI do?	71
UI features	71
UI configuration	72
UI and session	72
Scratching the surface	73
The source code	74
The generated code	74
Things of interest	75
Summary	76
Chapter 4: Components and Layouts	77
Thinking in components	77
Terminology	78
Component class design	78
Component	79
MethodEventSource	80
Abstract client connector	80
Abstract component	80
UIs	81
HasComponents	82
Single component container	82
UI	83
Panel	83
Windows	84
Window structure	84
Customizing windows	85
Labels	86
Label class hierarchy	87
Property	87
Label	88
Text inputs	89
Conversion	90
Validation	92
Change buffer	96
Input	97
More Vaadin goodness	102
Page	102
Third-party content	104
User messages	106
Laying out the components	110
Size	110
Layouts	112
About layouts	112

Table of Contents

Component container	112
Layout and abstract layout	113
Layout types	113
Choosing the right layout	116
Split panels	117
Bringing it all together	118
Introducing Twaattin	118
The Twaattin design	118
The login screen	118
The main screen	118
Let's code!	118
Project setup	119
Project sources	119
Summary	122
Chapter 5: Event Listener Model	125
Event-driven model	125
The observer pattern	125
Enhancements to the pattern	126
Events in Java EE	127
UI events	128
Event model in Vaadin	129
Standard event implementation	129
Event class hierarchy	130
Listener interfaces	131
Managing listeners	133
Method event source details	133
Abstract component and event router	134
Expanding our view	135
Button	135
Events outside UI	136
User change event	136
Architectural considerations	137
Anonymous inner classes as listeners	138
Components as listeners	138
Presenters as listeners	139
Services as listeners	140
Conclusion on architecture	140
Twaattin is back	141
Project sources	141
Additional features	144
Summary	145

Table of Contents

Chapter 6: Containers and Related Components	147
Data binding	147
Data binding properties	148
Renderer and editor	148
Buffering	148
Data binding	149
Data in Vaadin	149
Entity abstraction	149
Property	149
Item	156
Container	168
Containers and the GUI	176
Container datasource	177
Container components	181
Tables	186
Trees	204
Refining Twaattin	205
Prerequisites	206
Adaptations	206
Sources	206
The login screen	207
The login behavior	208
The timeline screen	208
The tweets refresh behavior	210
Column generators	212
Summary	215
Chapter 7: Core Advanced Features	217
Accessing the JavaEE API	217
Servlet request	218
Servlet response	220
Wrapped session	222
Navigation API	222
URL fragment	223
Views	223
Navigator	224
Initial view	227
Error view	227
Dynamic view providers	227
Event model around the Navigation API	230
Final word on the Navigator API	230
Embedding Vaadin	230
Basic embedding	231

Table of Contents

Nominal embedding	232
Page headers	232
The div proper	232
The bootstrap script	233
UI initialization call	233
Real-world error handling	236
The error messages	236
Component error handling	237
General error handling	240
SQL container	244
Architecture	245
Features	246
Queries and connections	246
Database compatibility	248
Joins	253
References	254
Free form queries	256
Related add-ons	259
Server push	260
Push innards	261
Installation	262
How-to	262
Example	263
Twaattin improves!	265
Ivy dependencies	265
Twaattin UI	266
Tweet refresher behavior	268
Twitter service	269
Summary	269
Chapter 8: Featured Add-ons	271
Vaadin add-ons directory	271
Add-ons search	272
Typology	272
Stability	272
Add-ons presentation	273
Summarized view	273
Detailed view	273
Noteworthy add-ons	276
Button group	276
Prerequisites	276
Core concepts	276
How-to	279
Conclusion	281

Table of Contents

Clara	281
Prerequisites	282
How-to	282
Limitations	284
Conclusion	285
JPA Container	285
Concepts	286
Prerequisites	286
How-to	296
Conclusion	298
CDI Utils	298
Core concepts	299
Prerequisites	300
How-to	300
Conclusion	305
Summary	306
Chapter 9: Creating and Extending Components and Widgets	307
Component composition	307
Manual composition	308
Designing custom components	311
Graphic composition	311
Visual editor setup	311
Visual Designer use	312
Limitations	315
Client-side extensions	316
Connector architecture	316
How-to	318
Shared state	321
How-to	321
Server RPC	323
Server RPC architecture	323
How-to	324
GWT widget wrapping	326
Vaadin GWT architecture	326
How-to server-side	326
How-to client-side	326
Widget styling	328
Example	328
Prerequisites	329
Server component	329
Client classes	330
JavaScript wrapping	332
How-to	333

Table of Contents

Example	333
Prerequisites	334
Core	334
Componentized Twaattin	337
Designing the component	337
Updating Twaattin's code	338
Data Transfer Object	338
Status component	339
Status converter	341
Timeline screen	341
Summary	343
Chapter 10: Enterprise Integration	345
Build tools	345
Available tools	346
Apache Ant	346
Apache Maven	346
Fragmentation	347
Final choice	347
Tooling	347
Maven in Vaadin projects	348
Mavenize Vaadin projects	348
Vaadin support for Maven projects	349
Mavenizing Twaattin	353
Preparing the migration	353
Enabling dependency management	354
Finishing touches	354
Final POM	355
Portals	355
Portal, container, and portlet	355
Choosing a platform	356
Liferay	356
GateIn	357
Tooling	359
A simple portlet	359
Creating a project	359
Portlet project differences	359
Using the portlet in GateIn	363
Configuring GateIn for Vaadin	365
Themes and widgetsets	365
Advanced integration	367
Restart and debug	367
Handling portlet specifics	368
Portlet development strategies	372
Keep our portlet servlet-compatible	372

Table of Contents

Portal debug mode	372
Updating a deployed portlet	373
OSGi	374
Choosing a platform	375
Glassfish	376
Tooling	380
Vaadin OSGi use cases	380
Vaadin bundling	380
Modularization	381
Hello OSGi	381
Making a bundle	381
Export, deploy, and run	383
Correcting errors	383
Integrating Twaattin	385
Bundle plugin	385
Multiplatform build	388
Cloud	389
Cloud offering levels	389
State of the market	390
Hello cloud	391
Registration	391
Cloud setup	391
Application deployment	394
Summary	395
Index	397

Preface

Vaadin is a component-based Java web framework for making applications look great and perform well, making your users happy. Vaadin promises to make your user interfaces attractive and usable while easing your development efforts and boosting your productivity. After having read this book, you will be able to utilize the full range of development and deployment features offered by Vaadin while thoroughly understanding the concepts.

Learning Vaadin 7 Second Edition is a practical systematic tutorial to understand, use, and master the art of RIA development with Vaadin. You will learn about the fundamental concepts that are the cornerstones of the framework, at the same time making progress on building your own web application. The book will also show you how to integrate Vaadin with other popular frameworks and how to run it on top of internal, as well as externalized infrastructures.

This book will show you how to become a professional Vaadin developer by giving you a concrete foundation through diagrams, practical examples, and ready-to-use source code. It will enable you to grasp all the notions behind Vaadin one-step at a time: components, layouts, events, containers, and bindings. You will learn to build first-class web applications using best-of-breed technologies. You will find detailed information on how to integrate Vaadin's presentation layer on top of other widespread technologies, such as CDI and JPA. Finally, the book will show you how to deploy on different infrastructures, such as GateIn portlet container and Cloud platform Jelastic.

This book is an authoritative and complete systematic tutorial on how to create top-notch web applications with the RIA Vaadin framework.

What this book covers

Chapter 1, Vaadin and its Context, introduces Vaadin, its features, its philosophy, and its surrounding environment.

Chapter 2, Environment Setup, describes how to set up the development environment, whether using Eclipse or IntelliJ IDEA.

Chapter 3, Hello Vaadin!, creates a basic Vaadin project and explains what happens under the hood.

Chapter 4, Components and Layouts, presents simple building blocks for any Vaadin application worth its salt.

Chapter 5, Event Listener Model, illustrates the interactions between users and your application and how they are implemented in Vaadin.

Chapter 6, Containers and Related Components, explains not only components presenting beans collections, but also how they can be bound to the underlying data.

Chapter 7, Core Advanced Features, portrays advanced use-cases addressing real-life problems, such as using the Navigation API, running Vaadin applications inside legacy ones, error handling customization, data source binding, and the ever-popular server push.

Chapter 8, Featured Add-ons, lists some extra-features such as GUI declarative description through XML, client widgets integration, JPA and CDI integration. These are available with additional libraries known as add-ons.

Chapter 9, Creating and Extending Components and Widgets, details how to create new and extend existing server-side components and client-side widgets.

Chapter 10, Enterprise Integration, describes how to deploy Vaadin applications in other contexts commonly found in enterprise environment: portals such as JBoss GateIn, OSGi platforms using Glassfish and finally "the cloud" with provider Jelastic.

What you need for this book

In order to get the most out of this book, it's advised to have a computer, a Java Developer Kit 6/7 installed as well as an Internet access.

Helpful tools include a Java IDE, Eclipse or IntelliJ IDEA, the Tomcat servlet container, and the Glassfish Application Server.

Who this book is for

If you are a Java developer with some experience in Java web development and want to enter the world of Rich Internet Applications, then this technology and book are ideal for you. *Learning Vaadin 7 Second Edition* will be perfect as your next step towards building eye-candy dynamic web applications on the JVM.

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: "Every component is declared as an attribute and assigned a `@AutoGenerated` annotation."

A block of code is set as follows:

```
public class DisableOnClickButtonExtension extends AbstractExtension {  
  
    public DisableOnClickButtonExtension(String disabledLabel) {  
  
        getState().setDisabledLabel(disabledLabel);  
    }  
  
    ...  
}
```

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

```
@Connect(DisableOnClickButtonExtension.class)  
public class DisableOnClickButtonConnector extends  
AbstractExtensionConnector {  
  
    @Override  
    public DisableOnClickButtonSharedState getState() {  
  
        return (DisableOnClickButtonSharedState) super.getState();  
    }  
  
    ...  
}
```

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: "Now, just go the **File** menu and click on **New**".



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 e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, 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

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

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 improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. 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 website 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.

1

Vaadin and its Context

Developing Java applications and more specifically, developing Java web applications should be fun. Instead, most projects are a mess of sweat and toil, pressure and delays, costs and cost cutting. Web development has lost its appeal. Yet, among the many frameworks available, there is one in particular that draws our attention because of its ease of use and its original stance. It has been around since the past decade and has begun to grow in importance. The name of this framework is **Vaadin**. The goal of this book is to see, step-by-step, how to develop web applications with Vaadin.

 Vaadin is the Finnish word for a female reindeer (as well as a Finnish goddess). This piece of information will do marvels to your social life as you are now one of the few people on Earth who know this (outside Finland).

We are going to see Vaadin in detail in later chapters; the following is a preview of what it is:

- A component-based approach that really works, and provides a bunch of out-of-the-box components as well as extensions
- Full web compatibility, in addition to Google Web Toolkit
- All development is made completely in Java
- Most of the code can be run server-side, taking advantages of Java static typing, with the full power of dynamic update tools such as JRebel
- Integration with Eclipse and IntelliJ IDEA IDEs
- Available with no charge under a friendly Open Source Apache license and much, much more

Before diving right into Vaadin, it is important to understand what led to its creation. Readers who already have this information (or who don't care) should go directly to *Chapter 2, Environment Setup*.

In this chapter, we will look into the following:

- The evolution from mainframe toward the rich client.
 - The concept of application tier
 - The many limits of the thin-client approach
 - What stands beyond those limits
- Why choose Vaadin today?
 - The state of the market
 - Vaadin's place in the market
 - A preview of what other frameworks Vaadin can be integrated with and what platforms it can run on

Rich applications

Vaadin is often referred to as a **Rich Internet Application (RIA)** framework. Before explaining why, we need to first define some terms which will help us describe the framework. In particular, we will have a look at application tiers, the different kind of clients, and their history.

Application tiers

Some software run locally, that is, on the client machine and some run remotely, such as on a server machine. Some applications also run on both the client and the server. For example, when requesting an article from a website, we interact with a browser on the client side but the order itself is passed on a server in the form of a request.

Traditionally, all applications can be logically separated into tiers, each having different responsibilities as follows:

- **Presentation:** The presentation tier is responsible for displaying the end-user information and interaction. It is the realm of the user interface.
- **Business Logic:** The logic tier is responsible for controlling the application logic and functionality. It is also known as the application tier, or the middle tier as it is the glue between the other two surrounding tiers, thus leading to the term middleware.

- **Data:** The data tier is responsible for storing and retrieving data. This backend may be a file system. In most cases, it is a database, whether relational, flat, or even an object-oriented one.

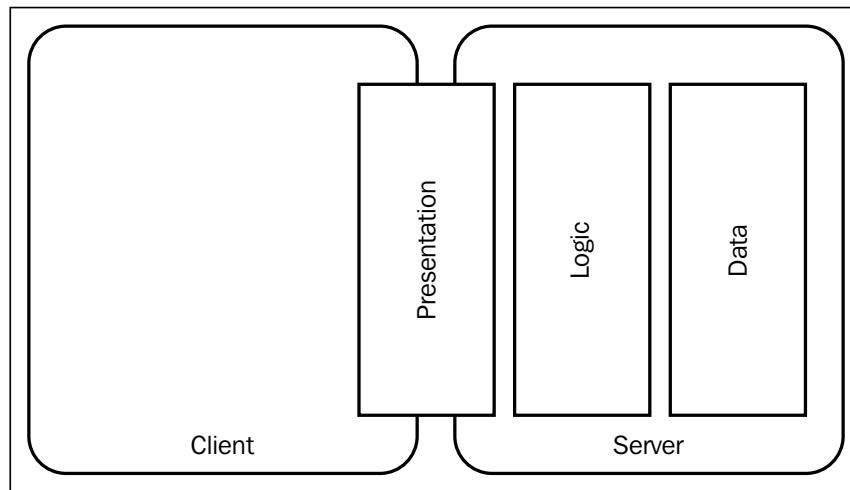
This categorization not only naturally corresponds to specialized features, but also allows you to physically separate your system into different parts, so that you can change a tier with reduced impact on adjacent tiers and no impact on non-adjacent tiers.

Tier migration

In the history of computers and computer software, these three tiers have moved back and forth between the server and the client.

Mainframes

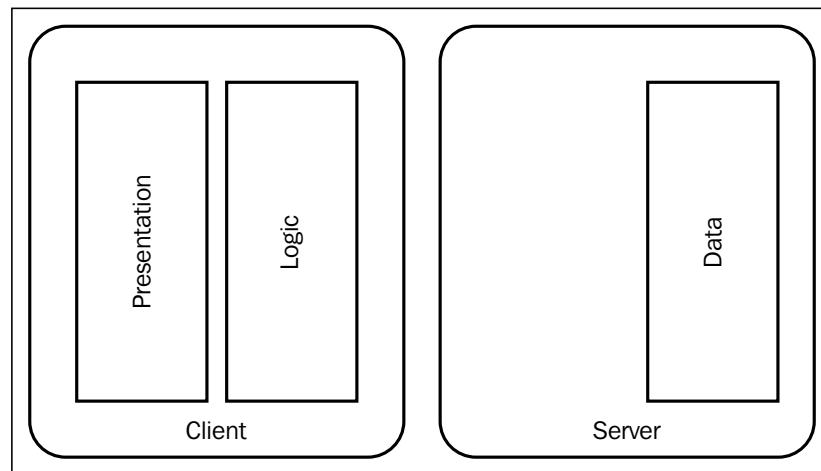
When computers were mainframes, all tiers were handled by the server. Mainframes stored data, processed it, and were also responsible for the layout of the presentation. Clients were dumb terminals, suited only for displaying characters on the screen and accepting the user input.



Client server

Not many companies could afford the acquisition of a mainframe (and many still cannot). Yet, those same companies could not do without computers at all, because the growing complexity of business processes needed automation. This development in personal computers led to a decrease in their cost. With the need to share data between them, the network traffic rose.

This period in history saw the rise of the personal computer, as well as the **Client server** term, as there was now a true client. The presentation and logic tier moved locally, while shared databases were remotely accessible, as shown in the following diagram:



Thin clients

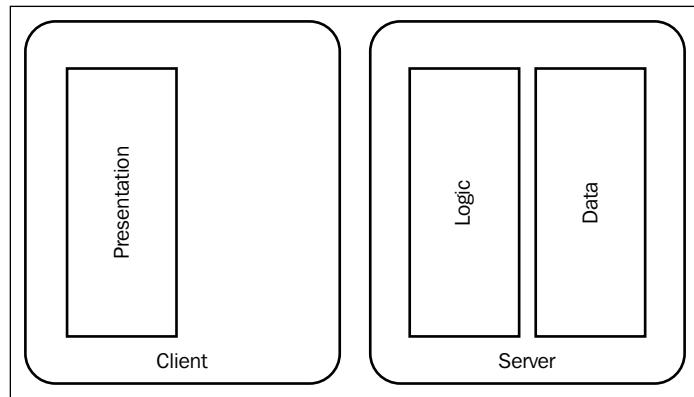
Big companies migrating from mainframes to client-server architectures thought that deploying software on ten client machines on the same site was relatively easy and could be done in a few hours. However, they quickly became aware of the fact that with the number of machines growing in a multi-site business, it could quickly become a nightmare.

Enterprises also found that it was not only the development phase that had to be managed like a project, but also the installation phase. When upgrading either the client or the server, you most likely found that the installation time was high, which in turn led to downtime and that led to additional business costs.

Around 1991, *Sir Tim Berners-Lee* invented the **Hyper Text Markup Language**, better known as **HTML**. Some time after that, people changed its original use, which was to navigate between documents, to make HTML-based web applications. This solved the deployment problem as the logic tier was run on a single-server node (or a cluster), and each client connected to this server. A deployment could be done in a matter of minutes, at worst overnight, which was a huge improvement. The presentation layer was still hosted on the client, with the browser responsible for displaying the user interface and handling user interaction.

This new approach brought new terms, which are as follows:

- The old client-server architecture was now referred to as **fat client**.
- The new architecture was coined as **thin client**, as shown in the following diagram:



Limitations of the thin-client applications approach

Unfortunately, this evolution was made for financial reasons and did not take into account some very important drawbacks of the thin client.

Poor choice of controls

HTML does not support many controls, and what is available is not on par with fat-client technologies. Consider, for example, the list box: in any fat client, choices displayed to the user can be filtered according to what is typed in the control. In legacy HTML, there's no such feature and all lines are displayed in all cases. Even with HTML5, which is supposed to add this feature, it is sadly not implemented in all browsers. This is a usability disaster if you need to display the list of countries (more than 200 entries!). As such, ergonomics of true thin clients have nothing to do with their fat-client ancestors.

Many unrelated technologies

Developers of fat-client applications have to learn only two languages: SQL and the technology's language, such as Visual Basic, Java, and so on.

Web developers, on the contrary, have to learn an entire stack of technologies, both on the client side and on the server side.

On the client side, the following are the requirements:

- First, of course, is HTML. It is the basis of all web applications, and although some do not consider it a programming language *per se*, every web developer must learn it so that they can create content to be displayed by browsers.
- In order to apply some common styling to your application, one will probably have to learn the **Cascading Style Sheets (CSS)** technology. CSS is available in three main versions, each version being more or less supported by browser version combinations (see *Browser compatibility*).
- Most of the time, it is nice to have some interactivity on the client side, like pop-up windows or others. In this case, we will need a scripting technology such as **ECMAScript**.

[ ECMAScript is the specification of which JavaScript is an implementation (along with **ActionScript**). It is standardized by the ECMA organization. See <http://www.ecma-international.org/publications/standards/Ecma-262.htm> for more information on the subject.]

- Finally, one will probably need to update the structure of the HTML page, a healthy dose of knowledge of the **Document Object Model (DOM)** is necessary.

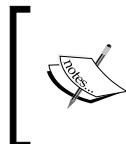
[ As a side note, consider that HTML, CSS, and DOM are W3C specifications while ECMAScript is an ECMA standard.]

From a Java point-of-view and on the server side, the following are the requirements:

- As servlets are the most common form of request-response user interactions in Java EE, every web developer worth his salt has to know both the Servlet specification and the Servlet API.

- Moreover, most web applications tend to enforce the **Model-View-Controller** paradigm. As such, the Java EE specification enforces the use of servlets for controllers and **JavaServer Pages (JSP)** for views. As JSP are intended to be templates, developers who create JSP have an additional syntax to learn, even though they offer the same features as servlets.
- JSP accept scriptlets, that is, Java code snippets, but good coding practices tend to frown upon this, however, as Java code can contain any feature, including some that should not be part of views – for example, the database access code. Therefore, a completely new technology stack is proposed in order to limit code included in JSP: the tag libraries. These tag libraries also have a specification and API, and that is another stack to learn.

However, these are a few of the standard requirements that you should know in order to develop web applications in Java. Most of the time, in order to boost developer productivity, one has to use frameworks. These frameworks are available in most of the previously cited technologies. Some of them are supported by Oracle, such as Java Server Faces, others are open source, such as **Struts**.



JavaEE 6 seems to favor replacement of JSP and Servlet by **Java Server Faces (JSF)**. Although JSF aims to provide a component-based MVC framework, it is plagued by a relative complexity regarding its components lifecycle.

Having to know so much has negative effects, a few are as follows:

- On the technical side, as web developers have to manage so many different technologies, web development is more complex than fat-client development, potentially leading to more bugs
- On the human resources side, different meant either different profiles were required or more resources, either way it added to the complexity of human resource management
- On the project management side, increased complexity caused lengthier projects: developing a web application was potentially taking longer than developing a fat-client application

All of these factors tend to make the thin-client development cost much more than fat-client, albeit the deployment cost was close to zero.

Browser compatibility

The Web has standards, most of them upheld by the World Wide Web Consortium. Browsers more or less implement these standards, depending on the vendor and the version. The ACID test, in version 3, is a test for browser compatibility with web standards. Fortunately, most browsers pass the test with 100 percent success, which was not the case two years ago.

Some browsers even make the standards evolve, such as Microsoft which implemented the `XMLHttpRequest` object in Internet Explorer and thus formed the basis for Ajax.

One should be aware of the combination of the platform, browser, and version. As some browsers cannot be installed with different versions on the same platform, testing can quickly become a mess (which can fortunately be mitigated with virtual machines and custom tools like <http://browsershots.org>). Applications should be developed with browser combinations in mind, and then tested on it, in order to ensure application compatibility.

For intranet applications, the number of supported browsers is normally limited. For Internet applications, however, most common combinations must be supported in order to increase availability. If this wasn't enough, then the same browser in the same version may run differently on different operating systems.

In all cases, each combination has an exponential impact on the application's complexity, and therefore, on cost.

Page flow paradigm

Fat-client applications manage windows. Most of the time, there's a main window. Actions are mainly performed in this main window, even if sometimes managed windows or pop-up windows are used.

As web applications are browser-based and use HTML over HTTP, things are managed differently. In this case, the presentation unit is not the window but the page. This is a big difference that entails a performance problem: indeed, each time the user clicks on a submit button, the request is sent to the server, processed by it, and the HTML response is sent back to the client.

For example, when a client submits a complex registration form, the entire page is recreated on the server side and sent back to the browser even if there is a minor validation error, even though the required changes to the registration form would have been minimal.

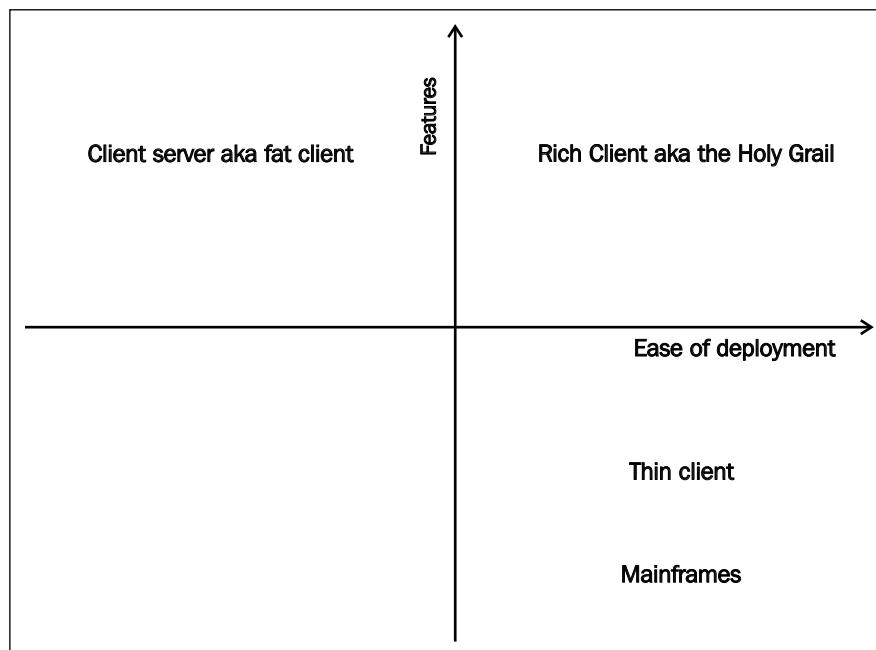
Beyond the limits

Over the last few years, users have been applying some pressure in order to have user interfaces that offer the same richness as good old fat-client applications. IT managers, however, are unwilling to go back to the old deploy-as-a-project routine and its associated costs and complexity. They push towards the same deployment process as thin-client applications. It is no surprise that there are different solutions in order to solve this dilemma.

What are rich clients?

All the following solutions are globally called rich clients, even if the approach differs. They have something in common though: all of them want to retain the ease of deployment of the thin client and solve some or all of the problems mentioned previously.

Rich clients fulfill the fourth quadrant of the following schema, which is like a dream come true, as shown in the following diagram:



Some rich client approaches

The following solutions are strategies that deserve the rich client label.

Ajax

Ajax was one of the first successful rich-client solutions. The term means **Asynchronous JavaScript with XML**. In effect, this browser technology enables sending asynchronous requests, meaning there is no need to reload the full page. Developers can provide client scripts implementing custom callbacks: those are executed when a response is sent from the server. Most of the time, such scripts use data provided in the response payload to dynamically update relevant part of the page DOM.

Ajax addresses the richness of controls and the page flow paradigm. Unfortunately:

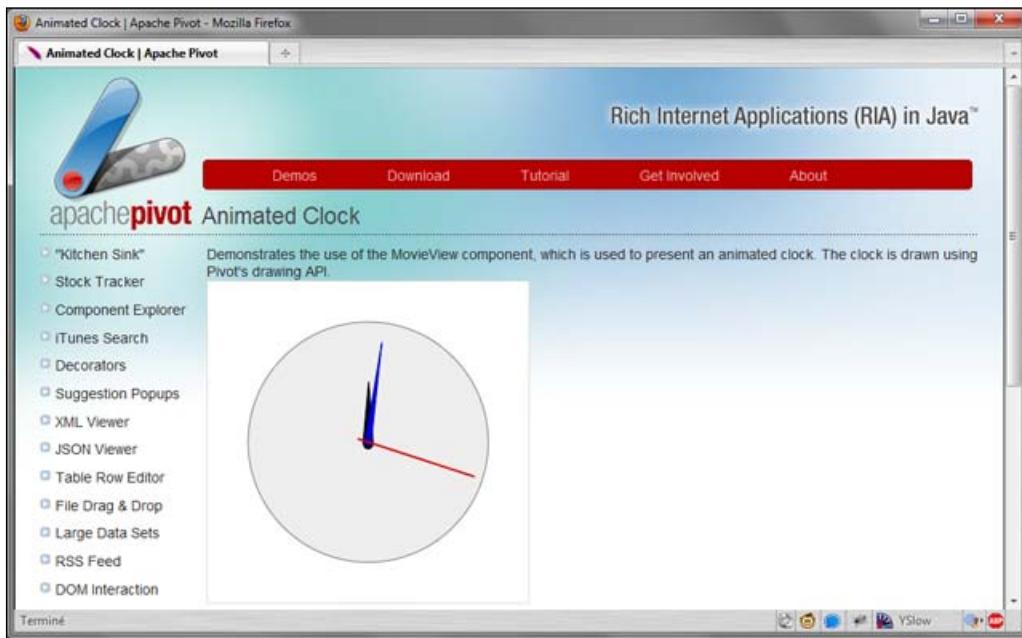
- It aggravates browser-compatibility problems as Ajax is not handled in the same way by all browsers.
- It has problems unrelated directly to the technologies, which are as follows:
 - Either one learns all the necessary technologies to do Ajax on its own, that is, JavaScript, Document Object Model, and JSON/XML, to communicate with the server and write all common features such as error handling from scratch.
 - Alternatively, one uses an Ajax framework, and thus, one has to learn another technology stack.

Richness through a plugin

The oldest way to bring richness to the user's experience is to execute the code on the client side and more specifically, as a plugin in the browser. Sun—now Oracle—proposed the applet technology, whereas Microsoft proposed **ActiveX**. The latest technology using this strategy is **Flash**.

All three were failures due to technical problems, including performance lags, security holes, and plain-client incompatibility or just plain rejection by the market.

There is an interesting way to revive the applet with the Apache Pivot project, as shown in the following screenshot (<http://pivot.apache.org/>), but it hasn't made a huge impact yet;



A more recent and successful attempt at executing code on the client side through a plugin is through Adobe's Flex. A similar path was taken by Microsoft's Silverlight technology.



Flex is a technology where static views are described in XML and dynamic behavior in ActionScript. Both are transformed at compile time in Flash format.

Unfortunately, Apple refused to have anything to do with the Flash plugin on iOS platforms. This move, coupled with the growing rise of HTML5, resulted in Adobe donating Flex to the Apache foundation. Also, Microsoft officially renounced plugin technology and shifted Silverlight development to HTML5.

Deploying and updating fat-client from the web

The most direct way toward rich-client applications is to deploy (and update) a fat-client application from the web.

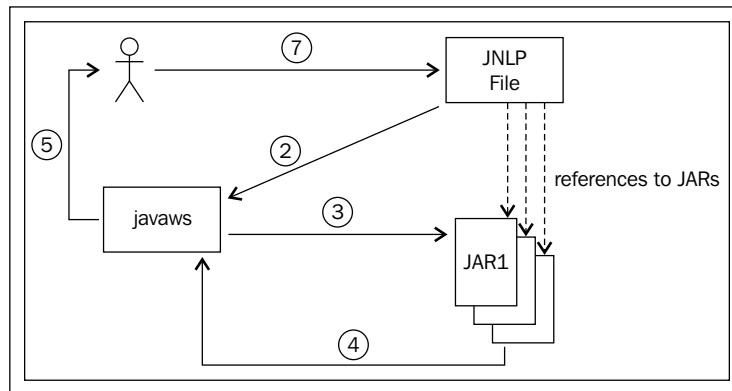
Java Web Start

Java Web Start (JWS), available at <http://download.oracle.com/javase/1.5.0/docs/guide/javaws/>, is a proprietary technology invented by Sun. It uses a deployment descriptor in **Java Network Launching Protocol (JNLP)** that takes the place of the manifest inside a JAR file and supplements it. For example, it describes the main class to launch the classpath, and also additional information such as the minimum Java version, icons to display on the user desktop, and so on.

This descriptor file is used by the `javaws` executable, which is bundled in the Java Runtime Environment. It is the `javaws` executable's responsibility to read the JNLP file and do the right thing according to it. In particular, when launched, `javaws` will download the updated JAR.

The detailed process goes something like the following:

1. The user clicks on a JNLP file.
2. The JNLP file is downloaded on the user machine, and interpreted by the local `javaws` application.
3. The file references JARs that `javaws` can download.
4. Once downloaded, JWS reassembles the different parts, create the classpath, and launch the main class described in the JNLP.



JWS correctly tackles all problems posed by the thin-client approach. Yet it never reaches critical mass for a number of reasons:

1. First time installations are time-consuming because typically lots of megabytes need to be transferred over the wire before the users can even start using the app. This is a mere annoyance for intranet applications, but a complete no go for Internet apps.
2. Some persistent bugs weren't fixed across major versions.
3. Finally, the lack of commercial commitment by Sun was the last straw.

A good example of a successful JWS application is **JDiskReport** (<http://www.jgoodies.com/download/jdiskreport/jdiskreport.jnlp>), a disk space analysis tool by *Karsten Lentzsch*, which is available on the Web for free.

Update sites

Updating software through update sites is a path taken by both **Integrated Development Environment (IDE)** leaders, NetBeans and Eclipse. In short, once the software is initially installed, updates and new features can be downloaded from the application itself.

Both IDEs also propose an API to build applications.

This approach also handles all problems posed by the thin-client approach. However, like JWS, there's no strong trend to build applications based on these IDEs. This can probably be attributed to both IDEs using the **OSGI** standard whose goal is to address some of Java's shortcomings but at the price of complexity.

Google Web Toolkit

Google Web Toolkit (GWT) is the framework used by Google to create some of its own applications. Its point of view is very unique among the technologies presented here. It lets you develop in Java, and then the GWT compiler transforms your code to JavaScript, which in turn manipulates the DOM tree to update HTML. It's GWT's responsibility to handle browser compatibility. This approach also solves the other problems of the pure thin-client approach.

Yet, GWT does not shield developers from all the dirty details. In particular, the developer still has to write part of the code handling server-client communication and he has to take care of the segregation between Java server-code which will be compiled into byte code and Java client-code which will be compiled into JavaScript. Also, note that the compilation process may be slow, even though there are a number of optimization features available during development. Finally, developers need a good understanding of the DOM, as well as the JavaScript/DOM event model.

Why Vaadin?

Vaadin is a solution evolved from a decade of problem-solving approach, provided by a Finnish company named Vaadin Ltd, formerly IT Mill.

Therefore, having so many solutions available, could question the use of Vaadin instead of Flex or GWT? Let's first have a look at the state of the market for web application frameworks in Java, then detail what makes Vaadin so unique in this market.

State of the market

Despite all the cons of the thin-client approach, an important share of applications developed today uses this paradigm, most of the time with a touch of Ajax augmentation.

Unfortunately, there is no clear leader for web applications. Some reasons include the following:

- Most developers know how to develop plain old web applications, with enough Ajax added in order to make them usable by users.
- GWT, although new and original, is still complex and needs seasoned developers in order to be effective.

From a Technical Lead or an IT Manager's point of view, this is a very fragmented market where it is hard to choose a solution that will meet users' requirements, as well as offering guarantees to be maintained in the years to come.

Importance of Vaadin

Vaadin is a unique framework in the current ecosystem; its differentiating features include the following:

- There is no need to learn different technology stacks, as the coding is solely in Java. The only thing to know beside Java is Vaadin's own API, which is easy to learn. This means:
 - The UI code is fully object-oriented
 - There's no spaghetti JavaScript to maintain
 - It is executed on the server side

- Furthermore, the IDE's full power is in our hands with refactoring and code completion.
- No plugin to install on the client's browser, ensuring all users that browse our application will be able to use it *as-is*.
- As Vaadin uses GWT under the hood, it supports all browsers that the version of GWT also supports. Therefore, we can develop a Vaadin application without paying attention to the browsers and let GWT handle the differences. Our users will interact with our application in the same way, whether they use an outdated version (such as Firefox 3.5), or a niche browser (like Opera).
- Moreover, Vaadin uses an abstraction over GWT so that the API is easier to use for developers. Also, note that Vaadin Ltd (the company) is part of GWT steering committee, which is a good sign for the future.
- Finally, Vaadin conforms to standards such as HTML and CSS, making the technology future proof. For example, many applications created with Vaadin run seamlessly on mobile devices although they were not initially designed to do so.

Vaadin integration

In today's environment, integration features of a framework are very important, as normally every enterprise has rules about which framework is to be used in some context. Vaadin is about the presentation layer and runs on any servlet container capable environment.

Integrated frameworks

A whole chapter (see *Chapter 9, Creating and Extending Components and Widgets*) is dedicated to the details of how Vaadin can be integrated with some third-party frameworks and tools. There are three integration levels possible which are as follows:

- **Level 1:** out-of-the-box or available through an add-on, no effort required save reading the documentation
- **Level 2:** more or less documented
- **Level 3:** possible with effort

The following are examples of such frameworks and tools with their respective integration estimated effort:

- **Level 1:**
 - **Java Persistence API (JPA):** JPA is the Java EE 5 standard for all things related to persistence. An add-on exists that lets us wire existing components to a JPA backend. Other persistence add-ons are available in the Vaadin directory, such as a container for Hibernate, one of the leading persistence frameworks available in the Java ecosystem.
 - A bunch of widget add-ons, such as tree tables, popup buttons, contextual menus, and many more.
- **Level 2:**
 - Spring is a framework which is based on **Inversion of Control (IoC)** that is the de facto standard for Dependency Injection. Spring can easily be integrated with Vaadin, and different strategies are available for this.
 - **Context Dependency Injection (CDI):** CDI is an attempt at making IoC a standard on the Java EE platform. Whatever can be done with Spring can be done with CDI. Given that CDI is a standard, we will have a look of its integration with Vaadin in *Chapter 8, Featured Add-ons*.
 - Any GWT extensions such as Ext-GWT or Smart GWT can easily be integrated in Vaadin, as Vaadin is built upon GWT's own widgets. This will be seen in complete detail in *Chapter 9, Creating and Extending Components and Widgets*, where we will create such new components.
- **Level 3:**
 - We can use another entirely new framework and languages and integrate them with Vaadin, as long as they run on the JVM: Apache iBatis, MongoDB, OSGi, Groovy, Scala, anything you can dream of!

Integration platforms

Vaadin provides an out-of-the-box integration with an important third-party platform: **Liferay** is an open source enterprise portal backed by Liferay Inc. Vaadin provides a specialized portlet that enables us to develop Vaadin applications as portlets that can be run on Liferay. Also, there is a widgetset management portlet provided by Vaadin, which deploys nicely into Liferay's Control Panel.

Using Vaadin in the real world

If you embrace Vaadin, then chances are that you will want to go beyond toying with the Vaadin framework and develop real-world applications.

Concerns about using a new technology

Although it is okay to use the latest technology for a personal or academic project, projects that have business objectives should just run and not be riddled with problems from third-party products. In particular, most managers may be wary when confronted by a new product (or even a new version), and developers should be too.

The following are some of the reasons to choose Vaadin:

- **Product is of highest quality:** The Vaadin team has done rigorous testing throughout their automated build process. Currently, it consists of more than 8,000 unit tests. Moreover, in order to guarantee full compatibility between versions, many (many!) tests execute pixel-level regression testing.
- **Support:**
 - **Commercial:** Although completely committed to open source, Vaadin Limited offer commercial support for their product. Check their Pro Account offering.
 - **User forums:** A Vaadin user forum is available. Anyone registered can post questions and see them answered by a member of the team or of the community.



Note that Vaadin registration is free, as well as hassle-free: you will just be sent the newsletter once a month (and you can opt-out, of course).

- **Retro-compatibility:**
 - **API:** The server-side API is very stable, version after version, and has survived major client-engines rewrite. Some part of the API has been changed from v6 to v7, but it is still very easy to migrate.
 - **Architecture:** Vaadin's architecture favors abstraction and is at the root of it all.

- **Full-blown documentation available:**
 - **Product documentation:** Vaadin's site provides three levels of documentation regarding Vaadin: a five-minute tutorial, a one-hour tutorial, and the famed *Book of Vaadin*.
 - **Tutorials**
 - **API documentation:** The Javadocs are available online; there is no need to build the project locally.
- **Course/webinar offerings:** Vaadin Ltd currently provides four different courses, which tackles all the needed skills for a developer to be proficient in the framework.
- **Huge community around the product:** There is a community gathering, which is ever growing and actively using the product. There are plenty of blogs and articles online on Vaadin. Furthermore, there are already many enterprises using Vaadin for their applications.
- **Available competent resources:** There are more and more people learning Vaadin. Moreover, if no developer is available, the framework can be learned in a few days.
- **Integration with existing product/platforms:** Vaadin is built to be easily integrated with other products and platforms. The Book of Vaadin describes how to integrate with Liferay and Google App Engine.

 [**Others already use Vaadin**
Upon reading this, managers and developers alike should realize Vaadin is mature and is used on real-world applications around the world. If you still have any doubts, then you should check <http://vaadin.com/who-is-using-vaadin> and be assured that big businesses trusted Vaadin before you, and benefited from its advantages as well.]

Summary

In this chapter, we saw the migration of application tiers in the software architecture between the client and the server.

We saw that each step resolved the problems in the previous architecture:

- Client-server used the power of personal computers in order to decrease mainframe costs
- Thin-clients resolved the deployment costs and delays

Thin-clients have numerous drawbacks. For the user, a lack of usability due to poor choice of controls, browser compatibility issues, and the navigation based on page flow; for the developer, many technologies to know.

As we are at the crossroad, there is no clear winner in all the solutions available: some only address a few of the problems, some aggravate them.

Vaadin is an original solution that tries to resolve many problems at once:

- It provides rich controls
- It uses GWT under the cover that addresses most browser compatibility issues
- It has abstractions over the request response model, so that the model used is application-based and not page based
- The developer only needs to know one programming language: Java, and Vaadin generates all HTML, JavaScript, and CSS code for you

Now we can go on and create our first Vaadin application!

2

Environment Setup

In this chapter, we will set up our IDE in order to ease the use of Vaadin and create new projects using this framework.

In particular, we will cover:

- Downloading and installing the right distribution of the IDE
- Checking that your currently installed IDE is the right distribution
- Installing the Vaadin plugin in your IDE
- Creating a new Vaadin project using our now enhanced IDE

The first section is dedicated to Eclipse from the Eclipse Foundation. The second section tackles **IntelliJ IDEA** installation and configuration for Vaadin development.

Depending on your own personal taste, you can go directly to your preferred section and ignore the other one, or browse both.

Finally, we will look at the configuration performed by the Vaadin plugin when we create a new Vaadin project if you want to configure your project in other IDEs or manually, since it creates a template project out-of-the-box.

Vaadin in Eclipse

In order to add Vaadin capabilities to Eclipse IDE, we will first need to have the **Web Tools Platform (WTP)**. Eclipse's WTP concerns itself with all that is centered on web standards in the Java ecosystem: servlets, JSP, HTML, JavaScript, and so on. As everything is a plugin in Eclipse, WTP itself is available as a collection of plugins.

Setting up Eclipse

If you already have Eclipse installed on your system, chances are that it already contains WTP and its dependencies. If not, you could start from scratch and install an Eclipse bundled with WTP (aka Eclipse for Java EE developers), or just have WTP and its dependencies added to your existing installation.

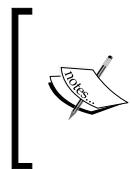


Note that the recommended method is to install the complete Java EE bundle, not only because it is recommended by Vaadin but also because manual handling of WTP dependencies can be a bore.



When Eclipse is not installed

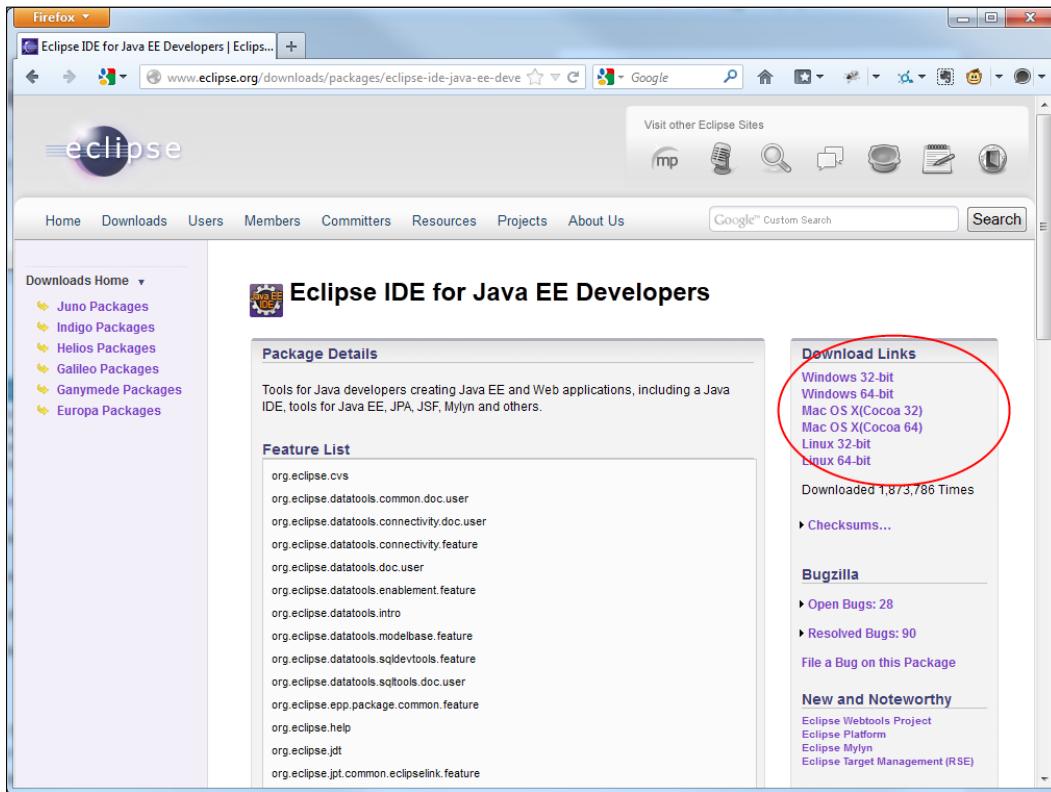
A more straightforward way to download an Eclipse bundled with WTP is to go to the Eclipse downloads website at <http://www.eclipse.org/downloads/> and choose Eclipse IDE for Java EE Developers. The exact URL changes with each Eclipse major release. At the time of this writing, it references the Juno (Eclipse 4.2) Service Release 2: <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/junosr2>.



Astute readers will note, and rightly so, that the "Eclipse IDE for Java EE Developers" distribution contains much more than simply the Eclipse IDE and WTP; EJB features for example. Although those features are unnecessary, it is the simplest way to have the WTP features.



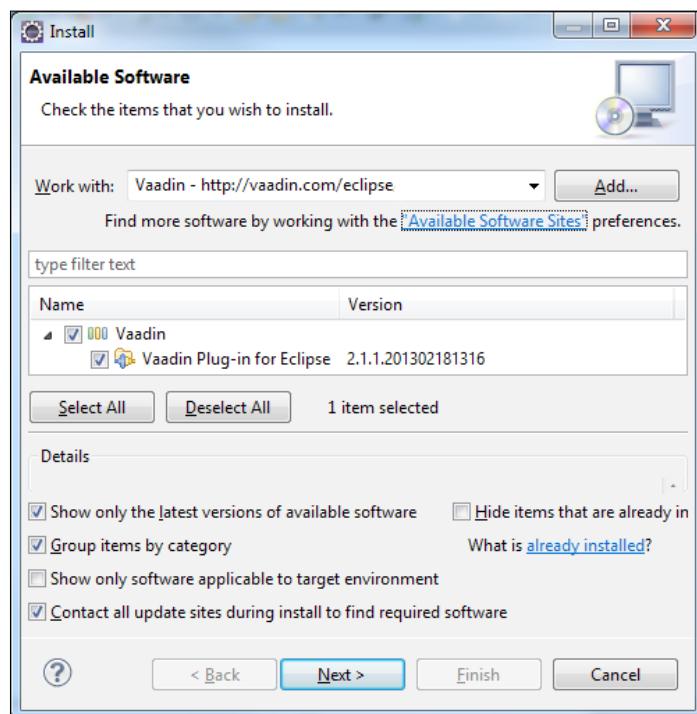
Now choose your OS carefully on the right panel of the screen and click on the proposed distribution site, as shown in the following screenshot:



Installing the Vaadin plugin

These are the steps we need to perform to install Vaadin:

1. In Eclipse, go to the **Help** menu and then click on **Install new software**.
2. Click on the **Add** button. You will be presented with a dialog prompting you for a name and an update site's location. Vaadin plugin manages library dependencies with Ivy. Just type `http://www.apache.org/dist/ant/ivyde/updatesite` in the location field. Repeat the operation with `http://vaadin.com/eclipse/`.
3. Click on the **Next** button and complete the wizard as follows:
 - Select the single Vaadin item. Don't forget to check **Contact all update sites during install to find required software**.



- Review the choices.
 - Accept the terms of the license agreement.
 - Finally, restart Eclipse in order to have the Vaadin features.
4. Now, we can check the installed features: go to the menu **Help | About Eclipse**. The opening pop up should display the Vaadin logo, as shown in the following screenshot:



Troubleshooting

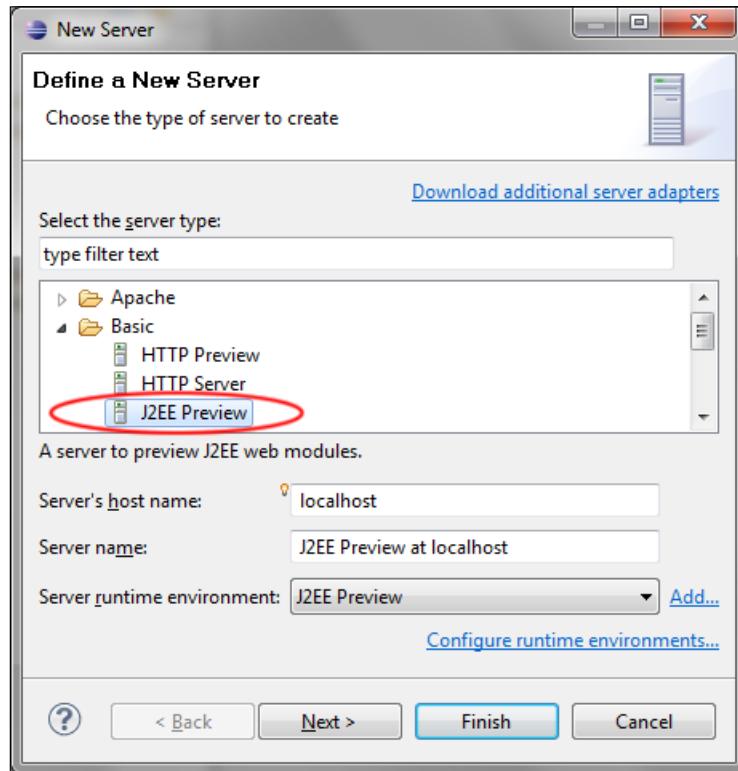
If Vaadin does not appear in the plugins list, restart Eclipse.

Congratulations, now we have completed the IDE setup. It is now time to create our first Vaadin project!

Creating a server runtime

Before creating the project itself, we need a server to run it on. Therefore, carry out the following steps:

1. On the **Server** tab, right-click and select **New Server**. The following screen should then pop up:



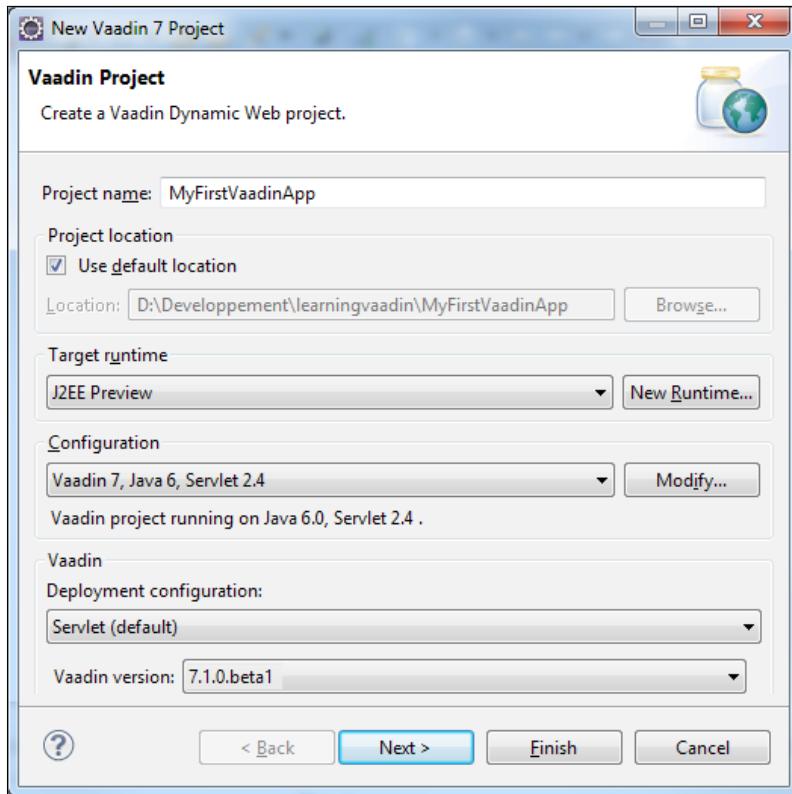
2. On the opening window, select **J2EE Preview**, as it is the simplest server Vaadin can run on. Click on **Finish**, and the newly created server should appear in the **Server** tab.

Creating our first Eclipse Vaadin project

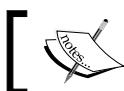
Carry out the following steps:

1. Go to the **File** menu and navigate to **New | Vaadin 7 Project**.
2. Set the project name as you wish-**MyFirstVaadinApp**, for example.

3. Displayed parameters should be kept as default.
4. Note that the Vaadin version should be set to the latest stable release by default. Since we chose Vaadin 7, it's 7.1.3 at the time of this writing.

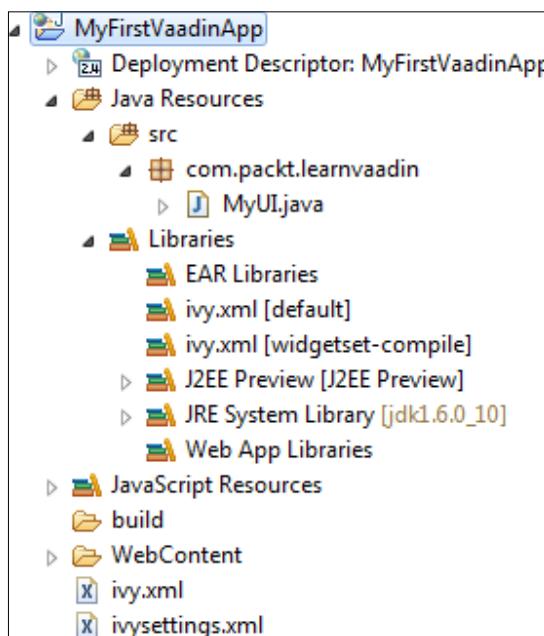


5. Click on the **Next** button. On the Java step, keep the source and the build folders as they are and click on **Next**.
6. On the Web Module step, change the context-root to `myfirstvaadinapp` and click on **Next**.



Context-roots are used to separate multiple web applications installed in the same servlet container and accessible on the same port.

7. In the last step, change the values as follow:
 - My First Vaadin Application for the Application name
 - com.packt.learnvaadin for the Base package name
 - MyUI for the Application class name
8. Finally, click on **Finish**. The project should look something similar to the one shown on the following screenshot, which displays the **Project Explorer** tab, when expanded:



Servlet mapping

There's one last action to complete our project. Open the `web.xml` deployment descriptor and search for `servlet-mapping`:

```
<servlet-mapping>
  <servlet-name>My First Vaadin Application</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

In order to be fully generic:

- Change the URL pattern from /* to /app/* as follows:

```
<servlet-mapping>
    <servlet-name>My First Vaadin Application</servlet-name>
    <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

- Add the /VAADIN/* mapping as follows:

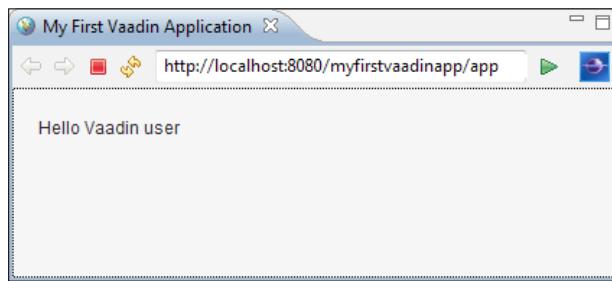
```
<servlet-mapping>
    <servlet-name>My First Vaadin Application</servlet-name>
    <url-pattern>/VAADIN/*</url-pattern>
</servlet-mapping>
```

The rationale behind these changes can be found later in the *Declaring the servlet mapping* section in this chapter.

Testing our application

Finally, select the project, right-click on the contextual menu, go to **Run As | Run on Server** and append /app at the end of the URL.

It should display a welcome message for us in Eclipse's internal browser, as shown in the following screenshot:



That is it. Congratulations, our first Vaadin project is running!

When Eclipse is already installed

If you already have Eclipse installed and want to use this existing installation, then the first thing to do is to check whether WTP is present, as it is a dependency of the Vaadin plugin.

Checking if WTP is present

In order to check whether an Eclipse installation has WTP, you have to launch it and go to menu **Help | About Eclipse**. The pop up that opens will show some information along with an icon list. Check the **wtp** icon, as can be seen in the following screenshot, to know whether WTP is installed:



If you have WTP installed, then head back to the section named *When Eclipse is not installed* in this chapter. Otherwise, please first follow the instructions in the following section.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

All relevant code as well as the full Twaattin source code is available online on GitHub at <https://github.com/nfrankel/Learning-Vaadin/>.

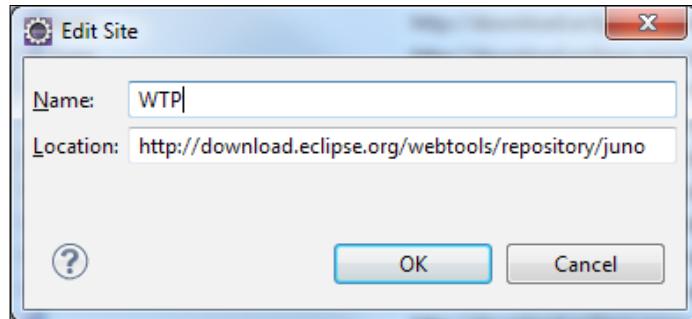
Adding WTP to Eclipse

In Eclipse, adding additional features to the core is done through a two-step process, which is as follows:

1. Add an update site, which is a URL that describes the deployment of additional components.
2. Use the update site to download and install them.

In order to add the WTP features to an existing Eclipse installation, we will first add an update site. As seen in the *Installing the Vaadin plugin* section in this chapter, go to the **Help** menu and then click on **Install new software** and carry out the following steps:

1. Click on the **Add** button. You will be presented with a dialog prompting you for a name and an update site's location (as shown in the following screenshot). So, type **WTP** in the first field and <http://download.eclipse.org/webtools/repository/juno/> in the second (the last part of the URL depending on the particular Eclipse release):

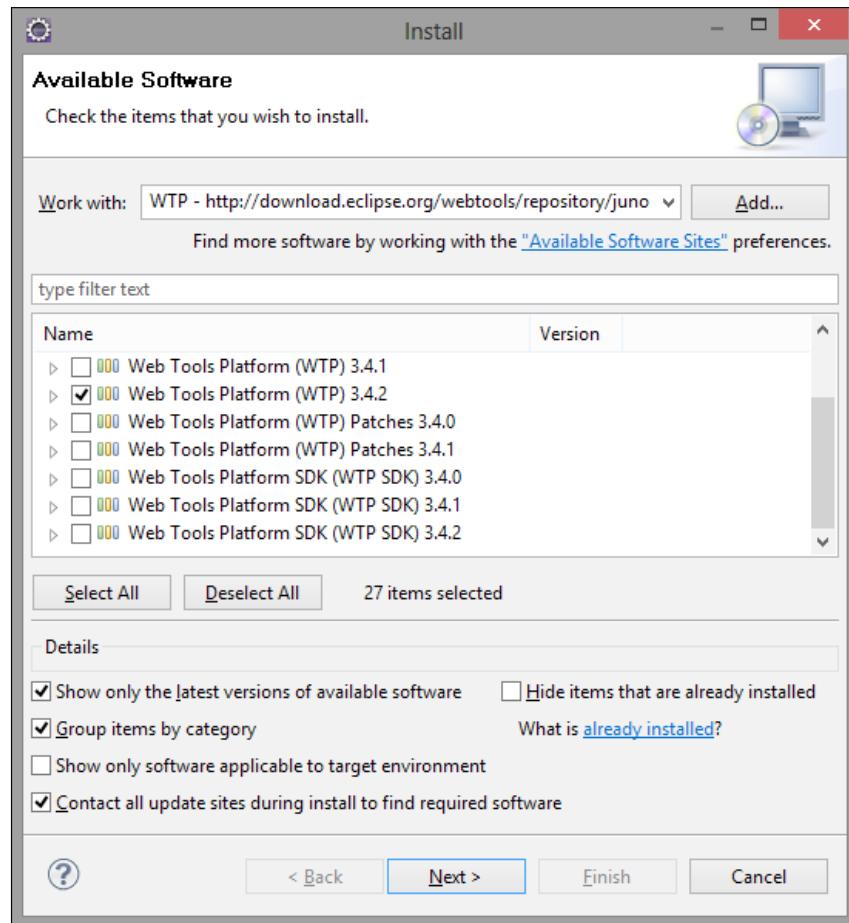


2. Clicking on **OK** will populate the preceding window. For the purpose of using Vaadin, we only need the latest version that does not include the SDK (at the time of this writing, it is Version 3.4.2).

Environment Setup



The SDK is only required when writing WTP plugins, which is not the case for us. The bare version is preferable as it is much lighter (and thus, faster to download).



Dependencies



Depending on the exact Eclipse installation and the precise WTP version, there may be some need to download additional WTP dependencies. As such, it is very important to select the option **Contact all update sites during install to find required software** to let Eclipse do just that.

3. Click on **Next** and complete the wizard as follows:
 - Review your choices
 - Accept the license agreement
 - Finally, restart Eclipse with our much wanted WTP features
4. Once restarted, you should see WTP installed as described in the previous *Checking if WTP is present* section.

Troubleshooting



If WTP still does not appear in the plugins list or behaves strangely, then make sure to restart Eclipse. Using a freshly installed Eclipse without restarting may have unpredicted side effects, such as leaving Eclipse in an unstable state. Therefore, it is a good practice to always restart it just after having installed a new plugin or a new version of the plugin.

Vaadin in IntelliJ IDEA

If you do not intend to use IntelliJ IDEA, you can safely skip this specific part and go to the *Vaadin and other IDEs* section.



IntelliJ IDEA is a commercial product from **JetBrains** (for more information, see <https://www.jetbrains.com/idea/>). Though there's a community edition, it works only with simple Java projects, and will not work with web applications, including Vaadin projects. Even if you do not intend to buy it, request an evaluation copy, it's valid for 30 days and you may be pleasantly surprised.

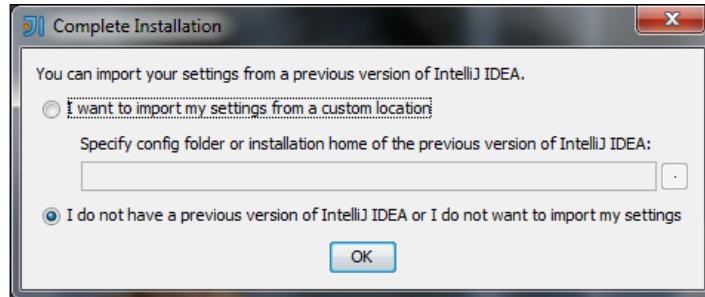
Setting up IntelliJ

At the time of this writing, the latest IntelliJ IDEA version is 12.1.3. There are two different distributions available, the simple Community edition which supports Java SE and the full Ultimate edition. We'll need the second one, since Vaadin needs JavaEE support. It can be downloaded from the following URL: <https://www.jetbrains.com/idea/download/index.html>.



Launch the installer, accept the License Agreement, and choose install location as well as shortcut menu, then start the application.

If IntelliJ IDEA wasn't installed previously on the machine, a pop up will be shown asking for a settings folder. Click on **OK**, since the radio button **I do not have a previous version of IntelliJ IDEA...** is selected by default.



Choose the right option, depending whether you bought a commercial license or want to try the product.

IntelliJ IDEA has an architecture which is based on plugins; the more the number of enabled plugins, the slower the IDE. The next step is to select which plugins should be enabled. Do not worry if there's a mistake, we will be able to enable/disable plugins later.

- The first window is about Source Configuration Management. Keep the one you want and click on **Next**.
- The second window is about JavaEE and frameworks. Choose at least **Application Servers** view.
- Notice there is a Vaadin choice-unfortunately, this embedded plugin is not compatible with version 7 of Vaadin, so there is no point in selecting it. Click on **Next**.
- The next window is about Application Servers. In this book, we will use either Eclipse's internal servlet container or Tomcat, since they are the simplest containers able to run Vaadin. In the last chapter, we'll also deploy Vaadin applications on CloudFoundry.
- Select **Tomcat** and **CloudFoundry**. Click on **Next**.

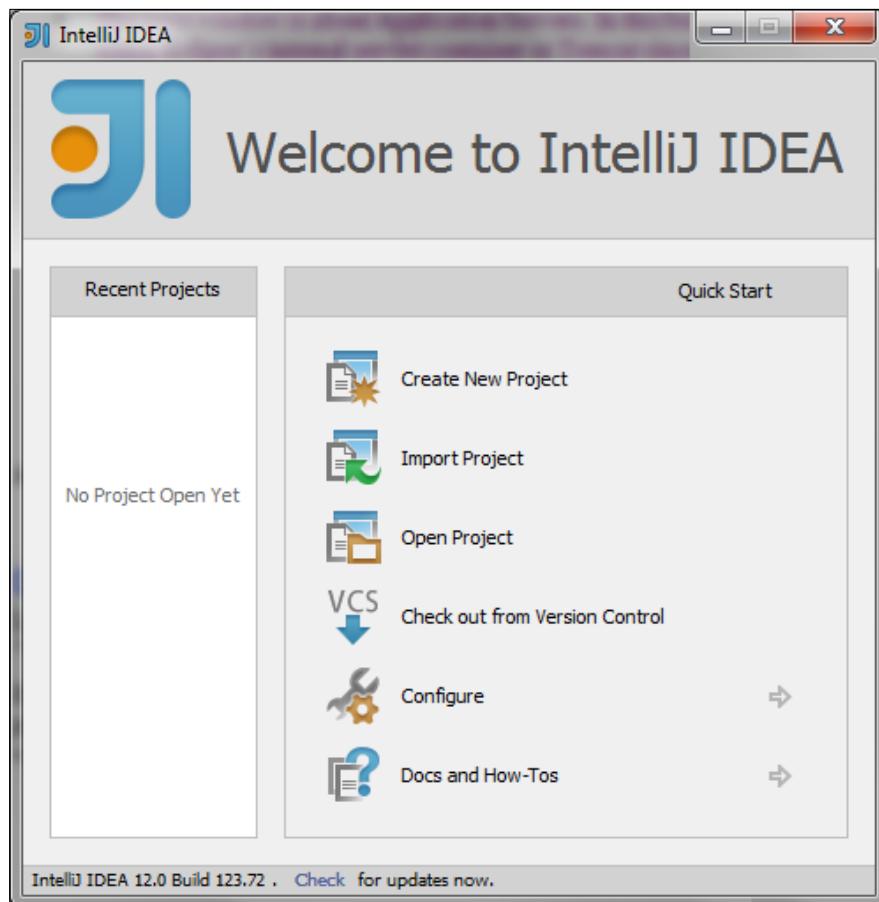


You may, however, use your favorite servlet container, like **Jetty** or more advanced application server like **RedHat JBoss Application Server**. In most cases, this won't matter, however.

Environment Setup

- The following window is about HTML/JavaScript. The good news is that those don't matter for developing Vaadin applications. Click on **Next**.
- The final window lists miscellaneous plugins. We need **Maven** and **Maven Integration Extension** for the Vaadin plugin we will download to work. Additionally, it's advised to at least keep **GitHub** selected, which is the chosen way to store the code source for the Twaattin example application. Click on **Next**.

At this point, IntelliJ will finally start.

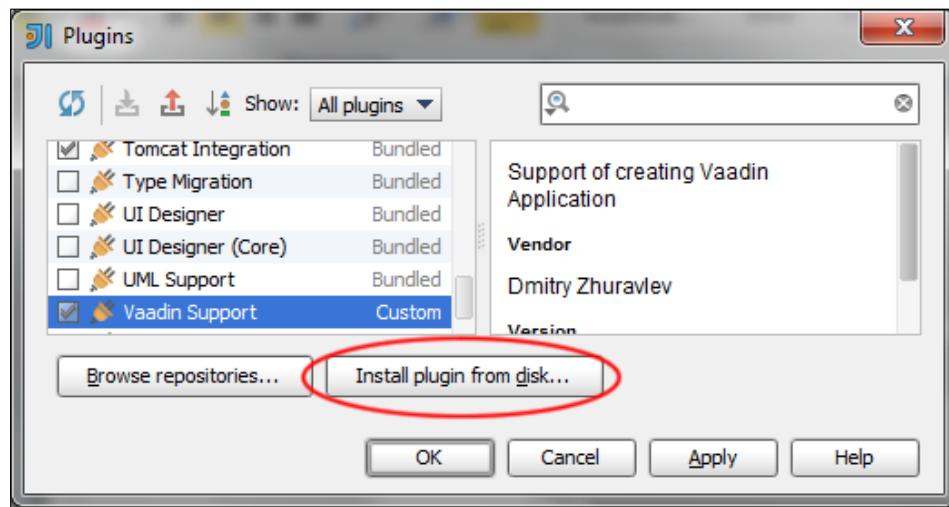


Adding the Vaadin 7 plugin

Though IntelliJ IDEA doesn't come with Vaadin 7 support, there's a plugin developed by *Dmitry Zhuravlev* and freely made available for anyone to use.

Go to <http://plugins.intellij.net/plugin/?idea&id=6727> and download the latest version (1.5.3 at the time of this writing).

Back to IntelliJ IDEA, select **Configure**, then **Plugins** (second item in the list). Click on the **Install plugin from disk...** button and select the previously downloaded plugin.

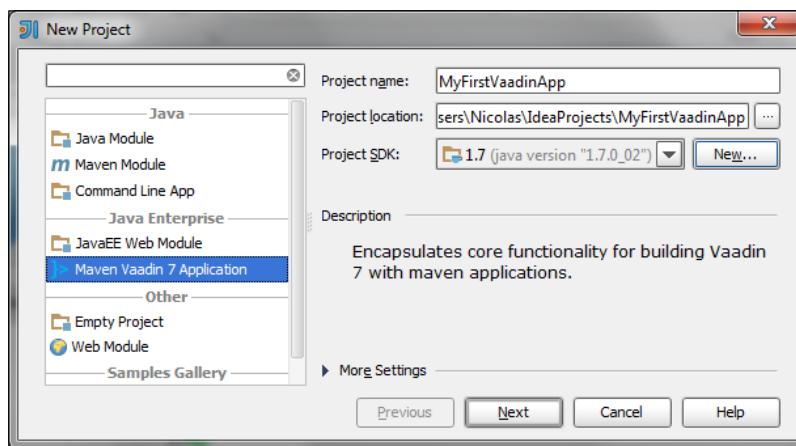


Finally, select the new **Vaadin Support** item and click on **OK**. Restart IntelliJ IDEA.

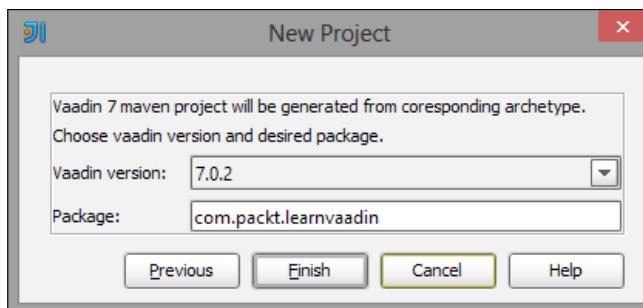
Now is the time to create a New Project!

Creating our first IntelliJ IDEA Vaadin project

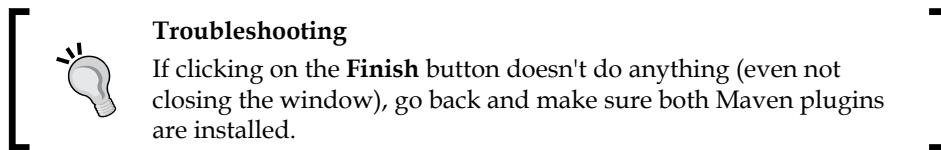
1. Select **Maven Vaadin 7 Application** (under the Java Enterprise title). Fill the fields according to the following:
 - Project name: MyFirstVaadinApp
 - Project location: as desired
 - Project SDK: choose the location of a Java SDK on the file system, it should at least be Java 6 (but 7 is ok)



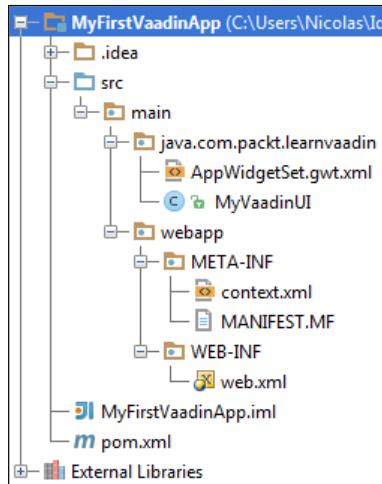
2. Click on **Next** and confirm Project creation.
3. Finally, choose the latest Vaadin 7 version made available by the plugin (7.0.2 at the time of this writing) and set com.packt.learnvaadin for the package name.



4. Click on **Finish**.



The result should look something like the following screenshot:



Notice the project uses Maven: if you've never used Maven before, that's not a problem. Just consider the project a standard Web Module.

Adjusting the result

In order to comprehend Vaadin in a step-by-step process, we will need to make some adjustments that will make deploying and running easier. Do not worry; we will cover those things in *Chapter 9, Creating and Extending Components and Widgets*. For the time being, please have faith, go to the webapp deployment descriptor and coldly remove the following snippet without any kind of remorse, it's not needed at this point:

```
<init-param>
    <description>Application widgetset</description>
    <param-name>widgetset</param-name>
    <param-value>
        com.packt.learnvaadin.AppWidgetSet
    </param-value>
</init-param>
```

Delete the `src/main/java/com.packt.learnvaadin.AppWidgetSet.gwt.xml` file accordingly.

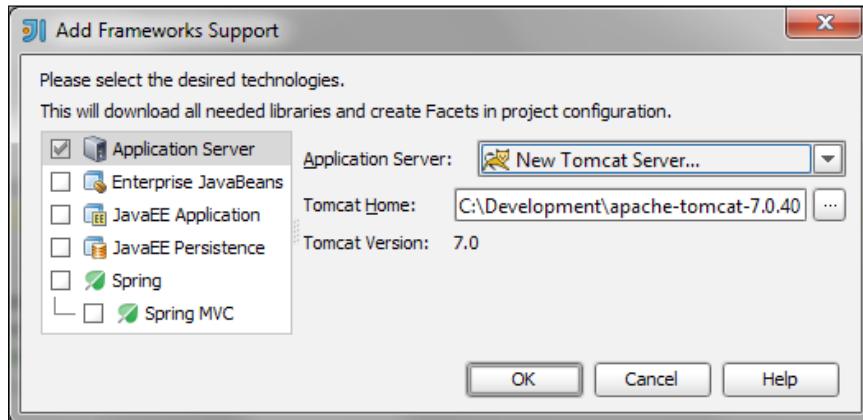
Adding framework support

This section assumes there's an available Tomcat installation on the local machine.

 [
 Installing Tomcat is as simple as browsing to the Tomcat 7 download page at <https://tomcat.apache.org/download-70.cgi>, choosing the latest version (7.0.40 at the time of this writing), downloading it, and unzipping it in the location of your choosing.]

In order to be able to run our newly created project, we need to tell IntelliJ IDEA how to run the application. Right-click on the project and choose **Add Framework support** (this is the second item).

In the opening window, select **Application Servers**, select **New Tomcat Server**, and point **Tomcat Home** to the directory where Tomcat is installed (or unzipped).



Deploying the application automatically

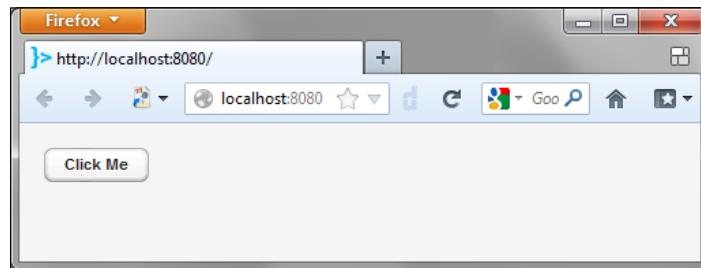
Click on the new Tomcat 7 combo-box in the toolbar and select the **Edit Configurations** item. In the opening window, choose the previously created local Tomcat Server, and go to the **Deployment** tab.

Click on the **+** button, choose artifact, and select `MyFirstVaadinApp.war`. Click on **OK**.

Testing the application

We can finally select **Run** in the toolbar.

It should automatically open the default browser and display a button for us to click.



Final touches

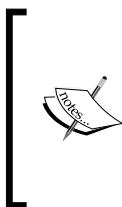
Compared to Eclipse, there are some differences. We have to close the gap in order for the rest of book's examples to be adequate for IntelliJ IDEA users.

Changing the Vaadin version

The downloaded plugin hardcodes available Vaadin versions, so that the latest version provided by the plugin isn't the latest version of Vaadin 7. To update to the latest version, just edit the `pom.xml` file at the root of the project and locate the following line:

```
<vaadin.version>7.0.2</vaadin.version>
```

Change the existing version with the latest (7.1.3 at the time of this writing).

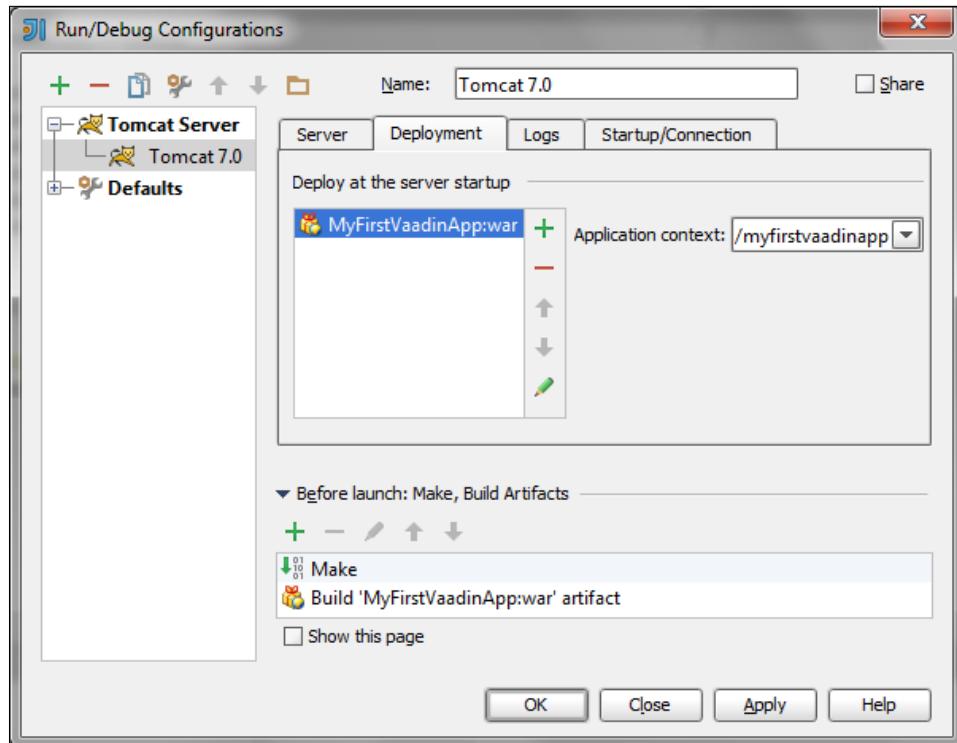


Changing the POM will trigger a confirmation dialog from IntelliJ IDEA: do you want to import the changes (that is, change the project configuration according to those changes) or auto-import them? It's advised to auto-import so that future changes will be automatically reflected in the configuration.

Launching the application again should yield the same results, albeit with the latest Vaadin framework version.

Context-root

IntelliJ IDEA provides the application at the root of the server, meaning it's available at `http://localhost:8080/`. In order for further instructions to be equally applicable to Eclipse projects as well as IntelliJ IDEA projects, we'll need to add a context-root.



Click on the **Edit Configuration** button in the toolbar and select the previously created Tomcat 7 server runtime. In the **Deployment** tab, select **MyFirstVaadinApp:war** and type `/myfirstvaadinapp` in the combo-box.



IntelliJ IDEA should automatically change the server startup page accordingly. If it is not the case, you can go to the **Server** tab and update the **Startup Page** field.

Servlet mapping

Finally, follow the instructions detailed in section *Servlet mapping* (in Eclipse).

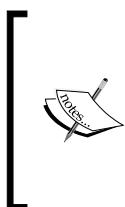
Now, even though the code itself is somewhat different from the Eclipse project, the configuration is the same.

Vaadin and other IDEs

In both IDEs, the Vaadin plugin helps us jumpstart a project in a matter of minutes. However, it may be interesting to understand what is really done in the created project by the plugin in case we need to do it manually in other IDEs which don't have such plugins.

Adding Vaadin libraries

First we should add Vaadin libraries and their dependencies to the web application's WEB-INF/lib folder. According to Java EE specifications, this means that our code can now access the Vaadin JAR as it is on the web application's classpath.



Previous versions of Vaadin only needed a single approximately 70 MB Vaadin JAR that contained all classes, all mandatory dependencies and compiled HTML. Starting with version 7, Vaadin has become more modular and started to rely on external dependencies: on one hand, it is now more complex to package what is needed, but on the other hand, we have the choice of what we package.

For the time being, there is no need to explicitly describe each library. All that are needed are available in the Vaadin 7 distribution at the root for Vaadin libraries and under the lib folder for their dependencies.

Creating the application

Then, we also need to create a class named com.packt.vaadin.MyUI as defined in our preceding sample application. Now, if we look at this class, we can see that it inherits from the com.vaadin.ui.UI class.

For now, suffice it to say that UI is the entry point class of our Vaadin application.

Adding the servlet mapping

The last thing the plugin does is update the WEB-INF/web.xml file, also known as the web deployment descriptor.

Looking at the file, we see the following Vaadin-specific lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.
  w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.
  com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>MyFirstVaadinApp</display-name>
  <context-param>
    <description>Vaadin production mode</description>
    <param-name>productionMode</param-name>
    <param-value>false</param-value>
  </context-param>
  <servlet>
    <servlet-name>My First Vaadin Application</servlet-name>
    <servlet-class>com.vaadin.server.VaadinServlet</servlet-class>
    <init-param>
      <description>Vaadin UI class to use</description>
      <param-name>UI</param-name>
      <param-value>com.packt.learnvaadin.MyUI</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>My First Vaadin Application</servlet-name>
    <url-pattern>/app/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>My First Vaadin Application</servlet-name>
    <url-pattern>/VAADIN/*</url-pattern>
  </servlet-mapping>
  <welcome-file-list>...</welcome-file-list>
</web-app>
```

Declaring the servlet class

The first thing to do is to declare the servlet. For servlet containers and full application servers, the servlet is provided by Vaadin and is `com.vaadin.server.VaadinServlet`. We will see in *Chapter 10, Enterprise Integration* that for more exotic platforms (such as **portlet**) containers and other classes are provided.

The servlet class is of utmost importance as it is the entry point of HTTP (as well as HTTPS) requests, as well as the exit point of their respective responses in Java web applications. The good news is that Vaadin takes care of handling requests (and sending responses) for us.

Declaring Vaadin's entry point

We saw in the preceding section that the real entry point in Vaadin is not the servlet but a delegated class that extends Vaadin's UI. We should make the servlet aware of this entry point, and this is done with the servlet initialization parameter aptly named UI.

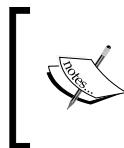
This parameter is not optional and Vaadin will vigorously complain if this parameter is omitted with the following explicit message:

com.vaadin.server.ServiceException: No UIProvider has been added and there is no "UI" init parameter.

If you ever encounter such a message, then your first reflex should be to check the deployment descriptor for the missing UI servlet initialization parameter.

Declaring the servlet mapping

Like in any Java web application, we should make our servlet available through a mapping that represents the URL part after the protocol, domain name, port (optional), and context-root that will activate the Vaadin servlet.



Refresher: In `http://packtpub.com:80/vaadin`, `packtpub.com` is the domain, `80` is the port, and `vaadin` is the context-root. The context-root we set up previously in the project is `/myfirstvaadinapp`.

In our previous setup, we configured the web application creation wizard to use the `/app/*` mapping.

Apart from specific cases, it is a bad idea to use the `/*` mapping for Vaadin servlet. That would mean that Vaadin servlet would have to handle every request in our application, including static resources, JAAS login form, and so on. We definitely don't want that.

Moreover, we will see later in *Chapter 7, Core Advanced Features* that a subcontext is also necessary in order to properly close the Vaadin applications.

The second servlet mapping is added because the internal Vaadin's themes and resources are referenced under `<context-root>/VAADIN/*` and thus, should be handled by the Vaadin servlet. As a developer, you can safely ignore this part (but don't remove it!).

Alternatively, we can also provide access to themes and resources ourselves, but this is the simplest option.

 If you run the Vaadin servlet under the more general `/*` mapping, then there is no need for this additional mapping.

Summary

In this chapter, we have seen how to:

- Correctly set up our IDE, depending on whether it is already present on our system or not
- Enhance our IDE with specific Vaadin features to make our project set up faster
- Create a new project using the plugin's help
- Add the Vaadin framework to a project when no plugin is available

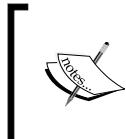
This chapter forms the basis of all your future work with Vaadin, so be sure to grasp all the concepts explained in it. After having created a basic "Hello World" project, the next chapter will detail Vaadin internals, as well as how to deploy our project outside the IDE.

3

Hello Vaadin!

In this chapter, we will cover the following:

- Key concepts behind the Vaadin framework
- An overview of its internal architecture
- Deploying a Vaadin application to a servlet container, be it in an IDE or outside it
- Updating the previously developed application with a very simple interaction in order to display "Hello Vaadin!"



In the rest of this book, we will use Eclipse IDE for detailed explanations and screenshots. However, there are enough similarities between Eclipse and IntelliJ IDEA such that those same explanations can safely be used in IntelliJ IDEA.

Now enough with the talk, let us begin.

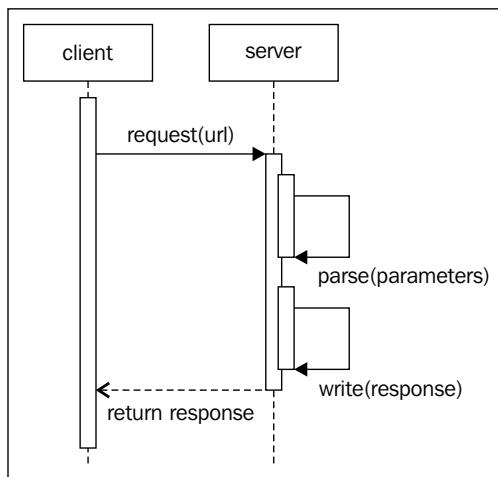
Understanding Vaadin

In order to understand Vaadin, we should first understand its goal regarding the development of web applications.

Vaadin's philosophy

Classical HTML over HTTP application frameworks are coupled to the inherent request/response nature of the HTTP protocol. This simple process translates as follows:

1. The client makes a request to access a URL.
2. The code located on the server parses request parameters (optional).
3. The server writes the response stream accordingly.
4. The response is sent to the client.



All major frameworks (and most minor ones, by the way) do not question this model; Struts, Spring MVC, Ruby on Rails, and others, completely adhere to this approach and are built upon this request/response way of looking at things. It is no mystery that HTML/HTTP application developers tend to comprehend applications through a page-flow filter.

On the contrary, traditional client-server application developers think in components and data binding because it is the most natural way for them to design applications (for example, a select-box of countries or a name text field).

The Play Framework (<http://www.playframework.org/>) takes a radical stance on the page-flow subject, stating that the Servlet API is a useless abstraction on the request/response model and sticks even more to it.

On the contrary, a few web frameworks, such as JSF, tried to cross the bridge between components and page-flow, with limited success. The developer handles components, but they are displayed on a page, not a window, and he/she still has to manage the flow from one page to another.

Vaadin's philosophy is two-fold:

- It lets developers design application through components and data bindings
- It isolates developers as much as possible from the request/response model in order to think in screens and not in pages and page flow

This philosophy lets developers design their applications the way it was before the web revolution. In fact, fat client developers can learn Vaadin in a few hours and start creating applications in no time.

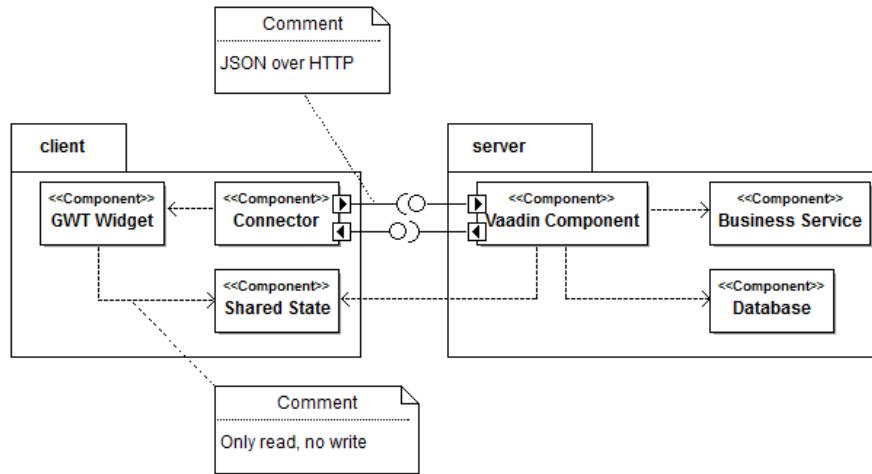
The downside is that developers who learned their craft with the thin client and have no prior experience of fat client development will have a hard time understanding Vaadin, as they are inclined to think in page-flow. However, they will be more productive in the long run.

Vaadin's architecture

In order to achieve its goal, Vaadin uses an inventive architecture. The first fact of interest is that it is shared to both a server and a client side.

- The client side manages thin rendering and user interactions in the browser
- The server side handles events coming from the client and sends changes made to the user interface to the client

- Communication between both tiers is done over the HTTP protocol using JSON over AJAX:



We will have a look at each of these tiers.

Client-server communication

Messages in Vaadin use two layers: HTTP and JSON. Both are completely unrelated to the Vaadin framework and are supported by independent third parties.

HTTP protocol

Using the HTTP protocol with Vaadin has the following two main advantages:

- There is no need to install anything on the client, as browsers handle HTTP (and HTTPS for that matter) natively
- Firewalls that let the HTTP traffic pass (a likely occurrence) will let Vaadin applications function normally

JSON message format

Vaadin messages between the client and the server use **JavaScript Objects Notation (JSON)**. JSON is an alternative to XML that has the following differences:

- First of all, XML has features to make it more standardized and type safe. Different technologies are available to enforce such grammar, but the most used is XML Schemas. This way, XML documents can be validated according to grammar.

- Without grammar, JSON is also much easier to process in many languages. In particular, JavaScript has built-in support for JSON processing. On the contrary, XML must be parsed to the DOM and only then can we inspect values and act upon them.
- The JSON syntax is lighter than the XML syntax. XML has both a start and an end tag, whereas JSON has a tag coupled with starting brace and ending brace. For example, the following two code snippets convey the same information, but the first requires 78 characters and the second only 63. For a more in-depth comparison of JSON and XML, refer to <http://json.org/xml.html>:

```
<person>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
</person>

{"person" : {
    "firstName": "John",
    "lastName": "Doe"
}}
```

The difference varies from message to message, but on an average, it is about 40 percent. It is a real asset only for big messages, and if you add server GZIP compression, size difference starts to disappear. The reduced size is no disadvantage, though.

- Finally, XML designers go to great lengths to differentiate between child tags and attributes, the former being more readable to humans and the latter to machines. The JSON message design is much simpler as JSON has no attributes.

The client part

The client tier is a very important tier in web applications as it is the one with which the end user directly interacts.

In this endeavor, Vaadin uses the excellent **Google Web Toolkit (GWT)** framework. GWT has been mentioned in *Chapter 1, Vaadin and its Context*. However, we will need a deeper understanding on how it is used in Vaadin.

In the GWT development, there are the following mandatory steps:

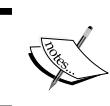
1. The code is developed in Java.
2. Then, the GWT compiler transforms the Java code in JavaScript, one set for each configured browser type.
3. Finally, the generated JavaScript is bundled with the default HTML and CSS files, which can be modified as a web application.

Although novel and unique, this approach provides the following interesting key features that catch the interest of end users, developers, and system administrators alike:

- Disconnected capability, in conjunction with HTML 5 client-side data stores
- Displaying applications on small-form factors, such as those of handheld devices
- Development only with the Java language
- Excellent scalability, as most of the code is executed on the client side, thus freeing the server side from additional computation

On the other hand, there is no such thing as a free lunch! There are definitely disadvantages in using GWT, such as the following:

- The whole coding/compilation/deployment process adds a degree of complexity to the standard Java web application development and according to most developers, is very (very) slow.
- Google GWT plugins are available for Eclipse and NetBeans and GWT development is supported natively by IntelliJ IDEA. Those are really necessary, because without them, developing is much slower and debugging almost impossible.



For more information about GWT dev mode, please refer to
<https://developers.google.com/web-toolkit/doc/latest/DevGuideCompilingAndDebugging>.

- There is a consensus in the community that GWT has a higher learning curve than most classic web application frameworks; although the same can be said for others, such as JSF.
- If the custom JavaScript is necessary, you have to bind it in Java with the help of a stack named **JavaScript Native Interface (JSNI)**, which is both counter-intuitive and complex.

- With pure GWT, developers have to write the server-side code themselves (if there is some).
- Finally, if everything is done on the client side, it poses a great security risk. Even with obfuscated code, the business logic is still completely open for inspection from hackers. Of course, this is dependent on the architecture, as logic can still be hosted server side.

Vaadin uses GWT features extensively and tries to downplay its disadvantages as much as possible. This is all possible because of the Vaadin's server part.

The server part

Vaadin's server-side approach plays a crucial role in the framework.

While in GWT, every code update needs a slow Java-to-HTML compilation (and even slower because of multiple target browsers), Vaadin code update only requires a simple, single Java-to-byte code compilation.



It is possible to add native GWT widgets and/or custom JavaScript to Vaadin 7 applications. An example of such use will be shown in *Chapter 9, Creating and Extending Components and Widgets*.



Remember throughout your Vaadin journey that you should favor coding server side first and foremost!

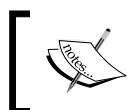
There are two important tradeoffs that Vaadin makes in order achieve this:

- As opposed to GWT, the user interface related code runs on the server, meaning Vaadin applications are not as scalable as pure GWT ones. This should not be a problem in most applications, but if you need to, you should probably leave Vaadin for some less intensive part of the application; stick to GWT or change to an entirely new technology.



While Vaadin applications are not very scalable when compared to applications architecture around a pure JavaScript frontend and a SOA backend, a study (for Vaadin Version 6) found that a single Amazon EC2 instance could handle more than 10,000 concurrent users per minute, which is probably higher than your average application. The whole result for you to reproduce can be found at <http://vaadin.com/blog/-/blogs/vaadin-scalability-study-quicktickets>.

- Second, each user interaction creates an event from the browser to the server (though they can be buffered to prevent unwanted network usage). This can lead to changes in the user interface's model in memory and, in turn, propagate modifications to the JavaScript UI on the client. The consequence is that server-side components simply cannot run while disconnected from the server! If your requirements include the offline mode, you will have to develop GWT widgets that may run offline yourself.



Note that Vaadin TouchKit, an add-on aimed at mobile application development, lets us have an offline mode. This is only true when using this add-on, though.



Client-server synchronization

The biggest challenge when representing the same model on two heterogeneous tiers is synchronization between each tier. An update on one tier should be reflected on the other, or at least fail gracefully if this synchronization is not possible (an unlikely occurrence considering the modern day infrastructure).

Vaadin's answer to this problem is a synchronization key generated by the server and passed on to the client on each request. The next request should send it back to the server or else the latter will restart the current session's application.

Communication problem

Take note of any unsaved data, and [click here](#) to continue. Invalid status code 0 (server down?)

Deploying a Vaadin application

Now we will see how we can put what we have learned to good use.

Vaadin applications are primarily web applications and they follow all specifications of Web Archive artifacts, as specified by JavaEE Version 1.4. As such, there is nothing special for deploying Vaadin web applications. Readers who are familiar with the WAR deployment process will feel right at home!



WAR deployment itself is dependent on the specific application server (or servlet/JSP container).



Inside the IDE

In the last chapter, we smoke-tested our brand new Vaadin application with Eclipse's mock servlet container. In most cases, we will need features not available on the latter, for example, data sources management.

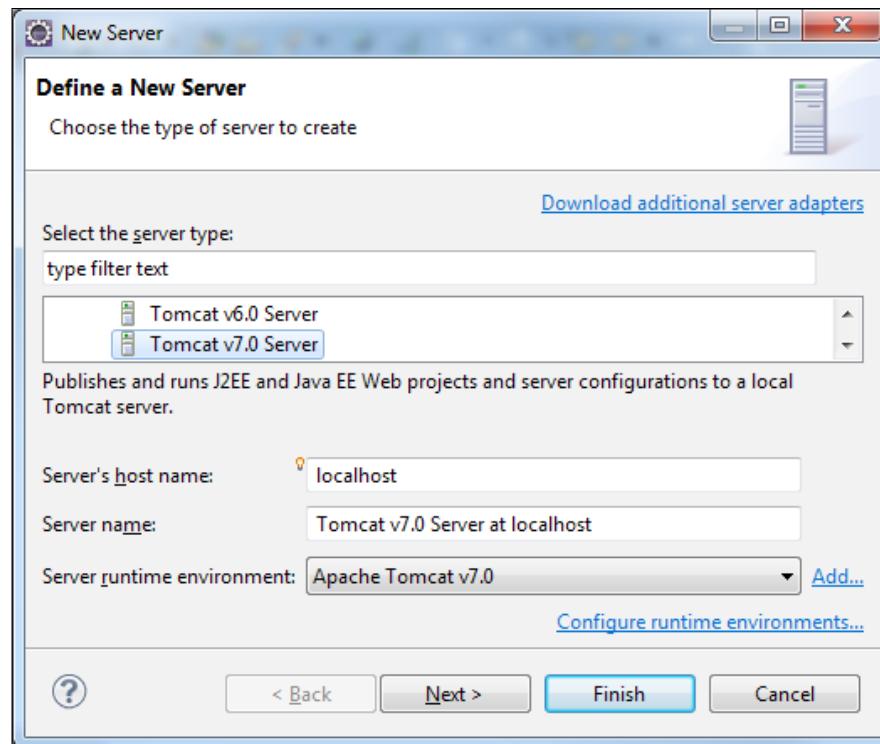
Creating an IDE-managed server

Although it is possible to export our project as a WAR file and deploy it on the available servlet container, the best choice is to use a server managed by the IDE. It will let us transparently debug our Vaadin application code.

The steps are very similar to what we did with the mock servlet container in *Chapter 2, Environment Setup*.

Selecting the tab

First of all, if the **Server** tab is not visible, navigate to **Window | Open perspective | Other...** and later choose **JavaEE**.



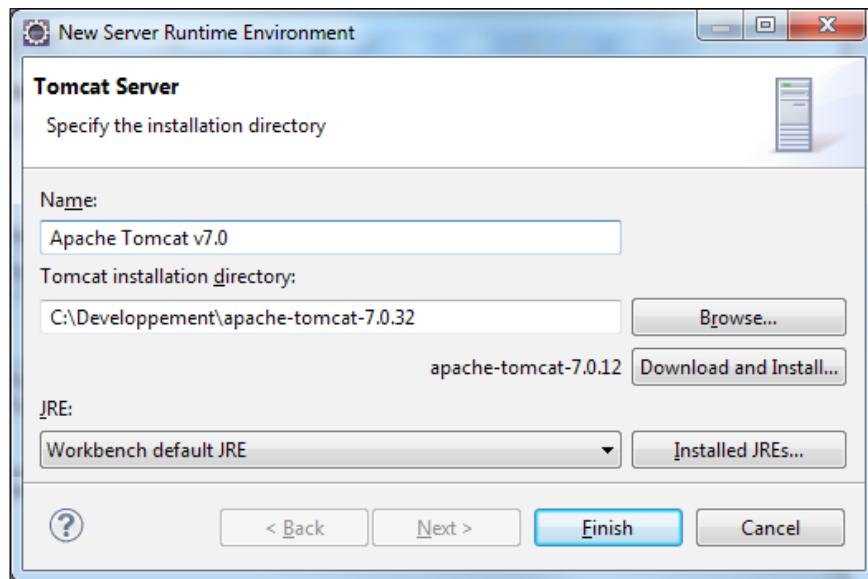
Creating a server

In order to be as simple as possible, we will use Tomcat. Tomcat is a servlet container, as opposed to a full-fledged application server, and only implements the servlet specifications, not the full Java EE stack. However, what it does, it does it so well that Tomcat was once the servlet API reference implementation.

Right-click on the **Server** tab and navigate to **New | Server**. Open Apache and select **Tomcat 7.0 Server**. Keep both **Server's hostname** and **Server name** values and click on **Next**.

Now the following two options are possible:

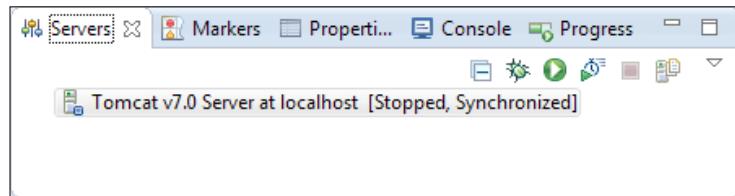
1. If you do not have Tomcat 7 installed, click on **Download and install**. Accept the license agreement and then select the directory where you want to install it to, as shown in the following screenshot:



2. If it is already installed, just point to its root location in the **Tomcat installation directory** field.
3. Click on the **Finish** button.

Verifying the installation

At the end of the wizard, there should be a new Tomcat 7.0 server visible under the **Servers** tab, as shown in the following screenshot. Of course, in case you chose another version or another server altogether, that will be the version or server displayed:



Adding the application

As Vaadin applications are web applications, there is no special deployment process.

Right-click on the newly created server and click on the **Add and Remove** menu entry. A pop-up window opens. On the left-hand side, there is a list of available web application projects that are valid candidates to be deployed on your newly created server. On the right-hand side, there is a list of currently deployed web applications.

Select the **MyFirstVaadinApp** project we created in *Chapter 2, Environment Setup* and click on the **Add** button. Then click on **Finish**.

The application should now be visible under the server.

Launching the server

Select the server and right-click on it. Select the **Debug** menu entry. Alternatively, you can do the following:

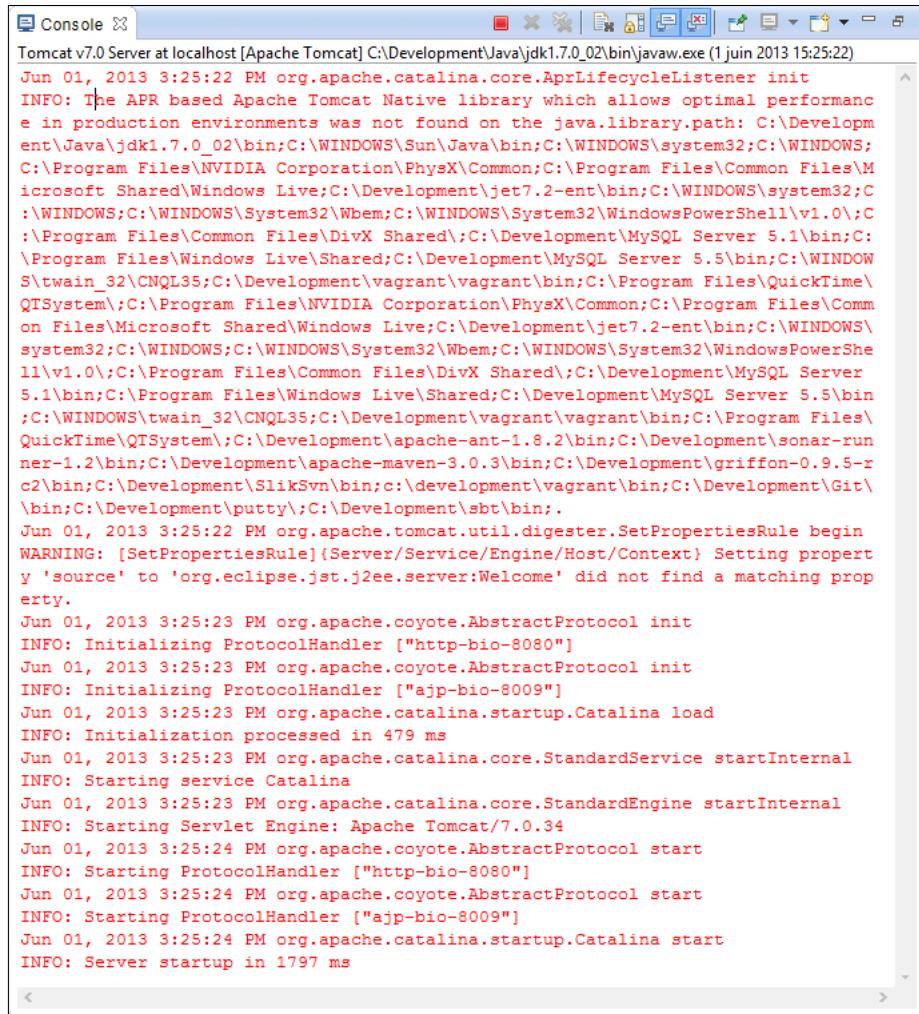
- Click on the **Debug** button (the one with the little bug) on the **Server** tab header
- Press *Ctrl + Alt + D*



Each IDE has its own menus, buttons, and shortcuts. Know them and you will enjoy a huge boost in productivity.

Hello Vaadin!

The **Console** tab should display a log similar to the following:



```
Console
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\Development\Java\jdk1.7.0_02\bin\javaw.exe (1 juin 2013 15:25:22)
Jun 01, 2013 3:25:22 PM org.apache.catalina.core.AprLifecycleListener init
INFO: The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: C:\Development\Java\jdk1.7.0_02\bin;C:\WINDOWS\Sun\Java\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\NVIDIA Corporation\PhysX\Common;C:\Program Files\Common Files\Microsoft Shared\Windows Live;C:\Development\jet7.2-ent\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0;C:\Program Files\Common Files\DivX Shared;C:\Development\MySQL Server 5.1\bin;C:\Program Files\Windows Live\Shared;C:\Development\MySQL Server 5.5\bin;C:\WINDOWS\TWAIN_32\CNQL35;C:\Development\vagrant\vagrant\bin;C:\Program Files\QuickTime\QTSystem\;C:\Program Files\NVIDIA Corporation\PhysX\Common;C:\Program Files\Common Files\Microsoft Shared\Windows Live;C:\Development\jet7.2-ent\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0;C:\Program Files\Common Files\DivX Shared;C:\Development\MySQL Server 5.1\bin;C:\WINDOWS\TWAIN_32\CNQL35;C:\Development\vagrant\vagrant\bin;C:\Program Files\QuickTime\QTSystem\;C:\Development\apache-ant-1.8.2\bin;C:\Development\sonar-runner-1.2\bin;C:\Development\apache-maven-3.0.3\bin;C:\Development\griffon-0.9.5-rc2\bin;C:\Development\SlkSvn\bin;c:\development\vagrant\bin;C:\Development\Git\bin;C:\Development\putty\;C:\Development\sbt\bin;
Jun 01, 2013 3:25:22 PM org.apache.tomcat.util.digester.SetPropertiesRule begin
WARNING: [SetPropertiesRule]{Server/Service/Engine/Host/Context} Setting property 'source' to 'org.eclipse.jst.j2ee.server:Welcome' did not find a matching property.
Jun 01, 2013 3:25:23 PM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-bio-8080"]
Jun 01, 2013 3:25:23 PM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["ajp-bio-8009"]
Jun 01, 2013 3:25:23 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 479 ms
Jun 01, 2013 3:25:23 PM org.apache.catalina.core.StandardService startInternal
INFO: Starting service Catalina
Jun 01, 2013 3:25:23 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.34
Jun 01, 2013 3:25:24 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Jun 01, 2013 3:25:24 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
Jun 01, 2013 3:25:24 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 1797 ms
```

This means Tomcat started normally.

Outside the IDE

In order to deploy the application outside the IDE, we should first have a deployment unit.

Creating the WAR

For a servlet container such as Tomcat, the deployment unit is a **Web Archive**, better known as a **WAR**.

Right-click on the project and select the **WAR file** from under the **Export** menu. In the opening pop up, just update the location of the exported file; choose the `webapps` directory where we installed Tomcat and name it `myfirstvaadinapp.war`.

Launching the server

Open a command prompt. Change the directory to the `bin` subdirectory of the location where we installed Tomcat, and run the `startup` script.

Troubleshooting

If you have installed Tomcat for the first time, chances are that the following message will be displayed:



`Neither the JAVA_HOME nor the JRE_HOME environment variable is defined`

`At least one of these environment variables is needed to run this program`

In this case, set the value of the `JAVA_HOME` variable to the directory where Java is installed on your system (and not its `bin` subdirectory!).

The log produced should be very similar to the one displayed by running Tomcat inside the IDE (as shown in preceding section), apart from the fact that **Apache Portable Runtime** is perhaps available on the classpath. As it is not mandatory in any way – it is meant to improve performance for production systems – that does not change a thing from the Vaadin point of view.

Using Vaadin applications

Vaadin being a web framework, its output is displayed inside a browser.

Browsing Vaadin

Whatever way you choose to run our previously created Vaadin project, in order to use it, we just have to open one's favorite browser and navigate to `http://localhost:8080/myfirstvaadinapp/app`. Two things should happen:

1. First, a simple page should be displayed with the message **Hello Vaadin user** (or a **Click Me** button if using IntelliJ IDEA).
2. Second, the log should output that Vaadin has started:

```
=====
Vaadin is running in DEBUG MODE.
Add productionMode=true to web.xml to disable debug features.
To show debug window, add ?debug to your application URL.
=====
```

Troubleshooting

In case nothing shows up on the browser screen, and after some initial delay, an error pop up opens with the following message:

 Failed to load the widgetset: /myfirstvaadinapp/VAADIN/widgetsets/com.vaadin.terminal.gwt.DefaultWidgetSet/com.vaadin.terminal.gwt.DefaultWidgetSet.nocache.js?1295099659815

Be sure to add the `/VAADIN/*` mapping to the `web.xml` as shown in *Chapter 2, Environment Setup* in the *Declaring the servlet mapping* section and redeploy the application.

Out-of-the-box helpers

Before going further, there are two things of interest to know that are precious when developing Vaadin web applications.

The debug mode

Component layout in real-world business use cases can be a complex thing, to say the least. In particular, requirements about fixed and relative positioning are a real nightmare when one goes beyond Hello world applications, as it induces nested layouts and component combinations at some point.

Given the generated code approach, when the code does not produce exactly what is expected on the client side, it may be very difficult to analyze the cause of the problem. Luckily, Vaadin designers have been confronted with them early on and are well aware of this problem.



```
507ms Handling type mappings from server
511ms Handling resource dependencies
512ms * Handling locales
513ms * Handling meta information
514ms * Creating connectors (if needed)
519ms * Updating connector states
524ms * Updating connector hierarchy
525ms * Sending hierarchy change events
530ms * Running @DelegateToWidget
531ms * Sending state change events
535ms * Passing UIDL to Vaadin 6 style connectors
536ms * Performing server to client RPC calls
537ms * Unregistered 0 connectors
539ms handleUIDLMessage: 26 ms
540ms Starting layout phase
567ms Measured 1 non connector elements
569ms No more changes in pass 1
570ms Total layout phase time: 28ms
570ms * Dumping state changes to the console
571ms UIDL: undefined
582ms Processing time was 80ms for 1744 characters of JSON
583ms Referenced paintables: 4
```

As such, Vaadin provides an interesting built-in debugging feature; if you are ever faced with such a display problem, just append `?debug` to your application URL. This will instantly display a neat window that gives detailed inside data about the application:

- The first tab displays the console.
- The second shows a view of the widgets hierarchy (or more specifically of the connectors, which will be explained in later chapters). It is a very useful debugging aid when components laid out on the server side are not shown in the browser.
- The third tab reveals the widget's inner state.
- The rest of the tabs manage settings, let us minimize the console and finally exit it.



Just be aware that this window is not native (it is just an artifact created with the client-side JavaScript). It can be moved with its upperbar, which is invaluable if you need to have a look at what is underneath it. Likewise, it can be resized by moving the cursor over a corner and dragging the mouse, just like a native window.

Although it considerably decreases the debugging time during the development phase, such a feature has no added value when in production. It can even be seen as a minor security risk, as the debug window displays information about the internal state of Vaadin's widget tree.

Vaadin provides the means to disable this feature. In order to do so, just add the following snippet to your `WEB-INF/web.xml`:

```
<context-param>
<description>Vaadin production mode</description>
<param-name>productionMode</param-name>
<param-value>true</param-value>
</context-param>
```

Now if you try the debug trick, nothing will happen.



Production mode is NOT default

As such, it is a good idea to always set the `productionMode` context parameter from the start of the project, even if you set it to `false`. Your build process would then set it to `true` for release versions. This is much better than forgetting it altogether and having to redeploy the webapp when it becomes apparent.

Restart the application, not the server

We have seen in the previous *Vaadin's architecture* section that Vaadin's user interface model is sent to the client through JSON messages over HTTP. The whole load is sent at the first request/response sequence when the UI instance is initialized; additional sequences send only DOM updates.

Yet, important changes to the component tree happen often during the development process. By default, refreshing the browser does display such changes.



Initializing the UI at each browser refresh is a new feature of Vaadin 7. In the previous version, state was kept between refreshes.

In order to keep the state between refreshes, we can annotate our UI with `@com.vaadin.annotations.PreserveOnRefresh`. In this case, we can still refresh explicitly through a URL parameter; change your server-side code, wait for the changes to take effect on the server, just append `?restartApplication` and watch the magic happen.



Increase performance

You should remove the `restartApplication` URL parameter as soon as it is not needed anymore. Otherwise you will rerun the whole initialization/send UI process each time you refresh the browser.

Behind the surface

Wow, in just a few steps, we created a brand new application! Granted, it does not do much, those simple actions are fundamental to the comprehension of more advanced concepts seen in further chapters. So let's catch our breath and see what really happened under the hood.

Stream redirection to a Vaadin servlet

The URL `http://localhost:8080/myfirstvaadinapp/app` can be decomposed in the following three parts, each part being handled by a more specific part:

1. `http://localhost:8080` is the concatenation of the protocol, the domain, and the port. This URL is handled by the Tomcat server we installed and started previously, whether normal or inside the IDE.

2. /myfirstvaadinapp is the context root and references the project we created before. Thus, Tomcat redirects the request to be handled by the webapp.
3. In turn, /app is the servlet mapping the Vaadin plugin added to the web deployment descriptor at the time the project was created. The servlet mapping uses the Vaadin servlet, which is known under the logical name My First Vaadin Application. The latter references the `com.vaadin.server.VaadinServlet` class.

Vaadin request handling

As you can see, there is nothing magical in the whole process; the URL we browsed was translated as a request that is being handled by the `vaadinServlet.service()` method, just like any Java EE compliant servlet would do.



Vaadin's servlet directly overrides `service()` instead of the whole group of `doXXX()` methods (such as `doGet()` and `doPost()`). This means that Vaadin is agnostic regarding the HTTP method you use. Purists and REST programmers will probably be horrified at this mere thought, but please remember that we are not manipulating HTTP verbs in request/response sequences and are instead using an application.

Compared to Vaadin 6, the Vaadin servlet does not do much but delegate operations to other classes. In order for most of these classes to be accessible outside the servlet, Vaadin 7 uses the `ThreadLocal` pattern. One of the servlet's most important activity is to put collaborating classes in the thread local at the beginning of the `service()` method and to clean them at the end.



Initial load time

Be wary of this last step when creating your own applications; an initial screen that is too big in size will generate an important latency followed by a strange update of your client screen. This is generally not wanted; either try to decrease your initial page complexity or use a change manager that will mitigate the user's feelings about it.

What does a UI do?

In Vaadin, a UI represents the topmost component in a components hierarchy. The central class representing an application is the `com.vaadin.ui.UI` class invoked during `VaadinServletservice()` method call.

You can think of a UI as the entire browser window (or tab in modern browsers) for full-fledged Vaadin applications, or an HTML page embedding a Vaadin webapp.

UI features

UI features include the following:

- Providing an entry point into the application. In this regard, the UI `init()` method is called when the servlet mapping whose `init` parameter matches the UI class name is accessed.

In our first example, the **Hello Vaadin user** label is displayed because the URL accessed is the root. It maps to a `VaadinServlet` instance which is parameterized with `MyUI`; its `init()` method displays the label.

- Keeping state or not between browser refreshes, depending of the presence of the `@PreserveOnRefresh` annotation.
- Setting the HTML title through the use of the `@com.vaadin.annotations.Title` annotation.
- Setting the overall theme for the UI, thanks to `@com.vaadin.annotations.Theme`.



Some themes are provided out of the box by Vaadin (`liferay`, `chameleon`, `reindeer`, and `runo`). You can also tweak them and reference them under a new theme name or create entirely new themes from scratch. Themes, being very simple to use but much more complex to create, are outside the scope of this book. Readers interested in going further down road can find documentation at <http://vaadin.com/book/-/page/themes.creating.html>.

UI configuration

In our first project, having a look at the web deployment descriptor, notice there is a UI servlet parameter configured for the Vaadin servlet:

```
<servlet>
    <servlet-name>My First Vaadin Application</servlet-name>
    <servlet-class>
        com.vaadin.server.VaadinServlet
    </servlet-class>
    <init-param>
        <param-name>UI</param-name>
        <param-value>
            com.packt.learnvaadin.MyUI
        </param-value>
    </init-param>
</servlet>
```

As such, there can be only a single Vaadin UI configured for each Vaadin servlet.



Sometimes we do not know the UI to use at the beginning because the instance must be defined dynamically at runtime. In this case, instead of defining a UI, it is possible to define a UIProvider. More info regarding providers can be found in the Vaadin wiki at <https://vaadin.com/wiki/-/wiki/Main/Creating%20an%20application%20with%20different%20features%20for%20different%20clients>.

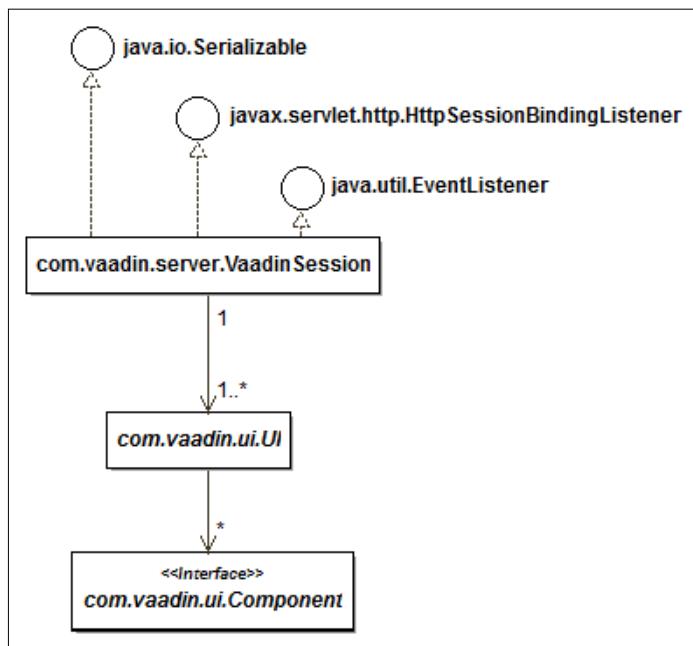
UI and session

The most important fact about the UI class is that one instance of it is created the first time a user session requests the Vaadin servlet; this instance is stored in the HttpSession of the current user. This means that having multiple UI instances opened at the same time is not a good thing regarding memory; the UI.close() method should be called once a UI is not needed anymore in order for Vaadin to remove the UI from the session.



In reality, UI instances are not stored directly in HttpSession, but within a com.vaadin.server.VaadinSession instance that is stored in the session. There is a 1-n relationship from VaadinSession to HttpSession meaning there is a possibility that more than one VaadinSession could relate to the same session. You should keep in mind that each session stores one and only one Vaadin session object for each configured Vaadin servlet.

Vaadin's object model encompasses `VaadinSession`, `UI`, and `AbstractComponent`, as shown in the following diagram:



Storing state on the server

Storing `UI` instances and their hierarchy in the session has a major consequence. Great care must be taken in evaluating the number of users and the average load of each user session because the session is more loaded than in traditional Java EE web applications, thus greater is the risk of `java.lang.OutOfMemoryError`.

Besides, when used on clustered application servers, state has to be serialized for replication, so it is a huge requirement on stored objects.



Scratching the surface

Having said all that, it is time to have a look at the both the source code that was created by the Vaadin plugin and the code that it generated and pushed on the client.

The source code

The source code was taken care of by Vaadin plugin:

```
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

public class MyUI extends UI {

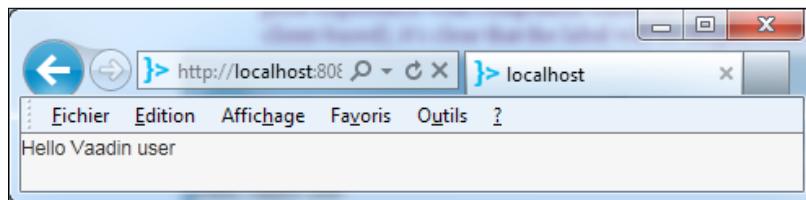
    @Override
    public void init(VaadinRequest request) {

        Label label = new Label("Hello Vaadin user");

        setContent(label);
    }
}
```

Though having no prior experience in Vaadin and only armed with some basic concepts, we can guess what the class does. That is the strength of Vaadin-compared to competitor frameworks, it is self-evident!

- In the first instruction of method `init()`, we create a new label. Labels are used to display static messages. They are often found in web forms as description for fields. Our label has a specific text. Notice it is displayed in the final screen.
- In the second instruction, we add the label to the UI. Even when you have no prior experience with component-based development (whether thin or fat client-based), it is clear that the label will be displayed in the window:



The generated code

In your favorite browser, right-clicking and selecting the menu that shows the source will only display JavaScript—gibberish to the inexperienced eye.

In fact, as the UI is generated with GWT, we do not see anything interesting in the HTML source – only the referenced JavaScript and a single `<noscript>` tag that handles the case where our browser is not JavaScript-enabled (an unlikely occurrence in our time, to say the least).

 There is a consensus on the Web that JavaScript-powered web applications should degrade gracefully, meaning that if the user deactivates JavaScript, applications should still run, albeit with less user-friendliness. Although it is a very good practice most of the time, JavaScript applications will not run at all in this case. GWT, and thus, Vaadin, are no exceptions in this matter.

Of much more interest is the current HTML/JS/CSS. In order to display it, we will need Google Chrome, Firefox with the Firebug plugin, or an equivalent feature in another browser.

 Firebug is a must-have for all web developers using Firefox. It's free and available from <http://getfirebug.com/>.

More precisely, locate the following snippet:

```
<div id="myfirstvaadinappapp-603887191" class=" v-app reindeer">
    <div class="v-ui v-scrollable" tabindex="1"
        style="height: 100%; width: 100%; ">
        <div class="v-loading-indicator"
            style="position: absolute; display: none;"></div>
        <div class="v-label v-widget v-has-width" style="width: 100%; ">
            Hello Vaadin user
        </div>
    </div>
</div>
```

Things of interest

First of all, notice that although only a simple message is displayed on the user screen, Vaadin has created an entire DOM tree filled with `<div>` elements that has both `style` and `class` attributes. We will see later in *Chapter 4, Components and Layouts*, that Vaadin (through GWT) creates, at least, such a `<div>` element for every layout and component. For now, just be aware of the following:

- The class `v-ui` denotes a UI
- The class `v-label` indicates a label

Summary

We saw a few things of importance in this chapter.

First, we had an overview of the Vaadin's philosophy. Vaadin creates an abstraction over the classic request/response sequence in order for developers to think in "applications" and no more in "pages".

In order to do that, the Vaadin's architecture has the following three main components:

- The client side that is JavaScript upon the Google Web Toolkit
- The server side that calls appropriate fragments of client code
- Communications between the client and the server is implemented with JSON messages over the HTTP protocol

Then we deployed the Vaadin application we developed in *Chapter 2, Environment Setup*. There is nothing special with Vaadin applications, they are simple web archives and are deployed as such.

The debug window, which is very convenient when debugging display and layout-related bugs, comes bundled with Vaadin.

Finally, we somewhat scratched the surface of how it all works, most notably the following:

- The handling of an HTTP request by Vaadin
- The notion of UI in the framework
- The code, both source-generated by the plugin and the HTML structure generated by the former

This chapter concludes the introduction to Vaadin. It is a big part on the path to learning Vaadin. If you feel the need to take a pause, then this is the right time to do so. If not, go on to learn about components and layouts in the next chapter!

4

Components and Layouts

In this chapter, we will examine the building blocks of Vaadin applications, namely the components. Technologies such as Swing, SWT, Flex, or JSF all provide components that are composed in order to produce a user interface. It is no mystery then that Vaadin also provides them.

Numerous components are available in Vaadin; even more are available as add-ons and we will see in *Chapter 9, Creating and Extending Components and Widgets*, how to build our own. However, the following that are provided out-of-the-box are fundamental:

- UI
- Window
- Label
- Field

Then, we will have a look on how these components can be arranged; this will let us detail the layouts:

- Layout
- Panel

Starting from this chapter, we will build an application that will be used throughout the rest of this book. The goal of this application will be to interact with Twitter.

Thinking in components

Components are at the core of any rich application framework worth its salt.

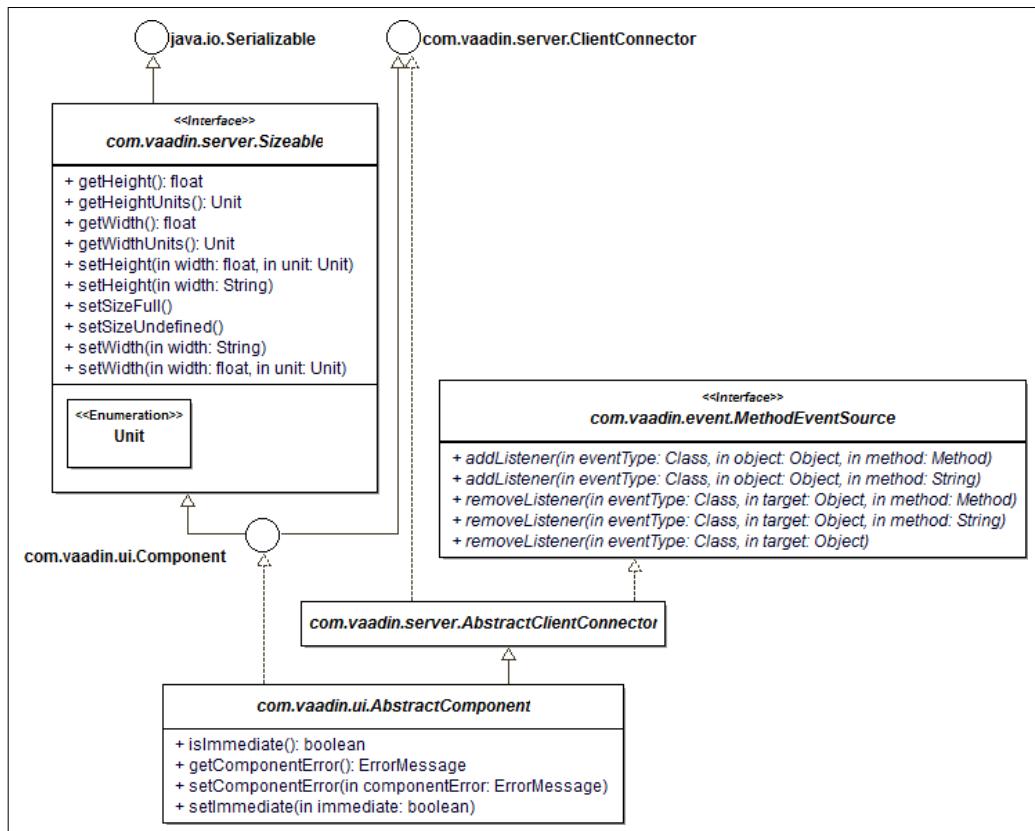
Terminology

In Vaadin, the term **widget** refers to the client-side UI component made with GWT, whereas the term **component** refers to the server-side Java-compiled component, as well as the whole GWT + Java class association.

Component class design

Before diving right into concrete components that we will manipulate in order to create our user interface, let's take some time to analyze the component class hierarchy design in Vaadin.

The following is a simplified components class diagram:



Component

The root of the Vaadin component hierarchy is the `Component` interface. This interface inherits, directly or indirectly, from other interfaces, some of which belong to the Java API and others to the Vaadin API:

- `Serializable`: It is critical that Vaadin components are serializable, as we have seen in *Chapter 3, Hello Vaadin!*, because the UI objects are tied to the HTTP session. Were components not serializable, it would still be possible to store the UI instances in the session, but some application servers would not be able to serialize them between cluster nodes, or even when stopping with active sessions. Therefore, Vaadin tackles this problem right from the start and makes all components serializable.



For example, **Apache Tomcat** uses serialization to store user sessions before terminating. Also, note the **Google App Engine** passes user sessions through cluster nodes using serialization.



From a code point of view, this changes nothing, as `Serializable` is a marker interface and as such, has no methods. The only thing we have to do when inheriting from Vaadin components is to manage class versioning. There are only three options: do nothing, add a `@SuppressWarnings` annotation on classes, or add a serial version unique identifier field (generated or not), like this:

```
private static final int serialVersionUID = 1;
```

For detailed information about class versioning, please look at this Stack Overflow answer: <http://stackoverflow.com/questions/285793/what-is-a-serialversionuid-and-why-should-i-use-it>.

- `ClientConnector`: The `ClientConnector` interface is the contract that all components must adhere to in order for the Vaadin framework to share common state between server-side components and their client-side widgets counterparts. This interface is used by developers who create their own components wrapping GWT widgets or native JavaScript, and will be seen in more detail in *Chapter 9, Creating and Extending Components and Widgets*.
- `Sizeable`: The `Sizeable` interface is of much more interest as it governs the size the components will have. We will describe it in detail in the *More Vaadin goodness* section later in this chapter.

MethodEventSource

MethodEventSource is an interface that is part of the **Observer pattern [GOF:293]**. It represents the subject part of the observer/subject pair. As such, it knows how to register/unregister the third-party observers.

This particular interface and its methods will be under intense scrutiny in *Chapter 5, Event Listener Model*, where we will detail the event model in Vaadin.

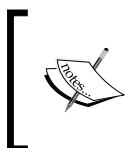
Abstract client connector

AbstractClientConnector will be very much needed by *Chapter 9, Creating and Extending Components and Widgets*; until then, suffice to say it implements all methods described in ClientConnector with an adequate implementation.

Abstract component

The AbstractComponent class implements all the aforementioned interfaces, so that **each and every component in Vaadin will have the same behavior for free**.

This also means that all the custom components we develop, as soon as we inherit from AbstractComponent, will be handled just in the way we intend.



Although this design may seem trivial, it is always a bad surprise when an application does not behave as expected. In Vaadin, all components have the same expected behavior, provided the developer did not implement his/her own, of course.



Properties of AbstractComponent affecting the display include the following:

Property	Type
caption	String
description	String

`caption` is the label associated with the field. It is displayed near the field, on a location depending on the layout (see *Layouts* section), while `description` is a detailed explanation about the field. Description is often displayed as a tooltip.

Note that both properties are defined on the `AbstractComponent` level but only make sense at the `AbstractField` level (see *The text field* section).

Immediate mode

`immediate` is a property of `AbstractComponent`. It governs how events are sent from the client to the server (more details on events will be seen in *Chapter 5, Event Listener Model*).

When `immediate` is set to `true`, the event is immediately sent to the server; when set to `false`, it is buffered in a local queue located on the client-side and sent along the next immediate event. Default value for `immediate` is `false` for most components, and `true` for buttons.

As an example, for a form, if we desire a modern behavior where each field is sent and validated immediately after the value is changed by the user, we set `immediate` to `true` on each field.



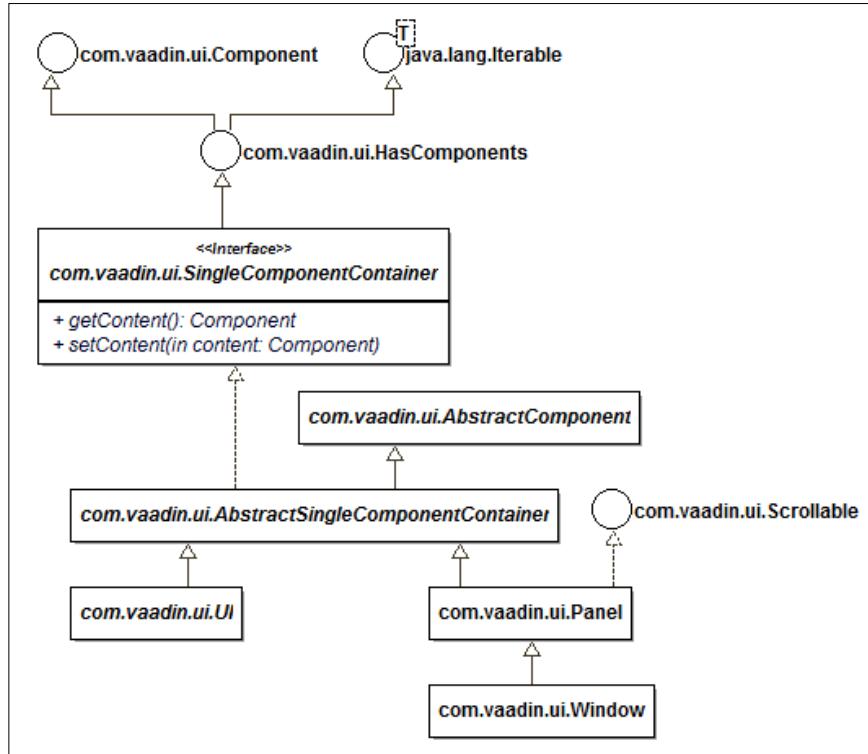
Troubleshooting

The `immediate` mode is the most common source of confusion for Vaadin beginners: if events seem not to be sent to the server, be sure to check the `immediate` property.

UIs

UIs are every rich application's root component. In fact, the Vaadin plugin already created a UI in our previous "Hello Vaadin" project (see *The source code* section from *Chapter 3, Hello Vaadin!*).

The UI class has a rich component hierarchy, as seen in the following diagram:



HasComponents

The `com.vaadin.ui.HasComponents` interface is a Vaadin-specific Iterable template with `Component`. It simply describes a specialized component that may contain other components.

Single component container

The `SingleComponentContainer` interface is a container which can only contain a single child component. Its direct implementation is `AbstractSingleComponentContainer`, which implements all needed methods.

In order to set its child, it declares the `setContent (Component)` method; and symmetrically the `getContent ()` one. This means that only a single component can be set as a child element of the container and it acts as its root.

UI

As seen in *Chapter 3, Hello Vaadin!*, the `UI` class is a specialized component representing the root of its components hierarchy. You may think of it as the canvas upon which added components will be painted and limited by the inner borders of the browser window.

Since `UI` extends `SingleComponentContainer`, we can only add a single component to it. In real-life, however, we may need to add more than one component: this requires the use of an intermediary container that can handle multiple children components. Such intermediaries are generally layouts (see *Layouts* section).

Theming

In Vaadin, a combination of CSS, images, and HTML layouts (refer to the *Layout types* section named later in this chapter) can be brought together and applied to the UI. This combination is called a theme.



Theme creation is a specialized topic and could easily fit a small book in itself. Interested readers can refer to <http://vaadin.com/book/-/page/themes.html> for more information.

Setting a theme can be done on a single UI. In order to do this, decorate the UI with the `@com.vaadin.annotations.Theme` annotation.

Available themes in Vaadin are `reindeer` (the default), `chameleon`, `liferay`, and `runo`. Additional themes can be obtained through the Vaadin directory as add-ons (see *Chapter 8, Featured Add-ons* for more information on Vaadin directory and add-ons).

Panel

The next class is a concrete one, `Panel`. As can be guessed, it represents a panel; by default, panels are displayed in a different background and delimited with visible borders.

`Panel` adds an interface to the `AbstractComponentContainer`, `Scrollable`. `Scrollable` lets us programmatically scroll our panel components. Note that the scrolling unit is the pixel.

Windows

Finally, `Window` is the last class in the window class hierarchy. Note that it is by no means a leaf class, meaning it can be extended should the need arise.



In **UML (Unified Modeling Language)**, leaf classes are classes that cannot be extended. In Java, that translates to final classes.



Windows can be displayed using the `addWindow(Window)` method of the `UI` class, and likewise be hidden using the `removeWindow(Window)` method of the same class.

There is no limit to the number of windows a `UI` can display.



Differences with earlier Vaadin versions

Before Vaadin 7, Windows were used both for top-level windows and pop ups. This led to API inconsistency, as some methods could either be called on one or the other. Vaadin 7 has clarified things: windows are pop ups. Top-level windows are represented by `UI`.



Window structure

Windows are composed out-of-the-box from the following elements:

- A title bar with:
 - A title located at the top-left corner
 - A X icon button located at the top right corner, for closing the pop up
 - A handle located at the bottom right corner, for resizing the window
- A canvas, the same as for UIs, where a single component can be laid out:



Customizing windows

Most of the time, the defaults for structure and behavior of a window won't fit our needs. However, the Vaadin framework allows us to customize both.

Basic configuration

Properties are available in order to customize windows. These are summed up in the following table:

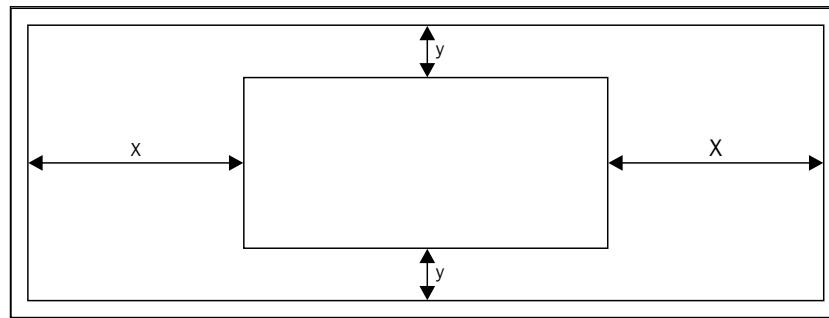
Property	Type	Default value
closable	boolean	True
resizable	boolean	True
draggable	boolean	True

- An unclosable window does not display the X icon button in the title bar. Thus, we have to provide another means to close it; or it will stay there indefinitely!
- An unresizable window does not display the handle at the bottom-right corner.
- Regular windows can be dragged around when the title bar is clicked on. However, we can remove this behavior with the `setDraggable(boolean)` method.

Location

Setting the right location on the user screen is very important from an ergonomic point of view. In order to do this, Vaadin provides a few methods.

Most of the time, centering the window relative to the parent UI fits our needs: just use the `center()` method.



In order to go beyond centering and to set the location of the upper-left corner pop up relative to the upper-left corner of its parent, Vaadin respectively provides the `setPositionX(int)` and `setPositionY(int)` methods. The position parameter's unit is the pixel.

Modality

Modality is the capacity for the window to intercept all events from the user to the underlying UI.

In effect, a modal window blocks all relations to the parent application until it is closed, whereas a non-modal window lets the user interact normally with the application.

By default, windows are non-modal. However, we can change the modality with the `setModal(boolean)` method of the `Window` instance.



Weak modality

Notice that windows are displayed as an HTML `<div>` element on the client-side, not as browser native windows. As a result, modality is enforced, neither by the browser nor the system, but by JavaScript. Hence, we should never rely on this weak modality in order to enforce security constraints, as it can easily be bypassed.

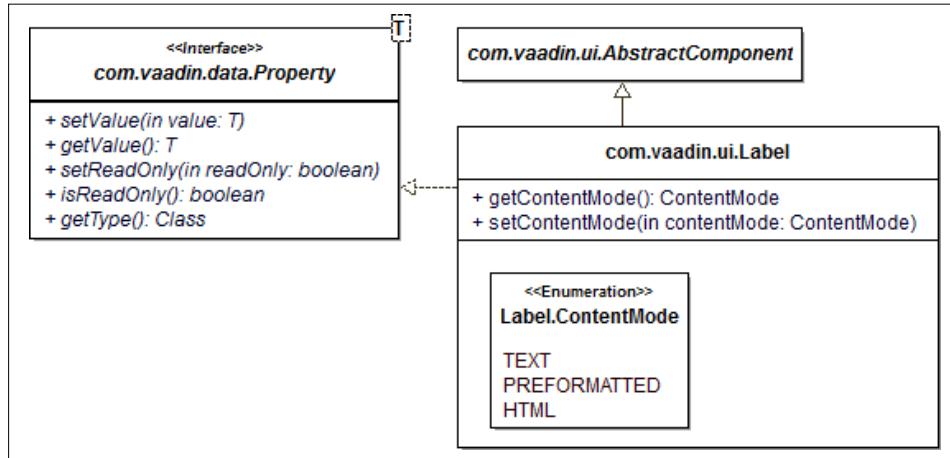
In case of doubt, the `isModal()` method of the `Window` class returns whether the window is indeed modal.

Labels

Labels are widgets used to display non-editable text content. In Vaadin, text fields use associated captions; therefore labels are seldom used (see *Field* section later in this chapter).

Label class hierarchy

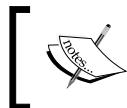
Label is a subclass of AbstractComponent but also implements the Property interface.



As the `Property` interface is implemented throughout the whole Component class hierarchy, it is best to have a look closely at it before going further.

Property

In essence, the `Property` interface simply designates a single value data holder, with accessors.



Note that Vaadin 7, as opposed to previous versions, does take advantage of **Java 5 Generics**: property value uses a template type. Do not forget to use this when needed!

Moreover, there is a read-only property: by contract, calling the `setValue(Object value)` on a read-only instance should throw a `Property.ReadOnlyException`.

Finally, Vaadin provides the `getType()` method to let us query the type stored by the `Property` implementation instance.

Label

As shown in the preceding class diagram, a `Label` is an `AbstractComponent` that implements the `Property` interface.

Formats

However, labels have a distinguishing feature that set them apart from other components; they allow for formatting. Formats are configured using the `ContentMode` enumeration.

Available formats are:

- **Text format:** It is not as simple as it sounds since the final output should be valid HTML. Hence, all HTML entity characters are translated into their equivalent HTML alphanumeric code. For example, the `<` entity is transformed by Vaadin into `<`. This process ensures the user really sees what the message is. This format is hinted at by the `ContentMode.TEXT` enumeration value. It is also the default.



For the complete entities list, visit the following URL:
<http://www.w3.org/TR/REC-html40/sgml/entities.html>

- **Preformatted:** Browsers conveniently lay out HTML paragraphs depending on the window size. If the size changes, then the paragraph will be laid out differently. Yet, it may be required that the paragraph be laid out in a predefined certain way, for example that line breaks occur at certain places. This is the case for computer code, especially for Python for example. For these use-cases, HTML provides the `<pre>` tag which lays out the paragraph exactly the way it is typed. By using preformatted, Vaadin will also use this tag to render the paragraph. This format is governed by the `ContentMode.PREFORMATTED` constant.
- **HTML:** This formatting is similar to the previous one, but also tidies the written HTML, so that the produced output is HTML-compliant. For example, the following server-code `HelloVaadin` will produce this HTML output: `HelloVaadin`. This is used with the `ContentMode.HTML` constant.

Using HTML in Java



Just because something is possible, this does not mean that it should be done. In particular, formatting HTML on the server-side is a bad idea—do it only sparsely. The right way to handle this is through theming.

Formatters are either set in the constructor, or changed later with the `setContentMode(ContentMode)`.

For example, the following code displays a welcome message into our application:

```
Label label = new Label ("Welcome to <i style='color:red'  
title='Vaadin rules!'>Vaadin", ContentMode. HTML);
```

The following screenshot shows the result:



Welcome to **Vaadin**

Vaadin rules!



It should be noted that even though label formatting is a feature brought by the `Label` class, the proper way to style labels is through themes (and thus CSS).

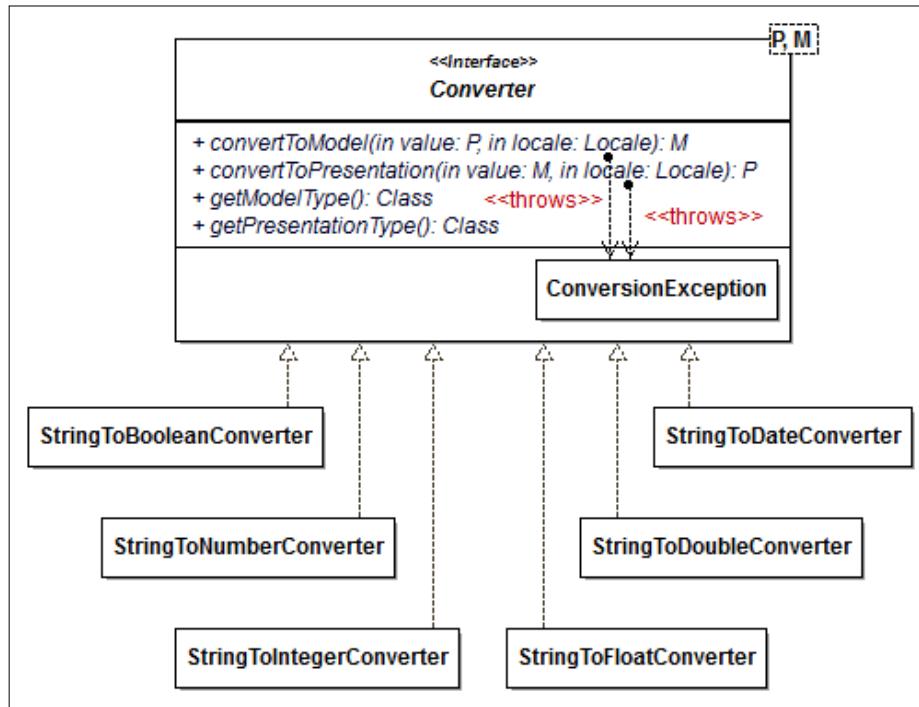
Text inputs

Text fields are the simplest components available to users in order to send data to the server part of an application.

In our case, however, they are also a very good entry point into Vaadin field class hierarchy, as they are devoid of more complex features.

Conversion

While users enter characters that translate into strings, there are times when the underlying data is more finely type. For example, age should be an integer, whereas birthdate should be a date. To handle such data, Vaadin 7 introduces the concept of **converters**: a converter takes a string representation of the data and converts it to the data type and back.



Vaadin provides a lot of converters out-of-the-box in the `com.vaadin.data.util.converter` package. They are pretty self-explanatory:

- `StringToBooleanConverter`
- `StringToNumberConverter`
- `StringToIntegerConverter`
- `StringToFloatConverter`
- `StringtoDoubleConverter`
- `StringToDateConverter`

Adding a converter to a field is achieved with the `setConverter(Converter<T, ?>)` method of the `AbstractField` class. From this point, it is easy to get the converted value with the `getConvertedValue()` method of the same class.



`setConverter()` comes in two flavors: one accepts a converter instance, the other a class. In the latter case, the class should be the class we can use to convert from, not the converter class. Therefore, it will have no effect if it does not belong to one of the above classes.

The following snippet creates a text field that converts raw values to integers. When the user clicks on the button, it outputs two times the converted value in the standard output.

```
final TextField tf = new TextField("Integer");
tf.setConverter(new StringToIntegerConverter());
Button button = new Button("Submit");
button.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        System.out.println(
            ((Integer) tf.getConvertedValue()) * 2);
    }
});
```

If the user types 4 in the field, the standard output displays 8, as expected.

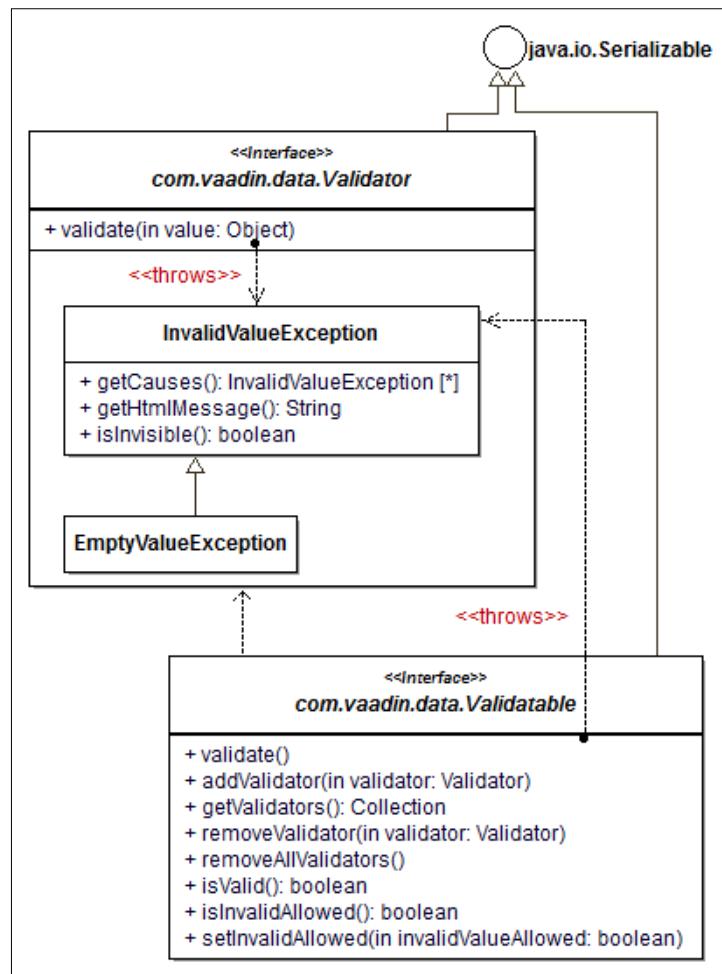
Now, if the user types four, there is no `ClassCastException` for Vaadin throws its own exception beforehand. In this case, the displayed message is **com.vaadin.data.util.converter.Converter\$ConversionException: Could not convert 'four' to java.lang.Integer.**

When implementing custom converters, there are two important requirements to enforce:

- A converter must have no side-effect, and must make no changes to the GUI
- Converter methods, from and to string representations, should be symmetric

Validation

Validation is a major feature of components. As soon as an application needs a user input, there is a need for this input validation. In Vaadin, the validation process is handled by the `Validatable`/`Validator` pair.



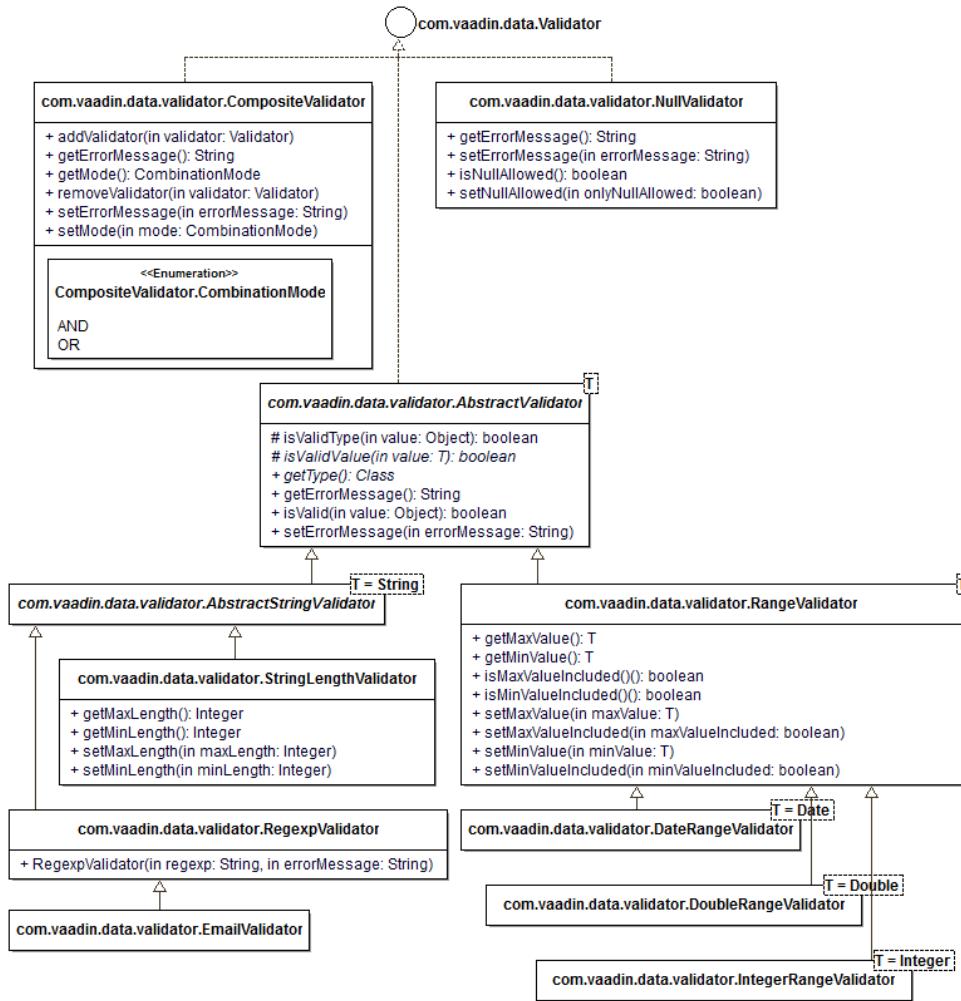
Validator

Validators are specialized objects that fulfill a double purpose:

- Checking if an `Object` is valid.
- Validating an `Object`. It will throw a `Validator.InvalidValueException` if the object is invalid.

Validators hierarchy

The validator hierarchy is shown in the following diagram:



Some concrete validators are available out-of-the-box in Vaadin's API, and should fit most of your needs. These are the following validators:

- `RegexpValidator` is used for validating regular expressions. This validator is very interesting as it is generic enough to be used for ZIP codes, telephone numbers, and similar objects. The regular expression is passed to the constructor, hence making the validator immutable.

[ For more information on regular expressions, visit the following URL:

<http://www.regular-expressions.info/>

For specific patterns and rules used in Java, visit the following URL:

<http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html>

- `EmailValidator` is used for validating e-mails, whereas `StringLengthValidator` is used for string length input.

[ **Regexp, e-mail, and string length validators**

Be aware that everything that can be validated with either an `EmailValidator` or a `StringLengthValidator`, can also be validated with a `RegexpValidator`. However, the former are much easier to use, are more semantically significant, and there is no risk of mistyping regular expressions.

- `CompositeValidator` implements the **Composite pattern [GOF:163]**. Each one holds a list of other `Validator` instances. When asking the composite for validation, it will ask each validator in the list for validation. Now, the composite may run in two different exclusive modes:
 - In **AND** mode, validating succeeds if no referenced validators fail validation. The **AND** mode is the default one.
 - In **OR** mode, validating succeeds if a single referenced validator succeeds.

[ The order in which the validators are executed has no importance whatsoever, as all validators in the list will run, even in **OR** mode a failed validation will not stop other validations.

If our needs go beyond that, and our input cannot be validated with a regular expression, then we should probably use `AbstractStringValidator` as the basis for our brand new validator.

Error message

Validators can be set with an error message at the `AbstractValidator` level. `InvalidValueException` instances are initialized using this error message. `InvalidValueException` accepts a single placeholder that is filled with the field value should the validation fail.



The exception handling mechanism itself will be explained in detail in *Chapter 5, Event Listener Model*.



For example, the following code snippet will display an error message relative to the value when the submit button is clicked:

```
// window is the main window
TextField tf = new TextField("Email");
EmailValidator validator =
    new EmailValidator("{0} is not an email");
tf.addValidator(validator);
Button button = new Button("Submit");
VerticalLayout layout =
    new VerticalLayout(tf, button);
layout.setMargin(true);
setContent(layout);
```



Validatable

A **Validatable** stands for objects that know how to validate their input based on a collection of internal validators.

Like **Validator**, **Validatable** has two main methods that must be kept in sync:

- `isValid()` that checks if an Object is valid
- `validate()` that throws a `Validator.InvalidValueException` if the object is invalid

Unlike **Validator**, it delegates this logic to its underlying validators.



Validatable coherence

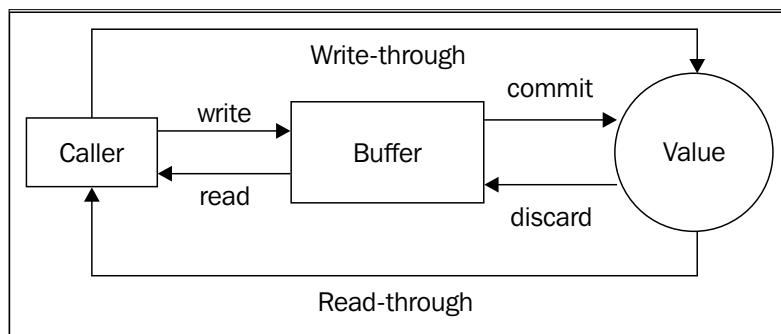
It is mandatory, by contract, that the `isValid()` and `validate()` methods be coherent with each other. Vaadin's implementations respect this rule as `AbstractValidator.validate()` effectively calls `isValid()`. However, when directly implementing **Validatable**, take care to enforce this in the code.

Change buffer

In computer software, a buffer is a zone that is used to temporarily store data.

In Vaadin, the `Buffered` interface represents an object that can flush or cancel changes made to its buffer to the underlying datasource object, but has the option to override this behavior altogether and ignore the buffer in one or both ways (read and write).

The following schema illustrates this:



- In a read-through mode, the value read from the buffer is always in sync with the underlying value
- In a write-through mode, the new value is immediately updated



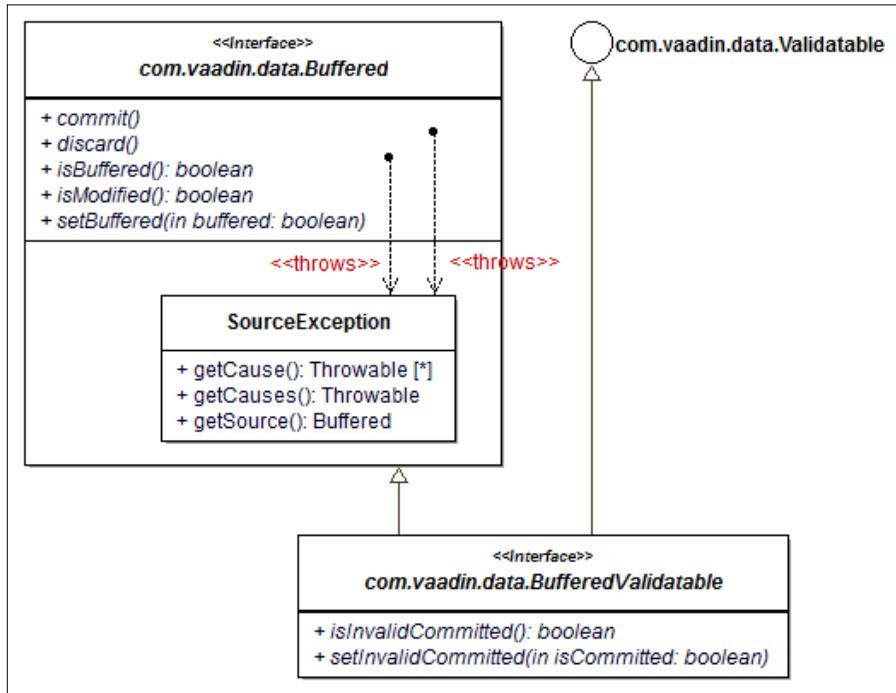
Note that each behavior is completely independent. A buffered object may be write-buffered but read-through or the opposite, read-buffered but write-through. Beware that those combinations are highly unorthodox, even if sometimes desired, and may result in puzzling behavior at first glance.

The `Buffered` interface is very similar to Relational Databases Management systems as the latter also uses a buffer to handle transactions and isolation levels for SQL Data Manipulation Language (`INSERT`, `DELETE`, and `UPDATE`) statements:

- `commit()` will update the real value with the value held in the buffer
- `discard()` will replace the buffered value with the real one, just like a rollback statement

Buffered and validatable

Input fields can be at the same time buffered and validatable. Vaadin introduces the `BufferedValidatable` to the hierarchy, which unsurprisingly inherits, from both `Buffered` and `Validatable`.



This interface just adds to its superinterfaces how to set/unset, that the current buffer invalid value has been committed, and query this information through the `invalidCommitted` property.



These methods are used by Vaadin's internals to verify if it needs to send change value events, they can comfortably stay in the dark.



Input

From this point on, Vaadin weaves display features into the data class hierarchy.

Focusable

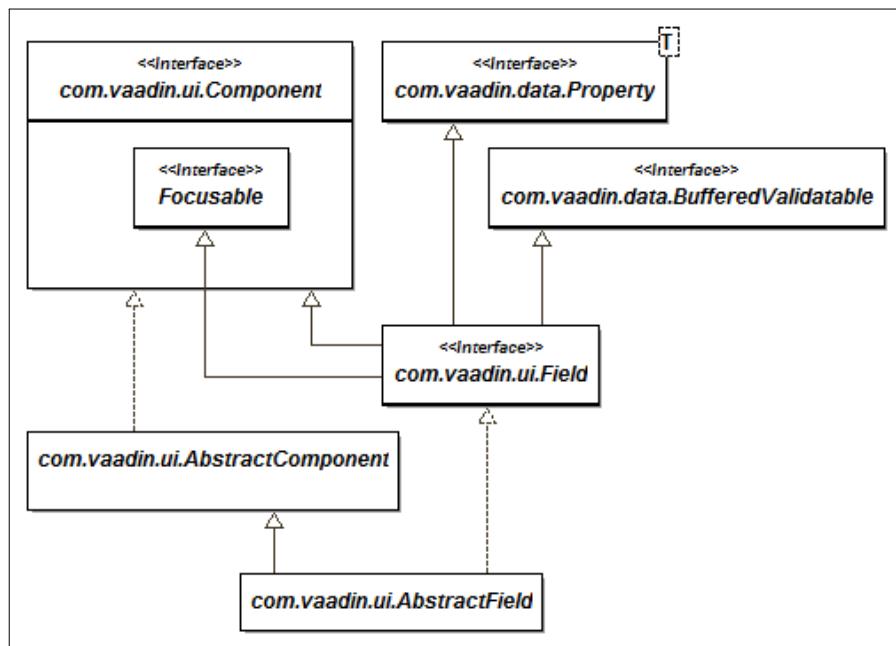
As inputs are meant to be displayed, they can also receive focus. Component describes the Focusable interface that has the following three methods:

- `setTabIndex(int)` to set the tabulation order
- `getTabIndex()` to get it
- Finally, `focus()` to programmatically give focus to a particular component

A thing worth noticing is that although Focusable is declared in Component, not all components are focusable, though fields are. In particular, labels, which are components but not fields, won't be able to receive focus.

Field

The Field interface inherits from BufferedValidatable, Focusable, and Component. Moreover, it adds features that characterize input fields.



In Vaadin, the following properties are inherent to a field:

Property	Type
required	boolean
requiredError	String

`required` is an indicator set to `true` if the field should throw an `EmptyValueException` if it is empty, while `requiredError` is the associated message. These are propagated as `InvalidValueException` instances, seen previously in the above sections.

[ As `Field` is an interface, it has no property *per se*, yet getter/setter combinations are handled by properties in the `AbstractField` child class so it is not far-fetched to call them properties even at the interface level.]

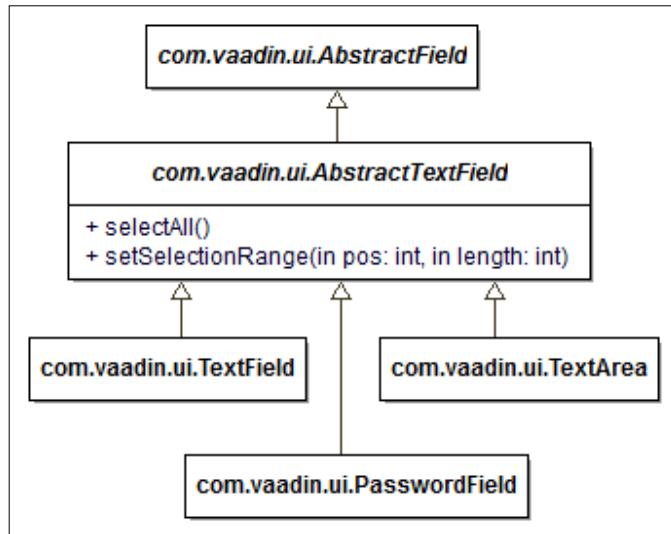
`AbstractField` is a straight abstract class that provides implementation of the `Field` interface. It also inherits from the `AbstractComponent` class, like `Label`, but can hold a value thanks to the `Property` interface.

Also, `AbstractField` provides the `isEmpty()` method to check whether the property managed by the field is `null` (or empty in the case of text fields).

There is not much more to say about it, except that all input components such as text field, select box, and so on provided by the Vaadin API inherit either directly or transitively from this abstract class. If we need to create our own component, then it should be the first class to consider extending, although it is by no means required.

The text field

The text field class hierarchy in Vaadin begins with `AbstractTextField`, which defines additional features to enhance `AbstractField`.



Descending in the hierarchy, there are specialized classes for some use-cases:

- Simple text field when nothing more is needed: this will translate on the browser to an HTML input of type `text`
- Password field that hides the characters typed in it which will be displayed with an HTML input of type `password`
- Text area that is represented by an HTML `textarea`

However, the real power lies in the `AbstractTextField` that factorizes common behavior and properties. The following table lists those properties that are described in the following table:

Property	Type	Default value	Misc.
<code>columns</code>	<code>int</code>	0	Set to 0 for implicit calculation
<code>cursorPosition</code>	<code>int</code>	N/A	
<code>inputPrompt</code>	<code>String</code>	<code>null</code>	
<code>maxLength</code>	<code>int</code>	-1	Set to -1 for unlimited length
<code>nullRepresentation</code>	<code>String</code>	<code>null</code>	
<code>nullSettingAllowed</code>	<code>boolean</code>	<code>false</code>	

Null

For a computer programmer, `null` and empty values are not the same thing and may have a very different meaning (or not) both in the code and in the database.

Some software handles the case while others do not. In the former case, the most well-known example is when we have to use a GUI to set a `NULL` value into the database.

The good news is that, Vaadin lets us handle `null` values differently from empty ones. On the other hand, if this is the desired behavior, the `null` value must be assigned a string representation. This means that this string will be interpreted by Vaadin as `null`, and it won't be available as a real string value anymore. For this reason, it is very important to assign only strings that have no meaning for the user. Examples of such meaningless strings include, but are not limited to: `<NULL>`, `<null>`, `#null`, or an empty string.

Note that this feature may only be used when `null` values are indeed allowed for the field: it is not the case by default and must be activated if necessary with `setNullSettingAllowed(true)` on the text field instance.

Input prompt

An input prompt is the text that is shown in the field itself as a hint for accepted values to the user. As soon as the user starts typing into the field, the prompt is hidden. The prompt font is usually shown with less visibility than the input font itself in order to have a clear discrimination between prompt and input, as shown in the following screenshot:



Setting the input prompt is simply done by calling the `setInputPrompt(String)` method. We can also query it with `getInputPrompt()`.

[

The value of the input prompt

]

Not only is setting an input prompt in your own applications an interesting alternative (as well as a cheaper one) to a carefully documented help, it is also a UI design pattern. Refer to <http://ui-patterns.com/patterns/InputPrompt> for more information.

Cursor

Vaadin lets us programmatically manage the cursor position within a field. This is simple and straightforward. Note that calling the `setCursorPosition(int)` method will also give focus to the field the method is called on.

We can also get approximate cursor position from the last time of UI synchronization.

Selection

We can select a text field's content with either of the following methods:

- `selectAll()` selects the whole of the text content
- `setSelectionRange(int, int)`, where the first parameter is the initial position index and the second the length to select, selects characters starting from the position (included and beginning with 0)

For example, the following code snippet will select "23":

```
TextField tf = new TextField();
tf.setValue("123456789");
tf.setSelectionRange(1, 2);
```

A "real-world" text field example

Can you guess what the next code snippet does?

```
TextField tf = new TextField("Age");
tf.setInputPrompt("Please enter age");
tf.setImmediate(true);
tf.setConverter(new StringToIntegerConverter());
IntegerRangeValidator validator =
    new IntegerRangeValidator("Age must be below 100", 1, 99);
tf.addValidator(validator);
```

Here is the answer: it creates a text field, complete with label and input prompt. This field accepts only integer values that are to be between 1 and 99. Conversion and validation happen as soon as the user leaves the field.

More Vaadin goodness

Many customizations are available in Vaadin to implement your customer requirements. Some of them do not need components.

Page

In Vaadin parlance, `Page` represents the browser window (or tab) whereas `UI` stands for the canvas inside it. Some configuration is available through pages, as shown in the following properties:

Property	Type
<code>title</code>	<code>String</code>
<code>location</code>	<code>URI</code>
<code>uriFragment</code>	<code>String</code>

Getting a handle on the page is easy as pie: `UI` offers the `getPage()` method. Alternatively, just use the static method `Page.getCurrent()` anywhere in the code.

 **Get current**
The `getCurrent()` method is available on many classes (including `UI`): when you are stuck when trying to get a handle on a specific object, check whether such a method is provided before trying to get clever.

Title

Pages can be set with their title. This title will be written in the generated HTML output under the `<title>` tag and will be visible in the title bar of most browsers. To do so, just annotate the UI with the `com.vaadin.annotations.@Title` annotation.

Alternatively, if the title has to be dynamic and cannot be set in stone, call the `setTitle(String)` method on `Page`.

Navigation

Under normal conditions, Vaadin implements the **Single Page Interface**, meaning that during the entire application lifecycle and throughout UI changes, Vaadin does not change the URL of the first request that launched it.



In order to know more about the Single Page Interface, refer to the following URL:

https://en.wikipedia.org/wiki/Single-page_application



To navigate to locations that are not served by the current Vaadin servlet (may they be under the same context-root or completely unrelated URIs), the `setLocation(String)` method (or `setLocation(URI)`) is very useful.

For example, the following snippet sets the browser location to a well-known search engine:

```
Page.getCurrent().setLocation("http://www.google.com");
```

URL fragment

Single Page Interface is an acceptable default for standard applications. Yet for web applications, users are used to bookmark different pages in order to easily find them later—it is the legacy of page-flow applications.

More importantly, some applications have real need for fine-grained URL, like online stores where each item can be set its own URL. This also let these different items be referenced separately by search engines such as Google.

In this case, the method `setUriFragment(String)` of the `Page` class is useful in that it lets developers append a URI fragment to the URL application. On the contrary, `getUriFragment()` lets us accede to the URI fragment, if there is one.

As an example, let us consider an online shop. Each item of the shop displays a list of related items. Hyperlinks on such related items include a URL fragment that may be used to display the associated item full-size:

```
String uriFragment = Page.getCurrentPage().getUriFragment();  
long itemId = transcodeFragmentToId(uriFragment);  
Item item = loadFromDatabase(itemId);
```

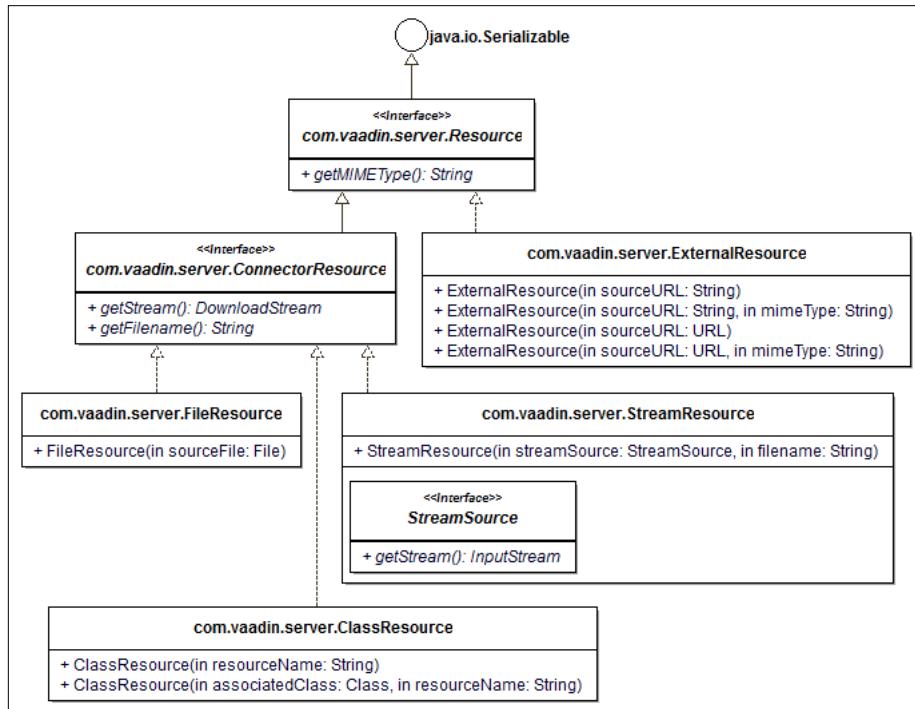
Accessing a component that contains such code will load the desired item from the database, as expected.

Third-party content

Sometimes content sent to the user should not be related to user interface, but to third-party content, whether in text or binary format. This is the case when the user requests an image, a PDF, or even a bare HTML document that should not be handled by the Vaadin framework.

Resources

In order to cover these cases, Vaadin provides an abstraction: Resource.



The `Resource` interface represents something provided to the terminal for presentation. How the terminal really handles the display is left to it.

In Vaadin, resources may come from two different locations:

- From inside the application (or at least from where the application is located, in the case of files):
 - Resources on the classpath
 - Resources from streams accessible by the code
 - File resources located on the server filesystem
- From outside the application, for example, resources accessible by an URL.

Browser window opener

These resources can be displayed in a native pop-up window and Vaadin provides a dedicated extension named `BrowserWindowOpener` to take care of that.



Extensions will be detailed in *Chapter 7, Core Advanced Features*. For now, let us say that an extension is a way of providing additional client-side behavior to an existing component.

Vaadin accepts the popup window name. If a window is already opened with the specified name, then Vaadin replaces its content with the new resource.

Some window names hold a special meaning. Those are the same as in JavaScript's `window.open()` and are summarized here for ease of reference:

- `_blank` always opens a new window, even if a previous blank named window is already open.
- `_self` indicates the current window, hence using it is equivalent to not using a window name at all.
- `_parent` and `_top` reference respectively the frame's parent and the frameset. If frames are not used, then it is the same as `_self`.

For example, the following snippet creates a button that opens a pop-up window displaying the Packt homepage each time it is clicked:

```
Button button = new Button("Click me");
try {
    URL url = new URL("http://www.packtpub.com/");
    Resource resource = new ExternalResource(url);
```

```
BrowserWindowOpener opener = new BrowserWindowOpener(resource);
opener.setWindowName("_blank");
opener.extend(button);
} catch (MalformedURLException e) {
    throw new RuntimeException(e);
}
```

User messages

Applications generally need to inform users. For example, when a user deletes or updates an entry, it is a good practice to let users know their operation succeeded or not.

Traditionally, there have been two ways to communicate these facts to the user for web applications:

- Opening an information box through JavaScript. This pop-up is modal, that is, it blocks inputs outside it and forces the user to acknowledge the displayed message.
- A space is reserved for a message on the screen, most of the time a banner at the top of the page. It may also be used for error messages: standard information messages are displayed in a neutral color, errors in red.

Vaadin notification system employs yet a third strategy: the framework displays the message in an overlay on top of the UI.

The notification class

In order to do that, the `com.vaadin.ui.Notification` class is used.

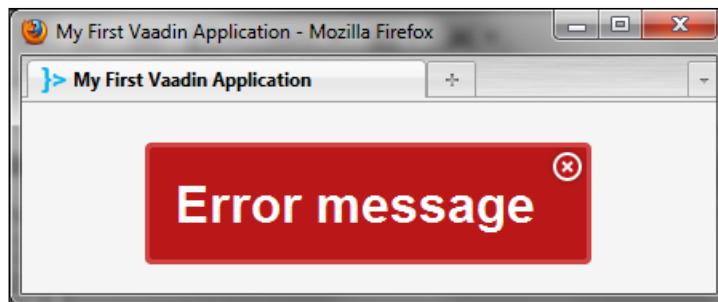
From a graphical point of view, a notification displays elements in a horizontal layout in the following order, from left to right: an icon, a caption, and description.

Each one corresponds to a Java property, thus having both a getter and a setter. The following table sums it up:

Property	Type	Constr. argument	Optional	Def. value
caption	String	yes	mandatory	N/A
description	String	yes	optional	null
icon	com.vaadin.terminal. Resource	no	optional	null

A notification is defined by its type. Currently, there are three types denoting different severity level and one special type. The former are, in order of gravity:

- Information, also known as humanized notifications, denotes fairly unimportant content, for example, operation acknowledgement such as "Entry deleted". Note that an information message fades as soon as Vaadin detects a keyboard input, or a mouse click, or move from the user.
 - Information notifications are created using `Type.HUMANIZED_MESSAGE` enumeration value in the `Notification` constructor, or leaving it unspecified, as it is the default style.
- Warning notifications are similar to information messages, except they fade only when a delay has passed or after some user interaction. In this way, they are more noticeable than information messages.
 - Warning notifications are created using the `Type.WARNING_MESSAGE` enumeration value in the `Notification` constructor.
- Error type notifications behave more like modal alert boxes. Their message is shown until the user clicks on the notification (the X in the top right corner is only for show).
 - In order to create error messages, use `Type.ERROR_MESSAGE`.



- Finally, tray notifications are used for low-severity messages. However, unlike the other notifications, they will stack in front of one another, always displaying the most recent. Clicking on the visible tray notification will dismiss it and display the underlying notification. This will continue until there are no more notifications to display.
 - Tray notifications are created using the `TYPE_TRAY_NOTIFICATION` constant in the `Notification` constructor.



Beware of humanized notifications

Be aware that if the user interacts in any way with the application while the notification is shown, he/she will likely be unable to read the content of the message. Therefore, it is advised to restrain the use of humanized notifications, for showing users that the system reacted to whatever they did, and should not be used for anything, but acknowledging that whatever operation they requested actually was performed.

Notifications additional properties

Tray notifications are displayed at the lower right corner; other notifications are centered on the screen. It is however possible to override this position with the `setPosition(Position)` method.

In addition, the delay for the notification can be configured as can the style name. All three, position, delay, and style name are Java properties.

Property	Type	Constructor argument	Mandatory / optional	Default value
position	Position	No	optional	Depends on the type
delayMsec	int	No	optional	1500 for Type . WARNING_MESSAGE 3000 for Type .TRAY_ NOTIFICATION
styleName	String	No	mandatory	Depends on the type ("tray", "warning" or "error")

Here are the different available positions for tray notifications:

Position.TOP_LEFT	Position.TOP_CENTER	Position.TOP_RIGHT
Position.MIDDLE_LEFT	Position.MIDDLE_CENTER	Position.MIDDLE_RIGHT
Position.BOTTOM_LEFT	Position.BOTTOM_CENTER	Position.BOTTOM_RIGHT

Displaying notifications

The standard steps to display notifications are the following:

- Create a new notification instance
- Customize the instance
- Call the `show(Page)` method on the notification instance

As an example, the following snippet displays a welcome message:

```
Notification notification =
    new Notification("Welcome Vaadin", "It's our first
        application");
notification.show(Page.getCurrent());
```

However, we had no need to customize the notification, neither its position nor its display delay. In fact, this is the use-case encountered more frequently: just create a plain notification and display it.

Therefore, Vaadin adds some very productive static methods to the `Notification` class:

- `show(String)`: Shows a standard humanized notification
- `show(String, Type)`: Shows a notification, letting us choose the type
- `show(String, String, Type)`: Shows a notification complete with configurable description and type

Hence, the previous snippet can be shortened to the following:

```
Notification.show("Welcome Vaadin", "It's our first application",
    HUMANIZED_MESSAGE);
```



Use static methods when possible

Of course, there is no magic involved: Vaadin creates the notification instance for us. Yet, as creating it ourselves has no interest if we just need standard behavior, it is well advised to use these static methods whenever possible. It cleans the code, if only a little, and lets us focus on the real meaningful parts.

Laying out the components

How the UI components are placed on the browser is reliant on the following two factors:

- The components size
- Their layout

Size

Previously, when we stared at the `Component` interface, we noticed it inherited from `Sizeable`: let us have a look at that now. Size in Vaadin is governed by two properties: `unit` and `value`.

Available units are exactly the same as those defined by the **W3C CSS1** specifications.



Visit <http://www.w3.org/TR/REC-CSS1/#units> for detailed information about W3C CSS1 specifications.

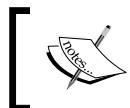


Units are governed by the `Sizeable.Unit` enumeration. As a quick reminder, these are summed up in the following table:

Unit	Type	Symbol	Description	Equivalence	Constant
inch	absolute	in			INCH
centimeter	absolute	cm		2.54 cm = 1 inch	CM
millimeter	absolute	mm		1000 mm = 1 m	MM
point	absolute	pt		12 pt = 1 pica	POINTS
pica	absolute	p		6 picas = 1 inch	PICAS
pixel	absolute	px			PIXELS
em	relative	em	Proportion of the letter in regards to the point size of the current font		EM
ex	relative	ex	Proportion of the letter in regards to the height of the letter x		EX
percentage	relative	%	Relative to the available space for the component As an example, in a vertical layout with 3 components 50% for one of the components means 50% of the available slot, which is 33% of the parent layout size)		PERCENTAGE

Setting the dimension of a component, either the height or a width, is done with either of the following two methods:

- `setHeight/setWidth(float, Unit)`: In this case, both the value and the unit are set. The latter parameter is taken from the constants defined in the `Unit` interface, as described in the previous table.
- `setHeight/setWidth(String)`: In this case, the Vaadin framework parses the string value in order to compute the desired length and its unit. The string is parsed by following exactly the same rules as followed for CSS 1 length units' specifications.



Calling `setHeight/setWidth()` with either the empty string or null as an argument will clear the length value and set the unit to pixel for the desired dimension.



There are also two shortcut methods:

- `setSizeFull()` is the equivalent of calling `setHeight("100%")` and `setWidth("100%)`
- `setSizeUndefined()` clears both the height and width information

For example, these two lines of code are equivalent:

```
myComponent.setHeight(100, Unit.PIXEL);
myComponent.setHeight("100px");
```



Which style to choose?

Using one or the other is more or less a question of taste. Using strings has the disadvantage of mistyping, but you could also use an unwanted unit constant. It is advised to use the style that suits you best but in a consistent manner throughout the project.



Layouts

In Vaadin, layouts are components: they all inherit from `AbstractComponentContainer`. In this aspect, layouts are full-fledged components and have an intrinsic size.

About layouts

The web browser terminal translates components into HTML elements. For layouts, this means each of them generates a `<div>` tag.

Depending on several factors (including user machine performance, web browser type and version, server performance, and network latency), users may experience unresponsive behavior when resizing a native window displaying a Vaadin application, if there are too many nested layouts.



There is already some nesting done by Vaadin, so it is better not to add more than three extra nested levels. Beyond that, you should take care to know your audience (intranet, extranet), and to extensively test your UI. Failing all that, the best move is to migrate from simple nested layouts to a more complex layout, `CustomLayout` being the ideal candidate (see section *Advanced layouts* later in this chapter for details).



Component container

The interface named `ComponentContainer` represents a component that may hold more than one other component. As such, it has methods that manage child components and more precisely:

- Add a single child component (or a bunch of children components)
- Remove a child component
- Replace a child component
- Remove all child components
- Move all child components from one component container to another

The abstract class that provides these implementations is `AbstractComponentContainer`. It also inherits from `AbstractComponent` and as such, has all features seen earlier.

Layout and abstract layout

Layout is the base interface for all layouts. It just knows how to use margin or not, on all four sides, or side-by-side. The space used for the margin is dependent on both the terminal and the theme used.

AbstractLayout is the straightforward implementation of Layout and inherits from AbstractComponentContainer.

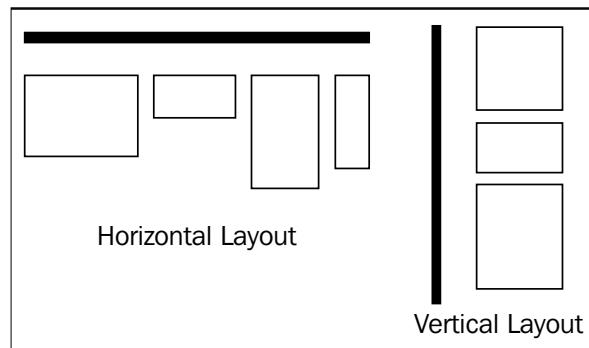
Layout types

Vaadin provides many different layouts out-of-the-box; there will always be one that will fit your needs. If you are really stuck with a particularly complex graphic design, then just take a look at CssLayout which is described in this section. The following are lists available types of layout:

Simple layouts

Simple layouts are efficient and let you forget about HTML and CSS.

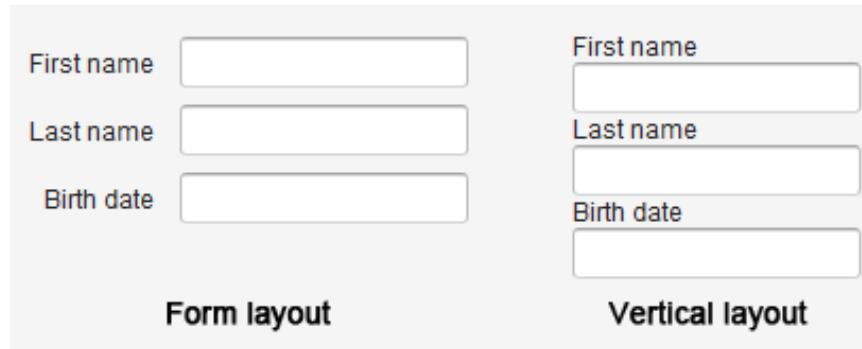
- **Horizontal and vertical layouts:** Horizontal and vertical layouts position child components respectively in a horizontal and vertical way. Those layouts are the simplest we can use: they just put components next to one another in a single neat row/column, regardless of the user screen's dimensions. It is the responsibility of the web browser to provide the means to display UI parts that are out of view, usually with the help of scrollbars.



- **Grid layout:** Of more interest is the GridLayout. As its name implies, Vaadin will lay out the child components in a grid-like fashion.

GridLayout's constructor needs both the number of columns and the number of rows.

- **Form layout:** Fields captions are usually displayed on the top of the relevant field; it is the default location. However, forms are usually label-field pairs where the label is displayed left of the field and each pair stacked vertically. Vaadin emulates this presentation when using the form layout. This is shown in the following screenshot:



Advanced layouts

When our needs are more elaborate than the preceding simple cases, Vaadin still has the answer. In fact, HTML and CSS let you have a more precise hand over the overall design of your screens.

Leaky abstractions



Using these layouts is a case of leaky abstraction: in most cases, Vaadin shields us from the mechanics and the technology of lower layers. Previous layouts generate an awful bunch of nested divs, which makes DOM heavy and slow to update. On the contrary, absolute, CSS, and custom layouts force us to dirty our hands with gritty details but give much more control over what is generated client-side.

- **Absolute layout:** The Absolute layout translates to absolute CSS positioning. As a reminder, it has four possible attributes: `top`, `bottom`, `left`, and `right`. Each attribute can be affected a length, containing both a value and a unit as seen in *Size* earlier in this chapter, and the browser will draw the component at the exact position relative to the screen.

In order to make that work, `AbsoluteLayout` adds the `addComponent(Component, String)` where the second argument is the absolute position written respecting the CSS specifications.

For example, the following code will display the label in the bottom left corner:

```
layout.addComponent(new Label("Made with Vaadin"),
    "left:10px;bottom:10px");
```

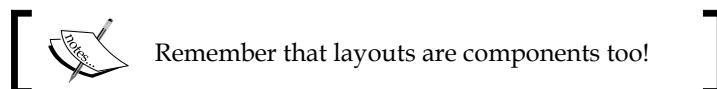


- **CSS layout:** The CSS layout goes even further than the previous absolute layout, in that it renders its children component in the same HTML <div> and lets the CSS position them on the final page.

A whole book could be devoted to CSS specifications and Vaadin theming targeted at designers. For mere developers like us, the only thing to know is that each Vaadin component is affected a CSS class in the generated HTML. In order to effectively use CSS, we have to tell the designer how the class name is computed.

The rule is very simple: the class name is "v-" concatenated with the name of the Java component in lowercase. The following table shows some examples of this rule:

Java component	CSS class name
TextField	v-textfield
Button	v-button
CssLayout	v-csslayout



- **Custom layout:** Vaadin can also be used as a templating engine with the help of the `CustomLayout`. In order to do this, carry out the following steps:
 1. First, find the desired HTML template to feed the layout, whether on the fly by loading the desired resource from an input stream, or through a theme template.



With theme templates, the template is fetched either from the web application's root or from an accessible JAR. In both cases, the layout is referenced under VAADIN/themes/<CURRENT_THEME>/layouts/<LAYOUT_NAME>.html.

2. Then, set the placeholders. Placeholders must be the `div` elements and should be set a `unique location` attribute.
3. Finally, add components to the `CustomLayout` by using `addComponent(Component, String)`. Vaadin will replace the previously defined `div` with the corresponding component using the second parameter as the key.

As an example, the following snippet of code warmly welcomes Vaadin:

```
// window is main window
CustomLayout layout = new CustomLayout
(new ByteArrayInputStream("<body><h1>Hello</h1>
<div location='who' />".getBytes()));
layout.addComponent(new TextField("name", "Vaadin!"), "who");
window.setContent(layout);
```



Choosing the right layout

It may seem a banal, but the right layout is the one that fits your needs. However, there are some general guidelines:

- It is a good practice to begin with a simple layout, that lets you forget HTML and CSS specifications, and then progress to advanced ones if there is a real added-value
- If you come from a client-server background (Swing, SWT, or something else), then realize there is a performance penalty when nesting too many levels of layout
- If only a small part of the screen is complex, then prefer a top-level simple layout that nests an advanced one over a single advanced one

Of course, these principles are too broad to be one-size-fits-all. Nonetheless, they should be useful most of the time: don't have any scruples in adapting them to each specific situation, though.

Split panels

In Vaadin, split panels are designed as component containers. They behave as their client-server counterpart: they contain two components separated by a split bar.

Available properties are as follows:

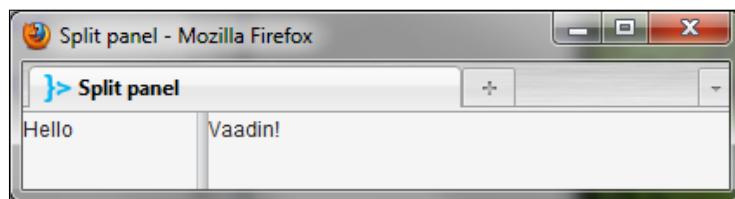
Property	Type	Default value
firstComponent	Component	null
secondComponent	Component	null
locked	boolean	false

Regardless of the locked indicator, we can programmatically set the position of the split bar with the `setSplitPosition()` method. It accepts the following parameters:

- The position value as a float, required.
- The position unit as a Unit (optional and defaults to PERCENTAGE). See the section named *Size* for a refresher on available units.
- A reverse indicator as a boolean, optional. If not specified or if set to false, then the position is measured by the first region, else it is by the second.

For example, the following code fragment displays a locked vertical split panel where the first pane takes 100 pixels and the second the rest of the screen:

```
AbstractSplitPanel panel = new HorizontalSplitPanel();
panel.setFirstComponent(new Label("Hello"));
panel.setSecondComponent(new Label("Vaadin!"));
panel.setLocked(true);
panel.setSplitPosition(100, PIXELS);
```



Bringing it all together

Before jumping into the next chapter, in order to learn from a real world example, we will begin creating an application in Vaadin. This application will be our main thread in bringing together all we learned of the framework.

Introducing Twaattin

Our application's focus will be to provide an interface to Twitter. As it will be developed with the Vaadin framework, it is only natural to name this brand new application Twaattin.

The Twaattin design

In this chapter, we will focus our design on what we learned here. From a component point of view, there will be the following two components:

- Twaattin won't let us connect to Twitter without first requesting a login and a password. At startup, Twaattin presents a login screen.
- The Twitter timeline is shown as the second screen.

The login screen

The login screen contains a login field, a password field, and a submit button.

If the login process fails, for whatever reason, then Twaattin does nothing but display an error message. If the login is successful, then Twaattin displays the Twitter timeline and shows a confirmation message.

The main screen

The main screen should aggregate tweets, stacked from the more recent to the more ancient vertically.

Let's code!

OK, the design is done; now we will begin the real work.

[ **Twaattin sources**
All sources are available on GitHub at this address:
<https://github.com/nfrankel/twaattin>.
Feel free to fork them to practice on your own.]

Project setup

Create another project like the one we did in *Chapter 2, Environment Setup*, although there are some changes:

- Twaattin should be the project name
- Use com.packtpub.learnvaadin.twaattin for the base package name
- The UI class name is TwaattinUI
- Finally, the context-root is better changed to twaattin

Project sources

The sources consist of files updated or created by hand.

UI

```
package com.packtpub.learnvaadin.twaattin.ui;

import static com.vaadin.server.Sizeable.Unit.PIXELS;

import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.HorizontalSplitPanel;
import com.vaadin.ui.UI;

public class TwaattinUI extends UI {

    private static final long serialVersionUID = 1L;

    @Override
    public void init(VaadinRequest request) {

        HorizontalSplitPanel panel = new HorizontalSplitPanel();
```

```
        panel.setFirstComponent(new LoginScreen());
        panel.setSecondComponent(new TimelineScreen());

        panel.setSplitPosition(300, PIXELS);

        setContent(panel);
    }
}
```

At this point, we'll present both components side-by-side in a horizontal split panel since we have not seen any way to interact with a Vaadin application.

The login screen

```
package com.packtpub.learnvaadin.twaattin.ui;

import com.vaadin.ui.Button;
import com.vaadin.ui.FormLayout;
import com.vaadin.ui.PasswordField;
import com.vaadin.ui.TextField;

public class LoginScreen extends FormLayout {

    private static final long serialVersionUID = 1L;

    private TextField loginField = new TextField("Login", "packtpub");
    private PasswordField passwordField =
        new PasswordField("Password");
    private Button submitButton = new Button("Submit");

    public LoginScreen() {

        setMargin(true);

        addComponent(loginField);
        addComponent(passwordField);
        addComponent(submitButton);
    }
}
```

Wow, that is our first real screen! There are a few occurrences worth noticing:

- First, as our UI references components, the login screen in turn references each component as private attributes.
- Objects instantiations are done when declaring the attribute. Doing it here or in the constructor is largely a matter of taste.

The timeline screen

```
package com.packtpub.learnvaadin.twaattin.ui;

import com.vaadin.ui.Label;
import com.vaadin.ui.VerticalLayout;

public class TimelineScreen extends VerticalLayout {

    private static final long serialVersionUID = 1L;

    public TimelineScreen() {

        setMargin(true);

        fillTweets();
    }

    public void fillTweets() {

        for (int i = 0; i < 10; i++) {

            Label label = new Label();

            label.setValue("Lorem ipsum dolor sit amet, consectetur "
                + "adipisicing elit, sed do eiusmod tempor incididunt "
                + "ut labore et dolore magna aliqua. Ut enim ad minim "
                + "veniam, quis nostrud exercitation ullamco laboris "
                + "nisi ut aliquip ex ea commodo consequat.");

            addComponent(label);
        }
    }
}
```

For now, the timeline window is just a placeholder: a bunch of labels stacked vertically. In all cases, there is no way it can be displayed at the time.

Summary

In the first section of this chapter, we have seen the building blocks of the Vaadin framework, namely components. This was a good reason to detail the Component class hierarchy, which was an excuse to have a look at the following classes and interfaces:

- Component is the root interface for widgets, and has important ancestors.
 - MethodEventSource to add listeners to the widget
 - Paintable to make the component displayable on the terminal
 - Sizeable to let the widget be resized
- UI is the base class to display components. It introduced us to ComponentContainer, a component that can hold other components.
- Label is the simplest widget in Vaadin. It showed the Property interface, a way to decouple the widget itself from its value.
- Text field is a plain input field. Yet, we discovered several features not present in plain labels, brought by its hierarchy:
 - Validation with the Validator/Validatable pair
 - Change buffering with the Buffered interface
 - Focus feature brought by Focusable
 - Finally, the Field interface and its properties

Although we only brushed the surface of the variety of components Vaadin offers, looking at each node in the class hierarchy allows us to easily understand future as-yet-unseen widgets.

Components are interesting intrinsically, but each UI worth its salt needs them to be laid out the way we want. The second section lets us browse the many layouts, which also are components, provided by Vaadin:

- Simple layouts such as vertical/horizontal layouts, grids, or even forms.
- More advanced ones, such as absolute, CSS, or custom layouts. These are much more powerful, but at the cost of a tighter coupling with the web browser terminal.

Finally, we examined split panels, which are specialized component containers.

We finished this chapter with the first Java classes of our Twaattin killer application!

As yet, components and layouts are next to useless because there is no interaction between the user and the application yet. The next chapter will address this, as we will dive into the event-listener model in Vaadin.

5

Event Listener Model

In this chapter, we will see how Vaadin components communicate with each other.

Components have to work together in order to achieve a common goal. Like ants, they cannot do so without a way to pass information through messages. In Vaadin and other software, this is done through events and listeners.

In the first section of this chapter, we will explain this whole event and listener thing. We will have a look at the famous observer pattern and the way it is used in Java EE applications.

Then we will get a grasp on how it is implemented in Vaadin and the different ways one can wire widgets together so one can be the subject and the other observers. A discussion will follow in order to determine which components category is more suitable to serve as observers.

Finally, we will go further into Twaattin and wire some event listener behaviors into it.

Event-driven model

Most of the time, web developers are blissfully ignorant of what is known as **event-driven software**. It is, however, the bread and butter of the client server application developers. We will have a detailed look at this model.

The observer pattern

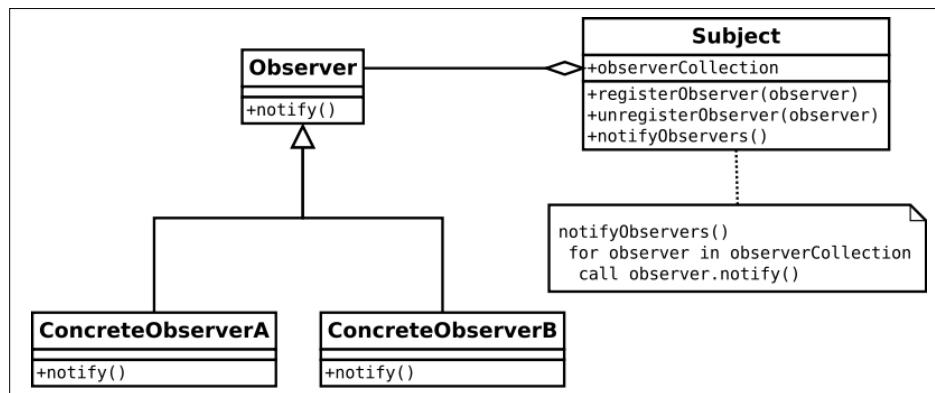
The event-driven model is based on a design pattern described in detail in *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides.

In this computer software book, the authors present answers to common software challenges. Each problem-solution pair is known as a design pattern and constitutes a design library one can draw upon when facing a particular quandary. The **observer** pattern is such a pattern. Here is the problem and its associated solution:

Suppose we have an object that is subject to changes throughout the application lifecycle. Now, this object has to tell other objects about these updates. For example, when a user clicks on a button (the main object) and until the server responds, the button has to be disabled, the menu has to be grayed, and a waiting cursor should be displayed.

We could of course handle the button click from the user, and having references to the other widgets call the adequate methods on them. Then again, the separation of concerns principle encourages us to decouple the button's code from other widgets.

That is where the Observer pattern comes into play. Using it, we will register widgets that are interested in being informed that a click occurred. At the time of the click, a single generic method (for example `notify()`) will be called in each of the observers. What the method really does is up to the implementation of each observer.



Enhancements to the pattern

The observer pattern is very general and has nothing to do with user interfaces. In order to manage the complexity of the latter, there are some necessary enhancements.

Event

An **event** is an action that is initiated outside the flow of the software:

- The code may or may not handle an event
- **Event-handling** may be either synchronous or asynchronous
- Those events can in turn fire other events

Event details

In the plain observer pattern, the `notify()` method has no parameters. In order to pass information from the subject to the observers, we can introduce one in the form of a detail object.

Detail attributes may include the timestamp of the event, the event's source, and other attributes depending on the event type.

Event types

Passing information through a parameter somewhat enhances the starting pattern, but lacks handling granularity. This means that an observer that is registered to two or more subjects is unable to distinguish between them when notified of an event.

Therefore, an area of improvement is the creation of different event types. The granularity of types is of course dependent on the considered system, no differentiation between types to a type per real event (click, value selection, key type, and so on).

Events in Java EE

Interestingly enough, events and the observer pattern are not widespread throughout Java EE. There are some notable exceptions, though:

- Java Messenger Service listeners are objects that connect to a JMS queue. As soon as a message is posted in the queue, the `onMessage()` method of the listener is called.
- Message Driven Beans are specialized Enterprise Java Beans that integrate a JMS listener in the EJB architecture, adding transactional and security features.
- In web application specifications, there are also some listener interfaces available since Version 2.3. As a reminder, they correspond to events regarding:
 - **Request:** It includes creation/destruction and attributes binding/unbinding in request scope

- **Session:** It includes creation/destruction, activation/passivation, and attributes binding/unbinding in session scope
- **Application:** It includes start/stop and attributes binding/unbinding in application scope
- Only recently did Java EE 6 introduce a whole event-listener model with JSR-299, also known as Context and Dependency Injection.

UI events

Vaadin provides its own event handling API. It is not specific to Java, but can also be used in PHP, ASP, or other web application technologies.

Client-server events

One type of event that has to be addressed is the sending of messages from the client tier to the server tier, which can be referred to as "HTTP events".

For example:

- When the user clicks on a submit button, the client asks an URL resource from the server, the server sends the whole page, and the browser displays it
- When the user needs to open a pop-up window, it is the same

In traditional web pages, you can navigate URLs by clicking on hyperlinks and submitting forms whether in straight HTML or through JavaScript.

Different interactions maybe represented by different URLs, or a single URL can interpret HTML parameters, so as to have different actions depending on those parameters and their respective values.

Client events

Likewise, client events – events that are limited to the client – are implemented in JavaScript. Examples of such events include the following:

- Changing values of a particular drop-down list depending on the selected value of another. When sex is **Male**, selectable title values should be set to **Mr**
- Clicking on a submit button disables all the buttons on a page so as to prevent multiple submissions
- On the contrary, checking a checkbox labeled as **Accept general terms and conditions** enables the **submit** button

Limitations of URL and custom JavaScript for events

In addition, there are several limitations to these particular client event implementations:

- First, they are as low-level as can be. This directly translates into a lack of abstractions and therefore, a lack of productivity. In modern computer software, you don't have to think about bits.
- Then, both the client and server code have to be kept synchronized. Changing one end can have side effects on the other one. Unit testing is thus impossible; one has to rely on integration tests that are more complex, more heavyweight, and more fragile.
- Finally, as was mentioned earlier in *Chapter 1, Vaadin and its Context*, JavaScript is somewhat browser-dependent and code that goes beyond "Hello world" has to take the differences between browsers, versions, and platforms into account. Not only does it vastly increase the code complexity and decrease readability, but also makes bugs more probable. All these factors have a direct impact on costs.

Event model in Vaadin

The event model in Vaadin is twofold:

1. Implement the Observer pattern.
2. Add an abstraction layer to the HTML/JavaScript event.

In Vaadin, there are two different ways to add event routing to our objects.



Note that Vaadin handles the sending of client-side browser events and firing its own events on the server side.



Standard event implementation

The first way for Vaadin to offer event routing capabilities is with an implementation based on typed event-listener pairs. Each event has a corresponding pair.

For example, in order to act on focus and blur events, Vaadin provides the following pairs:

- `BlurListener / BlurEvent`
- `FocusListener / FocusEvent`

[ This design is very similar to what is done in AWT. However, there is a subtle difference in the implementation: the pairs are specialized and are related to a single occurrence type. For example, in AWT there is a single event-listener for both focus and blur event, with the listener having to implement two methods even if it needs only one of the two, whereas in Vaadin there is one method for focus and one method for blur events, leading to a decrease in useless code.]

Event class hierarchy

At the root of Vaadin event class hierarchy lies a core Java library class, namely `java.util.EventObject`. As a reminder, event object is just a thin wrapper around the event source and it has access to the event source.

Event

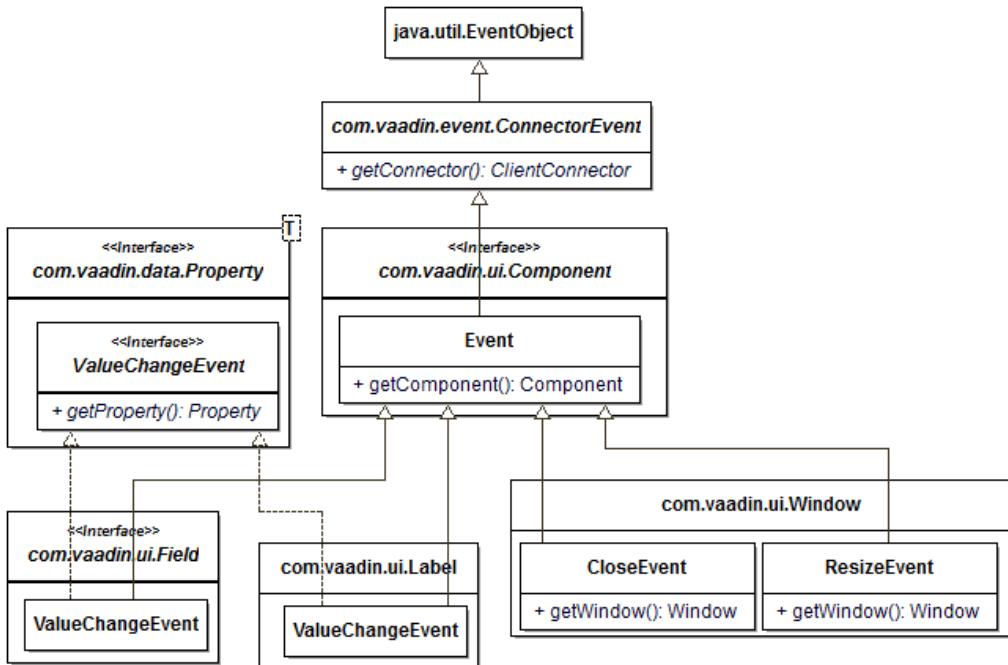
The first class that Vaadin introduces, and that it inherits from the former is `com.vaadin.event.ConnectorEvent`. Connectors will be covered in detail in *Chapter 9, Creating and Extending Components and Widgets*; for the time being, let's forget about this hierarchy level.

The next class in the hierarchy is `com.vaadin.ui.Component.Event`. The only contribution of `Event` is to narrow the return type of `Event.getSource()`. It provides the `getComponent()` method that returns a `Component` instance, which is just a way to ease the life of the developer.

Typed events

Subclasses of `Event` are of much more interest to us. In fact, each of the widgets seen in *Chapter 4, Components and Layouts*, holds at least one event inner class of its own.

[ The design of these event classes shows a good use of inner classes, as the event type is only pertaining to its outer class. For example, `CloseEvent` defined in `Window`, has no sense for labels.]



See how the subclasses of `Event` add a getter method with the narrowest return type possible in order to avoid casting the event source? Should you extend the hierarchy further, it is advised to do so, even if nothing enforces it. It is one of those nice comfortable features that make Vaadin so comfortable to code with.

Listener interfaces

There are many listeners available in Vaadin, and they all share some common features:

- There is one listener for each event type; events and listeners go in pairs
- There is no inheritance hierarchy between them
- They extend `Serializable` so as to be serialized during session serialization
- They are designed as inner interfaces of the relevant class; for example, `ClickListener` which waits for click events on buttons is defined in the `Button` class

- Also, they are defined as being `static`
- They have a semantic significance, meaning one can understand what it does without looking at the documentation
- They are single method interfaces and this method:
 - Has a name pertaining to the listener
 - Has a single parameter which is an event coupled to the listener

In order to continue with the `ClickListener` class, the only method's signature is `buttonClick(ClickEvent event)`.

Window

Now, remember our old friend the `Window` component from *Chapter 4, Components and Layouts*? At the time, nothing was said about it, but it contains the following two listeners (as well as two events, but let's focus on the former):

- `CloseListener`: This listener triggers when a window is closed.
- `ResizeListener`: This listener is called when the user resizes it.

The following code will display a notification when the window is closed:

```
public class MyUI extends UI implements Window.CloseListener {

    @Override
    public void init(VaadinRequest request) {

        // Some content has to be set or app will appear buggy
        setContent(new Label());
        Window window = new Window();

        window.addCloseListener(this);
        addWindow(window);
    }

    @Override
    public void windowClose(CloseEvent event) {

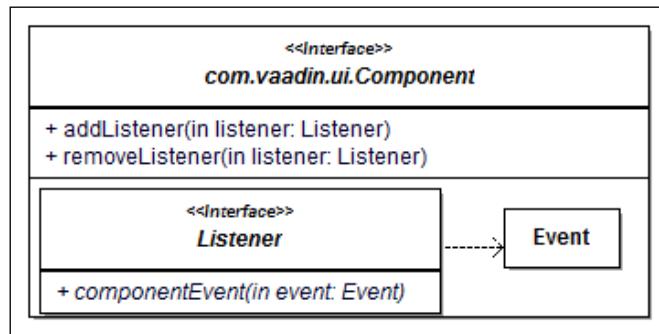
        Notification.show("Window has just been closed");
    }
}
```

Managing listeners

Each widget that may be an event source has two methods for each event-listener pair:

- A method to add a specific listener, `addXXXListener(XXXListener)`
- The symmetric method to remove it, `removeXXXListener(XXXListener)`

This leads us to the following schema, inner classes' structure notwithstanding:



This design is used throughout Vaadin, so it is better to keep it in mind, as it is very helpful when using a component-listener-event triplet we don't know. In addition, if we implement our own, it is better to copy this design in order to be consistent.



Previous versions of Vaadin used overloading to allow components to manage different listener types. The current version uses different methods. It is discouraged to use the overloaded version in favor of those new methods to avoid explicit type casting.



Method event source details

In order to enforce the **Single Responsibility Principle**, components rely on delegate listeners management to another abstraction. The framework introduces the `MethodEventSource` interface, which is an implementation of the Observer pattern mentioned earlier in this chapter.



For detailed information about the Single Responsibility Principle, visit the following URL:
http://wikipedia.org/wiki/Single_responsibility_principle



It has the following methods:

- `addListener(Class<?>, Object, Method)` will add a single listener where:
 - `Class<?>` is the event class the listener will respond to
 - `Object` is the listener object itself
 - `Method` is the method that will be called when the event type is received. Other event types won't trigger anything
- `removeListener(Class<?>, Object, Method)` where the parameters are the same as above; it will remove the previously added listener

Additionally, not using the third argument will remove all listeners for the event type on the target object.



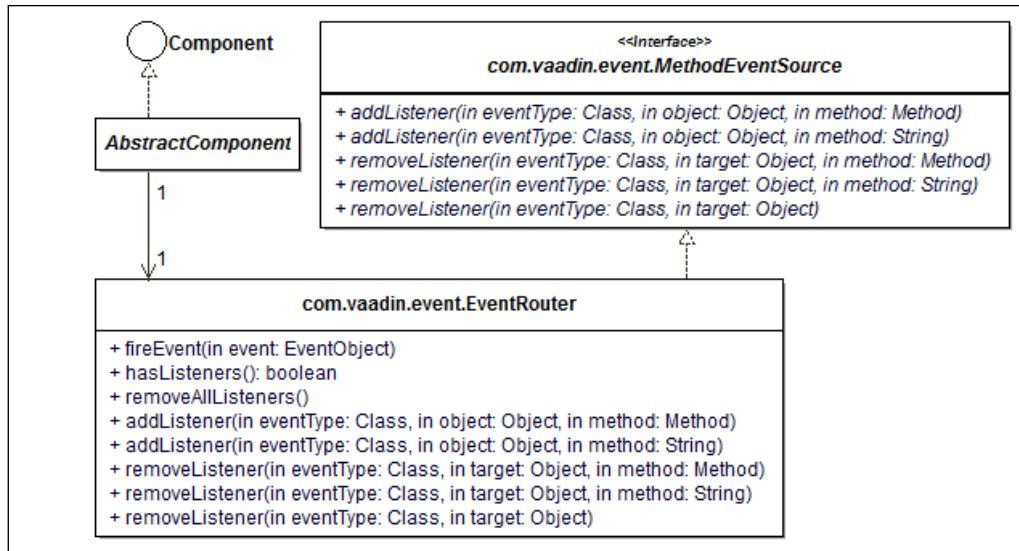
It is advised not to use `MethodEventSource` directly but to prefer the friendlier and typed listener management methods found in the component classes. However, it can be used to implement your own event handling for your application.

Abstract component and event router

Now, the contract for every abstract component is also meant to be a method event source. In order to achieve this, Vaadin introduces a concrete implementation of the method event source, the `EventRouter` class.

This class has the following two important advantages:

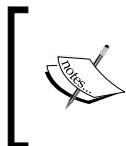
- Abstract components can delegate the event routing logic to the router, thus decoupling event, and display concerns
- Additionally, if there is a need to encapsulate an event router in our own components, then we choose not to inherit from `AbstractComponent`, but rather use our own implementation



Expanding our view

The respective event and listener classes described earlier are by no means extensive. As for the components, it makes no sense to plainly list them, it has no benefit, and is out of the scope of this book. It would only paraphrase the Javadocs anyway.

However, a concrete example will show us that there is no need for it, as Vaadin implementation is uniform throughout the class hierarchy. What we learned here can be extended to any out of the box component.



If you want to challenge my claim, feel free to do so. Take a component, verify that it is designed as it should be and you will be pleasantly surprised. That is what makes Vaadin such an enjoyable framework to work with.

Button

Buttons are such standard components that Vaadin, of course, provides it.

The most important event that can happen on a button is the click. From what we have learned, `Button` should provide an inner class `ClickEvent` or something similar. This event should have a single method, `getButton()` that returns a `Button`, on which the user can click.

Also, Button should encompass a static inner interface `ClickListener` that has a single method with the signature `buttonClick(ClickEvent)`.

Finally, there has to be an `addClickListener(ClickListener)` method on the `Button` class itself.

Looking at <http://vaadin.com/api/com/vaadin/ui/Button.ClickEvent.html>, we can challenge our theory. It appears completely in line with what we have just guessed before:

Class Button

```
java.lang.Object
└ com.vaadin.server.AbstractClientConnector
  └ com.vaadin.ui.AbstractComponent
    └ com.vaadin.ui.Button
```

Nested Class Summary

static class	Button.ClickEvent Click event.
static interface	Button.ClickListener Interface for listening for a Button.ClickEvent fired by a Component .
static class	Button.ClickShortcut A ShortcutListener specifically made to define a keyboard shortcut that is

Method Summary

void	addBlurListener(FieldEvents.BlurListener listener) Adds a BlurListener to the Component which gets fired when a Field loses focus.
void	addClickListener(Button.ClickListener listener) Adds the button click listener.
void	addFocusListener(FieldEvents.FocusListener listener) Adds a FocusListener to the Component which gets fired when a Field receives focus.

Events outside UI

In general, but also in Vaadin, events are not limited to interactions with user interface.

User change event

In fact, there is one event that is of particular interest to us because there is a good chance that it will be used throughout your future Vaadin applications. It is the user changed event; its structure follows the guidelines we saw earlier in this chapter.

In most applications, once a user logs in, the user's name, login, and so on, is displayed on the screen. Vaadin takes that into account and provides the following:

- A way to store an object representing the user in the application. Vaadin uses `VaadinSession`, an abstraction around `HttpSession` and `PortletSession` that behaves as a hash map. An advantage of `VaadinSession` over plain `HttpSession` is that hash map keys can be either `String` or `Class`. In the latter case, it means you don't have to rely on the `String` constant trick.

Note that Vaadin makes no assumption on the particular object type used. It can be a `java.security.Principal`, a Spring Security user details, a plain `String`, or your own custom implementation.



For more details on Java security, JAAS, and Principal, visit
<http://java.sun.com/developer/technicalArticles/Security/jaasv2/>

For more details on Spring Security and UserDetails, visit
<http://static.springsource.org/spring-security/site/docs/3.0.x/reference/springsecurity-single.html#d0e1588>

- An event model centered about changes made to the user stored in the application.
- A user change event holds both the previous user and the new user. This design lets us manage both login events (when old user is `null` and new user is not) and logout events (when old user is not `null` but new user is) and react accordingly.

Architectural considerations

Until recently, we described the event model in Vaadin but we did not map these listeners to any component (in the conceptual sense) or objects belonging to a particular layer in our architecture.

In fact, there are so few constraints enforced on listener classes (and even fewer when using the alternative implementation) that any object is a candidate for being a listener in its own right, isn't it?

Yet, just because something is possible, that does not mean that it is necessarily the right thing to do. Here, we stray from the pure Vaadin learning path to something that is more conceptual and thus more subject to debate. Some architectural choices are in order for listeners.

Anonymous inner classes as listeners

As Swing developers know, the vast majority of Swing examples found on the Web, and even some Oracle's own code, use a vast amount of anonymous inner classes.

This Java language feature lets developers implement an interface or override a class while creating a new instance, as shown in the following example:

```
window.addMouseListener(new ResizeListener() {  
  
    public void windowResized(ResizeEvent e) {  
  
        Notification.show("Window resized");  
    }  
});
```

Anonymous inner classes have both pros and cons:

- Inner classes have access to their wrapping scope final variables and their wrapping class attributes
- Overuse of them renders the code confused and decreased readability

Overall, beyond simple "Hello world" applications or prototypes, it is discouraged to use them in applications.

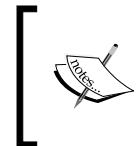
Components as listeners

"Components as listeners" is the simplest option possible. In fact, there is nothing that prevents us to make them so, as it is legal to do so.

However, this model has some limits that are worth mentioning:

- When going beyond simple "Hello world" applications, the sheer number of event listening methods just clutters the code and dreadfully impairs its readability.
- From a design point of view, this grouping denies the Single Responsibility Principle that forms the basis of good object-oriented software: component responsibilities should be limited to displaying and event **firing**, and not include event handling.
- As a corollary, this coupling also prevents us from reusing separately both components and behaviors, which is a shame since it should be a feature of OO design.

Nonetheless, this design lets us create quickly simple applications that just work. If used only within this restricted perimeter, or for prototypes, then it is the right choice as it follows the **Keep It Simple Stupid (KISS)** principle.



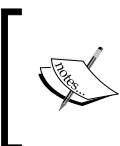
For more details on the KISS principle, visit the following URL:
http://en.wikipedia.org/wiki/KISS_principle



Presenters as listeners

In "standard" web applications, as well as Swing applications, the **Model View Controller (MVC)** design pattern is a common occurrence. In Java, this pattern is implemented like so:

- A Java Server Page represents the view. It has only presentation logic and the controller sets the data.
- The controller is a servlet that:
 - Requests data from the model
 - Feeds the data to the view
 - Redirects the control flow to the latter
- Finally, the model is implemented either as Session Enterprise Java Beans or as **Plain Old Java Objects (POJO)**, depending on each application's architecture.



In order to know more about Plain Old Java Objects, refer to the following URL:
http://en.wikipedia.org/wiki/Plain_Old_Java_Object



In rich client applications, the MVC model is often replaced with the Model-View-Presenter design pattern. There are some slight differences between them, but the most important is that the view is in charge for managing UI events, whereas it is the controller's responsibility in MVC.

In such architecture, presenters are listeners par excellence.

Services as listeners

Most applications are also designed as layered. The first is the presentation, the second the service (also known as business), and finally the data access. Such designs lets us change a layer and affect only the calling layer.

One could think about services being called through an event handling mechanism; for example, in order to populate a widget's data in response to a button click.

However, this does not work well in Vaadin, as event model implementation for listener methods have no return value, so we would need to pass a reference to be updated, which would couple our service layer.

Conclusion on architecture

In regards to architecture, there is no universal good or bad design. As in building construction, architecture in software is contextual and should always be thought of from this point of view. A whole book could be entirely devoted to this subject, and it won't even begin to cover all the cases, as architectural considerations are somewhat empirical and somewhat based on personal experiences.

However, the following points represent some of the factors that can influence architectural decisions:

- **Application size:** A small application with a short lifespan is more lenient towards cluttering. In this case, we could probably use anonymous inner classes without too many side effects. On the contrary, a big application will require more structuring in order to be manageable.
- **Expected lifespan:** For a constant ROI, the lesser the lifespan of an application, the lesser the cost. As such, an application that is a temporary solution should not be designed as a software jewel. For example, a Proof of Concept application is just that; it probably will be discarded after the proof has been made. Don't waste time on architecture!
- **Team experience:** In this regard, experience is not only quantitative, but also qualitative. While it is true that more experienced developers are naturally inclined to use more structured solutions, you should also consider how developers actually code. If your team already practices MVP daily, then maybe it would be a good thing not to change things too much.
- **QA:** Of course, QA is the biggest reason of all to adopt a specific architecture. If the norm is to use components as listeners, then don't hurt your head too much and do as you are told.

In previous sections of this book, there are some examples of possible architectures. There is nothing to stop you from designing other solutions. In fact, you are encouraged to do so if none fit your particular needs for an application.

Twaattin is back

We will put what we have seen in this chapter regarding events and listeners to good use: Twaattin awaits us!

We have left it in a state where login and timeline screens were displayed side-by-side, because we did not know how to manage user interaction. As now we can, let's amend our coding to conform to the following changes:

- Set the login screen as the first displayed screen
- When the user presses the login button, authenticate the user
 - Sets the login as a session attribute
 - Send a user changed event
 - The listener of such event displays the timeline screen

Project sources

Most sources display only differences with *Chapter 4, Components and Layouts*. Remember that the full source code is available on GitHub: <https://github.com/nfrankel/twaattin/tree/chapter4>.

The UI

```
package com.packtpub.learnvaadin.twaattin.ui;  
  
...  
  
public class TwaattinUI extends UI {  
  
    @Override  
    public void init(VaadinRequest request) {  
  
        setContent(new LoginScreen());  
    }  
}
```

The UI now just displays the login screen, since it will be replaced by the timeline screen if login succeeds.

The login screen

```
import com.packtpub.learnvaadin.twaattin.presenter.LoginBehavior;  
...  
  
public class LoginScreen extends FormLayout {  
  
    ...  
  
    public LoginScreen() {  
  
        ...  
  
        submitButton.addClickListener(new LoginBehavior(loginField,  
                passwordField));  
    }  
}
```

In the login screen, we just add a single line in order for a login behavior to act as a listener when the button is clicked.

The login behavior

```
package com.packtpub.learnvaadin.twaattin.presenter;  
  
import static com.vaadin.ui.Notification.Type.ERROR_MESSAGE;  
  
import java.security.Principal;  
  
import com.packtpub.learnvaadin.authentication.  
AuthenticationException;  
import com.packtpub.learnvaadin.authentication.  
SimpleUserPasswordAuthenticationStrategy;  
import com.packtpub.learnvaadin.authentication.  
UserPasswordAuthenticationStrategy;  
import com.packtpub.learnvaadin.twaattin.ui.TimelineScreen;  
import com.vaadin.server.VaadinSession;  
import com.vaadin.ui.Button.ClickEvent;  
import com.vaadin.ui.Button.ClickListener;  
import com.vaadin.ui.Notification;  
import com.vaadin.ui.PasswordField;  
import com.vaadin.ui.TextField;  
import com.vaadin.ui.UI;
```

```
public class LoginBehavior implements ClickListener {

    private static final long serialVersionUID = 1L;

    private final TextField loginField;
    private final PasswordField passwordField;

    public LoginBehavior(TextField loginField,
        PasswordField passwordField) {

        this.loginField = loginField;
        this.passwordField = passwordField;
    }

    @Override
    public void buttonClick(ClickEvent event) {

        try {

            String login = loginField.getValue();
            String password = passwordField.getValue();

            Principal user =
                new SimpleUserPasswordAuthenticationStrategy()
                    .authenticate(login, password);

            VaadinSession.getCurrent().setAttribute(
                Principal.class, user);

            UI.getCurrent().setContent(new TimelineScreen());

            Notification.show("You're authenticated into Twaattin");

        } catch (AuthenticationException e) {

            Notification.show(e.getMessage(), ERROR_MESSAGE);
        }
    }
}
```

When the user presses the **submit** button of the login form, the `buttonClick()` method of the presenter is called. If the authentication succeeds, it sets the user in the session; if not, it displays a failure message. The real authentication logic is delegated to specific package having nothing to do with Vaadin (the interested reader might want to read up on GitHub).

Additional features

However, now we can log in, two more features would be nice to have:

- Since we could log in to more than one user, the ability for the application to display the user login is necessary
- Additionally, being able to log out without closing the browser and deleting the cookie is a good thing

Some small changes are in order to implement these new features.

The timeline window

```
import static com.vaadin.server.Sizeable.Unit.PERCENTAGE;
import static com.vaadin.ui.Alignment.MIDDLE_RIGHT;

import java.security.Principal;

import com.packtpub.learnvaadin.twaattin.presenter.LogoutBehavior;
import com.vaadin.server.VaadinSession;

public class TimelineScreen extends VerticalLayout {

    public TimelineScreen() {
        ...

        Label label = new Label(VaadinSession.getCurrent()
            .getAttribute(Principal.class).getName());
        Button button = new Button("Logout");
        button.addClickListener(new LogoutBehavior());
        HorizontalLayout menuBar = new HorizontalLayout(label,
            button);
        addComponent(menuBar);
        ...
    }

    ...
}
```

We just need a label to display the user and a logout button that listens to click events, just as we have seen previously, and to add the logout behavior.

The logout behavior

```
package com.packtpub.learnvaadin.twaattin.presenter;
import java.security.Principal;
import com.packtpub.learnvaadin.twaattin.ui.LoginScreen;
import com.vaadin.server.VaadinSession;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.Button.ClickListener;
import com.vaadin.ui.Notification;
import com.vaadin.ui.UI;

public class LogoutBehavior implements ClickListener {

    private static final long serialVersionUID = 1L;

    @Override
    public void buttonClick(ClickEvent event) {

        VaadinSession.getCurrent().setAttribute(Principal.class,
            null);

        UI.getCurrent().setContent(new LoginScreen());

        Notification.show("You've been logged out");
    }
}
```

Logout behavior is symmetric to the login. It removes the user from the session (which is done by setting it to `null`), displays the login screen, and informs the user that he/she has been logged out.

Summary

In this chapter, we tackled the concept of events and listeners. Both form the basis for the Observer design pattern implementation. The latter can be summed up as, when objects want to be notified of certain occurrences in another object, they register as observers and when specific events happens, they trigger behavior depending on each object's implementation.

Then we learned that this pattern is used throughout that client software's user interfaces, but web developers are seldom aware of it. There are some event model implementations in Java EE, but they are unrelated to UI.

In Vaadin, however, we can keep our event-listener related knowledge (or acquire it) because it is fully observer-compliant.

Then, we discussed architectural considerations. The thing to remember here is that the architecture is based on each project's own features. There is no right or wrong answer, but this section hinted at some factors one has to take into account to determine which types are the best for listeners.

Finally, we went further into building Twaattin. Now we have the entire login-logout behavior fully implemented.

In the next chapter, we will connect objects and collections to UI components.

6

Containers and Related Components

In previous chapters, we learned about Vaadin's component-based approach, how it is implemented, and how those components send events to each other in a nicely decoupled way. This chapter is about the second important part of the framework: data binding.

This chapter is separated into two sections. In the first section, after a general view on data binding, we will have a thorough look at the three abstractions available in Vaadin: property, item, and container.

In the second section, we will discover two new components that are able to display containers, tables, and trees. As tables are present in so many applications, Vaadin provides a great deal of features that will demand some time to learn in detail.

Data binding

Data binding is the ability of an application to link the value displayed by a component with the underlying data. However, it is not a monolithic feature, but has the following properties that we can choose to/not to implement, depending on the technology:

- Accessing the data either in the read mode, that is, displaying the data through the component, or in the write mode, that is, updating the data through the component
- Storing changes made to the component's value in a buffer so they can be committed
- Binding the data to the component so that changes to the underlying data value change the value displayed to the user

Data binding properties

Properties of data binding are renderer and editor, buffering, and value-component binding.

Renderer and editor

The important aspect is that there is a transformation necessary between the data, which is an object in its own right, and its graphical representation.

For example, one can ask how a date should be shown to the end user. Most mature frameworks create the following two abstractions in order to standardize this process:

1. A renderer component that is able to display data items on the screen, thereby creating a string representation of the data.

In our example, the date can be processed by the renderer that will probably format it in some way, based on either a standard locale or even the user's locale.

2. An editor component that allows us to change objects; this can be done either from a string representation or from a specialized component.

Again, with our date example, the editor could let us change the date from its string representation and take the risk that days and month are not at the same place depending on which region of the globe you come from, or display a nice calendar.

Buffering

Buffering is a way for components to discard changes made to it concerning the underlying data. In this regard, the component also provides a way to commit the buffered value to the underlying data, or to reset the buffer from the latter.

Note that it is different from an HTML text field that holds a value, as the component could disappear from a view and still hold the buffered value as long as it is accessing the same object.

Data binding

Data binding comes in two flavors: the easiest is when updating the UI component's displayed value really changes the underlying data.

The other flavor is very addictive once tasted: imagine a value having the ability to send change events to components to which it is bound. For example, when changing the value of the person's first name variable in the code, the value is magically changed for the user in the GUI.

In order to achieve this, data is to be wrapped by a custom component that adds a whole event-listener model around it.

Data in Vaadin

The good news is that Vaadin brings renderers, editors, and buffering to the table. The bad news is that bidirectional data binding is not supported out of the box.

Entity abstraction

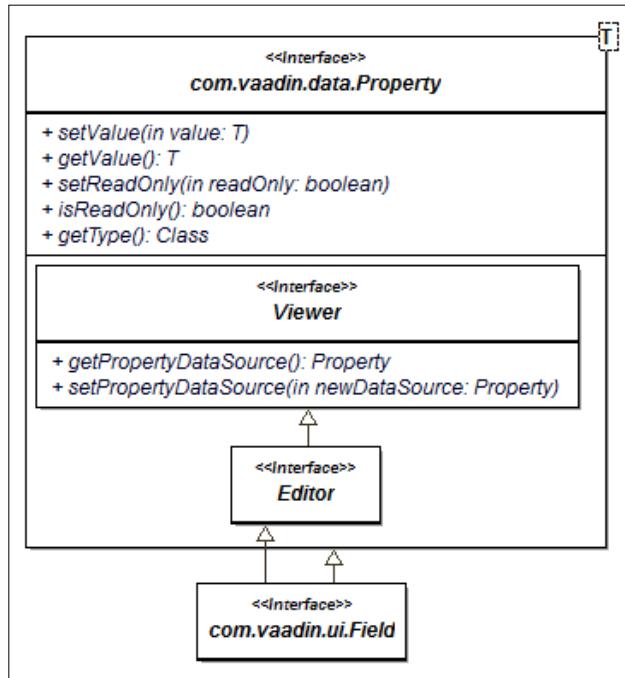
Regarding the entity abstraction problem, Vaadin provides a clean design with the following three interfaces corresponding to a different grouping level:

- At the property level, for example, the birthdate of a person
- At the item level, the item here being the entity, for example, a person
- At the container, representing a collection of related items, for example, a list of persons

Property

We have seen the `Property` interface in *Chapter 4, Components and Layouts*. As a reminder, it represents a single isolated value, with accessors available for value, a read-only indicator, and datatype (only getter available). In *Chapter 5, Event Listener Model*, we have seen that it also provides a change event listener.

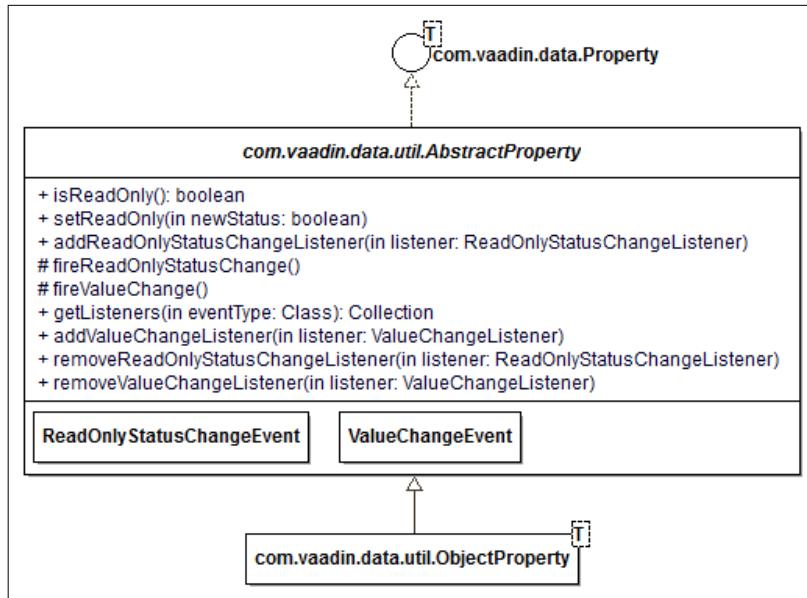
Now is a good time to learn that `Property` also provides two interfaces: `Viewer` and `Editor`, which `Field` extends. The following is a figure showing those concepts:



The following two facts are worth noticing:

1. First, viewer and editor have a slightly different meaning than shown:
 - A viewer represents a class that is able to use a property as a datasource.
 - An editor is only a marker interface. It means that the property can be changed through the editor. If not implemented, the property can still be set, but only in the code with the `setValue()` method. Likewise, even if implemented, we still cannot call the previous method if the property is in the read-only mode.
2. Second, for a property to be an editor, it has to be a viewer, which makes sense – ever tried to edit a value you couldn't see?

Finally, as we have seen before, `Field` is the single subinterface of `Property`, which is implemented by all components in the class hierarchy (with the notable exception of components that do not wrap around an editable value such as labels, menu bars, and layouts).



AbstractProperty

`AbstractProperty` is the common superclass of all `Property` implementations. It is a straight implementation of `Property`, `ReadOnlyStatusNotifier`, and `ValueChangeNotifier`.

It provides the following:

- A `readOnly` property
- Methods for adding both `ReadOnlyStatusChangeListener` and `ValueChangeListener`
- Methods for firing instances of `ReadOnlyStatusChangeEvent` and `ValueChangeEvent`

For a quick refresher on the whole event-listener implementation in Vaadin, please refer back to *Chapter 5, Event Listener Model*.

ObjectProperty

Moving on to the next class in the hierarchy: `ObjectProperty` that forms the basis of a wrapper around values connected to components. The following code snippet connects a date object to a label:

```
Property<Date> property = new ObjectProperty<Date>(new Date());  
  
Label label = new Label();  
  
label.setPropertyDataSource(property);
```

It will display the current date as follows:

```
Thu Jan 10 23:23:56 CET 2013
```

Note that Vaadin core components automatically listen to the change events of their respective datasources: changes to the underlying property will be propagated to the label component.

Property formatter

Astute readers may have some questions regarding the previous example: we have seen previously that a viewer component should bridge the object world and their string representations. The date displayed earlier is just a straightforward result of `toString()`.

In order to convert in our own way, we should just use the `Converter<P, M>` interface seen in the *Conversion* section of *Chapter 4, Components and Layouts*, and implement the desired `getFormat(Locale)` method.

Let us change our previous example somewhat:

```
Property<Date> property = new ObjectProperty<Date>(new Date());  
  
Label label = new Label();  
  
label.setPropertyDataSource(property);  
  
StringToDateConverter converter = new StringToDateConverter() {  
  
    @Override  
    protected DateFormat getFormat(Locale locale) {
```

```

        return new SimpleDateFormat("dd/MM/yy");
    }
};

label.setConverter(converter);

```

The same can be done with an editable component, with the expected result. We just have to replace the label with a text field and it yields the correct result as follows:



20-01-13

The previous example is just that—an example. If you need a text field that holds the date value, just pick a Vaadin component made just for this case: `com.vaadin.ui.DateField`, or even better, `com.vaadin.ui.PopupDateField` that comes bundled with a nice calendar editor; you will love it.

Just remember that common use cases have a high probability of having been dealt with by Vaadin developers.

Handling changes

As seen in *Chapter 4, Components and Layouts*, all Vaadin components are buffered regarding the data they hold and as such, have two important operations: `commit()` and `discard()`. Now that we have seen properties and events, it is the right time to put it all together.

Consider the following use case; we have a text field. Once updated, its changes can either be saved or cancelled. The following code does exactly that:

```

import static com.vaadin.ui.Notification.Type.TRAY_NOTIFICATION;

import com.vaadin.data.Property;
import com.vaadin.data.util.ObjectProperty;
import com.vaadin.event.FieldEvents.TextChangeEvent;
import com.vaadin.event.FieldEvents.TextChangeListener;
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.Button;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.Button.ClickListener;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Notification;

```

```
import com.vaadin.ui.TextField;
import com.vaadin.ui.UI;

public class CommitDiscardUI extends UI {

    @Override
    protected void init(VaadinRequest request) {

        Property<String> prop = new ObjectProperty<String>("ABC");

        final TextField tf = new TextField(prop);

        tf.setBuffered(true);

        tf.addTextChangeListener(new TextChangeListener() {

            @Override
            public void textChange(TextChangeEvent event) {

                Notification.show("Text change (event) : "
                    + event.getText(), TRAY_NOTIFICATION);
            }
        });

        Button commitButton = new Button("Save");

        commitButton.addClickListener(new ClickListener() {

            @Override
            public void buttonClick(ClickEvent event) {

                Notification.show("Before commit (property) : "
                    + tf.getPropertyDataSource().getValue(),
                    TRAY_NOTIFICATION);

                tf.commit();

                Notification.show("After commit (property) : "
                    + tf.getPropertyDataSource().getValue(),
                    TRAY_NOTIFICATION);
            }
        });
    }
}
```

```

        Button discardButton = new Button("Cancel");

        discardButton.addActionListener(new ClickListener() {

            @Override
            public void buttonClick(ClickEvent event) {

                Notification.show("Before discard (property) : "
                    + tf.getPropertyDataSource().getValue(),
                    TRAY_NOTIFICATION);

                tf.discard();

                Notification.show("After discard (property) : "
                    + tf.getPropertyDataSource().getValue(),
                    TRAY_NOTIFICATION);
            }
        });

        HorizontalLayout layout = new HorizontalLayout(
            tf, commitButton, discardButton);

        setContent(layout);
    }
}

```

The graphical result is as follows:



From this point on, try the following sequential interactions:

1. Change the text.
2. Click on **Cancel**. Prior to the click, `tf.getValue()` returns the previously entered value that is stored in the buffer. In essence, `tf.getValue() != datasource.getValue()`. After the click, the buffer is reset to "ABC", which is the value of the datasource.
3. Change the value again and click on **Save**. As the field is in a read-through mode, prior to the click, `tf.getValue()` delegates to `datasource.getValue()` and returns "ABC"; after the click, `tf.setValue()` calls `datasource.setValue()`.

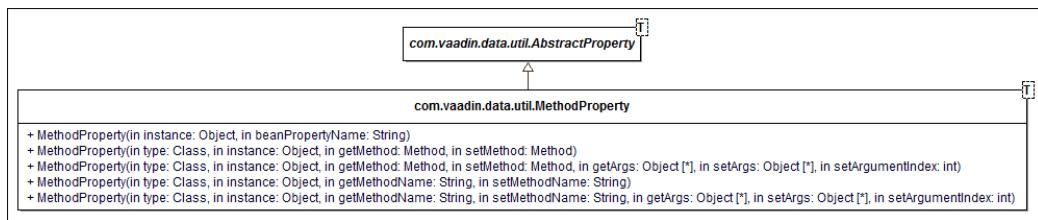
Item

The preceding section taught us about formatting, parsing, and buffering individual fields. We did not address the unrelated fields syndrome described earlier.

Method property

When faced with the challenge of displaying a structured object, the first solution to our quandary would be to get a reference to each single attribute, wrap it inside `ObjectProperty`, and then connect it to the right field. That would be too cumbersome and against Vaadin's principles.

There is a shortcut to this whole thing in the form of the `MethodProperty` class. `ObjectProperty` envelops a single object, whereas `MethodProperty` encapsulates a pair of accessor methods.



For example, let's consider our previous `Person` class, which has properties for first name, last name, and birthdate and a read-only ID. In order to bind a person to a datasource, we would have to do the following:

```
MethodProperty<Person> firstName = new MethodProperty<Person>
    (person, "firstName");
```

It is understood that it would have to be done for each single field we want to display. It is better than the previous solution, but still not satisfactory.

The right level of abstraction

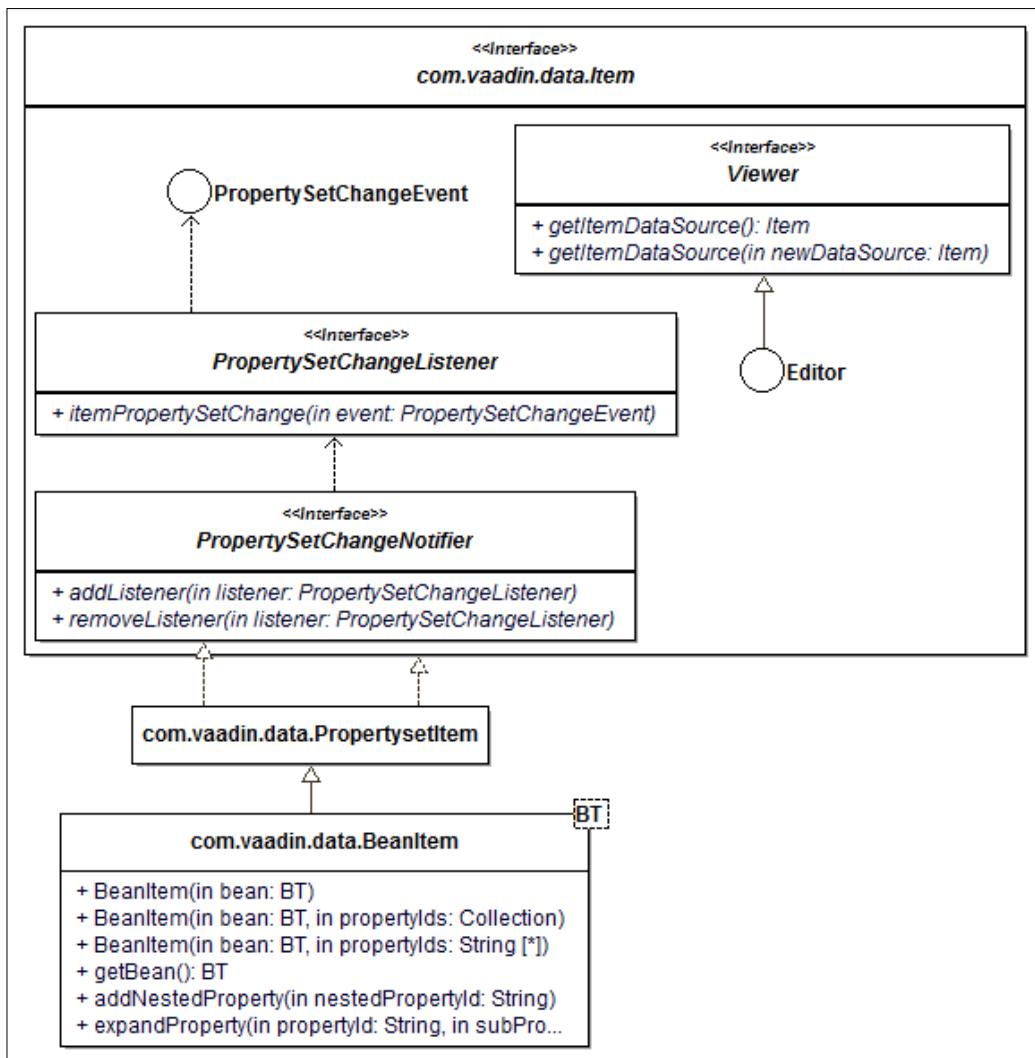
In this case, the right thing to do is to use one of Vaadin's most important interfaces: `Item`.

Just as `ObjectProperty` wraps a simple object and `MethodProperty` wraps a structured object's methods pair, `Item` wraps a bag of properties. Even better, wrapping `BeanItem`, which is the standard implementation of `Item`, around such an object will automatically enclose each of its property inside `MethodProperty`!

JavaBean standard



BeanItem relies on the JavaBean standards, meaning Vaadin will only expose JavaBean properties, methods that start with either get (or is for Boolean) for read accessors and set for write accessors.





Notice that the previous class diagram represents every aspect of Vaadin seen previously in the same class diagram: event model and viewer/editor pair.

Also, note that in order to see the big picture, interfaces and serializable classes are not represented as such. Just keep this in mind for future development.

This wrapping may be done in different ways, but the reference to the enclosed object is immutable, that is, it cannot be changed after the item instantiation. The constructor is available in the following two flavors:

- When only the wrapped bean is specified, automatic wrapping occurs based on reflection. Vaadin will try to find the appropriate bean descriptor (see <http://download.oracle.com/javase/6/docs/api/java/beans/BeanDescriptor.html> for more information), and if not found will default to listing methods and finding those, which begin with get/set.
- In addition to the wrapped bean, properties to be wrapped can also be specified as a collection of strings or as a string array.



Reflection and valid properties

In all three cases, Vaadin will wrap a property only if a valid getter/setter pair can be found (actually, a property with no setter is equally valid from a Vaadin point of view, but enforces the property to be read-only). In the first case, this is easy as the framework will provide only valid ones. For the last two, extra-care has to be taken in order to synchronize the string parameter values with real properties.

Remember, both reflection and string implies painful refactoring.

Once wrapped inside `BeanItem`, we can query for the right `Property` and set each as a field datasource very easily.

The following code uses a `Person` instance and wraps it inside `BeanItem`, ready to be used in our application:

```
import java.util.Date;

public class Person {

    private final Long id;
    private String firstName;
    private String lastName;
    private Date birthdate;
```

```
public Person(Long id) {
    this.id = id;
}

public Long getId() {
    return id;
}

public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}

public Date getBirthdate() {
    return birthdate;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public void setBirthdate(Date birthdate) {
    this.birthdate = birthdate;
}

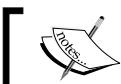
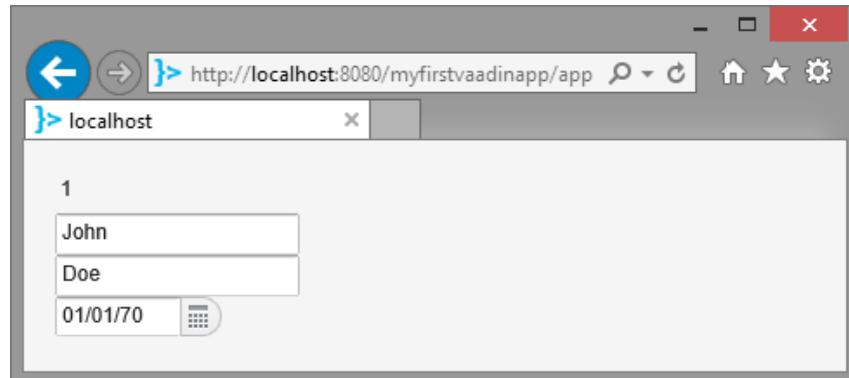
import java.util.Date;

import com.vaadin.data.util.BeanItem;
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.FormLayout;
import com.vaadin.ui.Label;
import com.vaadin.ui.TextField;
import com.vaadin.ui.UI;

public class PersonUI extends UI {
```

```
@Override  
protected void init(VaadinRequest request) {  
  
    Person person = new Person(1L);  
  
    person.setFirstName("John");  
    person.setLastName("Doe");  
    person.setBirthdate(new Date(0));  
  
    BeanItem<Person> item = new BeanItem<Person>(person);  
  
    TextField id = new TextField (item.getItemProperty("id"));  
    TextField firstName =  
        new TextField(item.getItemProperty("firstName"));  
    TextField lastName =  
        new TextField(item.getItemProperty("lastName"));  
    DateField birthdate =  
        new DateField(item.getItemProperty("birthdate"));  
  
    FormLayout layout = new FormLayout(id, firstName, lastName,  
        birthdate);  
  
    layout.setMargin(true);  
  
    setContent(layout);  
}  
}
```

The following screenshot shows the final result:



Notice that as id has no setter; Vaadin is smart enough to set the TextField object we asked for as read-only.



Field group

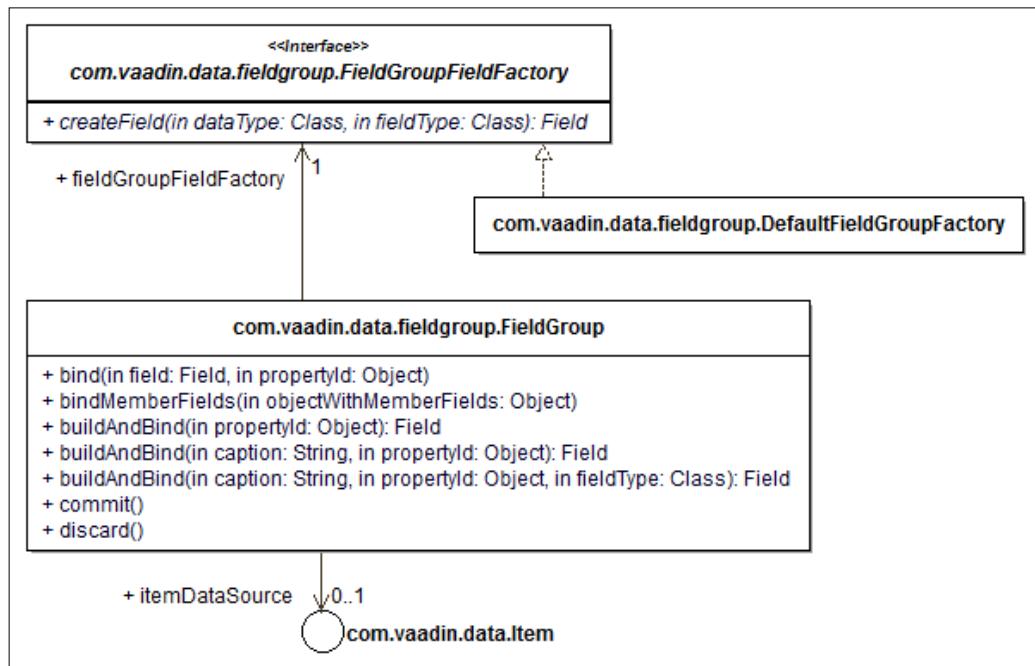
Our screen is nice and good, but probably misses a **Commit** and a **Cancel** button, as shown for the single date field previously.

If we try to do it, we are going to run into some hardship. Which one? Well, in our previous date example, committing or discarding involved a single field. Now, with three fields, are we going to commit/discard each field individually? That wouldn't be very productive; surely there must be a way to make it easier.

In fact, there is one, in the form of the `FieldGroup` class. Field groups have a nice feature: they can have `Item` as their datasource, but they also have both `commit()` and `discard()` methods that will commit/discard all wrapped item properties globally.

The main responsibilities of `FieldGroup` include the following:

- Creating a field from a property, the field type adapted to the property type.
- Binding an item's property to an already existing field. Alternatively, the class also provides methods to create fields and binds properties to it in a single method call.
- Committing and discarding: in this case, it calls the relevant method on each field belonging to the group.





Binding limitation

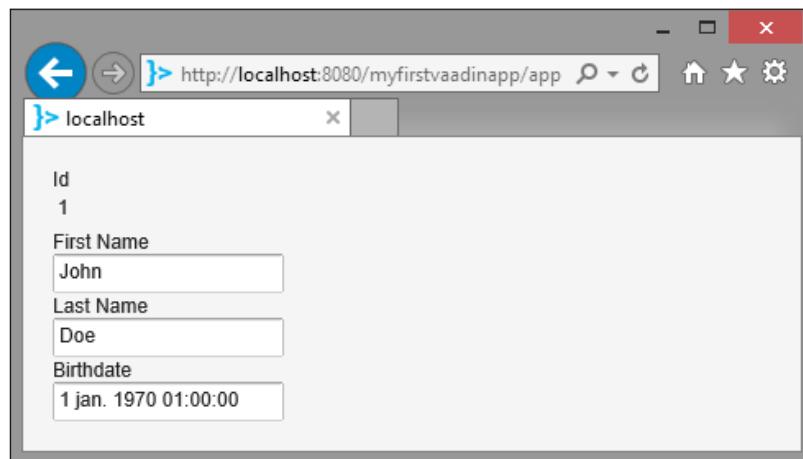
Once bound, a property cannot be bound furthermore. This means that at any one point, only a single field will interface through that particular property.

Now, we could replace the body of the previous snippet with the following:

```
FieldGroup group = new FieldGroup(item);

Field<?> id = group.buildAndBind("id");
Field<?> firstName = group.buildAndBind("firstName");
Field<?> lastName = group.buildAndBind("lastName");
Field<?> birthdate = group.buildAndBind("birthdate");
```

We get the following result, without any further configuration:



This approach has both pros and cons. Among the advantages, we can list the following:

- It wraps the item datasource
- Field captions are given for free
- When there is no setter, Vaadin is smart enough to only display a label

However, for the birthdate, only a text field is built, even though we would have needed a date field. The good news is that these defaults are fully configurable!

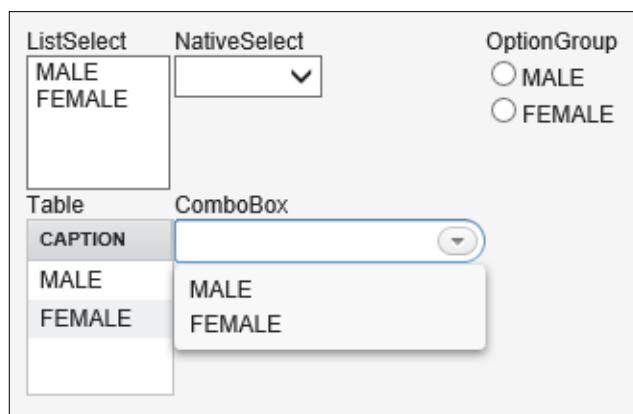
Configuring field types

By default, Vaadin creates checkboxes for Boolean types.

For enum types, it is mandatory to choose the exact field type between all available types, meaning *we have to use* the `buildAndBind()` method that accepts the class parameter or Vaadin will throw a `BindException` exception.

```
group.buildAndBind(null, "gender", ListSelect.class);
```

Available types are represented in the following screenshot. Please play around before going further:



Yet, it is always possible to override this behavior with little effort through the use of `FieldGroupFieldFactory`. The factory is the delegate responsible for creating fields and has a single `<T extends Field> T createField(Class<?>, Class<T>)` method. By default, `FieldGroup` uses an instance of `DefaultFieldGroupFieldFactory` that has the previously described behavior.

In order to get a `DateField` for `Date` properties, we only have to extend the default factory. Let us do this while favoring composition over inheritance:

```
import java.util.Date;
import com.vaadin.data.fieldgroup.DefaultFieldGroupFieldFactory;
import com.vaadin.data.fieldgroup.FieldGroupFieldFactory;
import com.vaadin.ui.DateField;
import com.vaadin.ui.Field;

public class AdvancedDateFieldGroupFieldFactory implements
    FieldGroupFieldFactory {
```

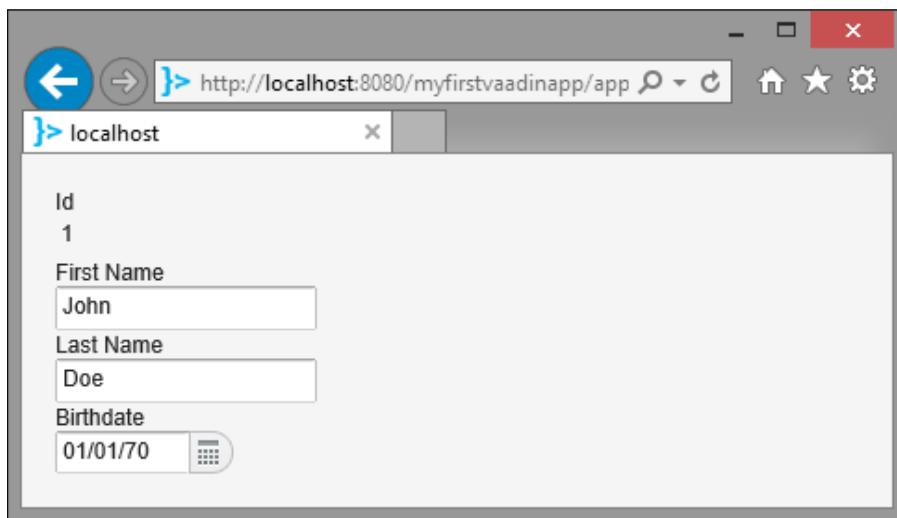
Containers and Related Components

```
private FieldGroupFieldFactory delegate =  
    new DefaultFieldGroupFieldFactory();  
  
    @Override  
    public <T extends Field> T createField(Class<?> dataType,  
        Class<T> fieldType) {  
  
        if (dataType.isAssignableFrom(Date.class)) {  
  
            return (T) new DateField();  
        }  
  
        return delegate.createField(dataType, fieldType);  
    }  
}
```

Now, we just need to set this factory to the previous field group to get the desired result.

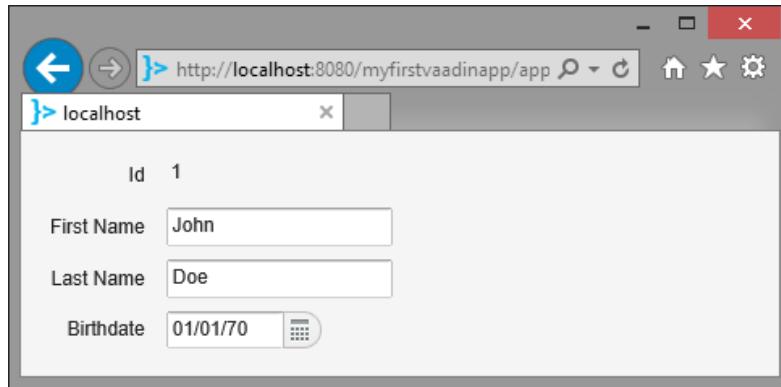
```
group.setFieldFactory(new AdvancedDateFieldGroupFieldFactory());
```

The output is as follows:



Replacing the vertical layout to a form layout even gives a better visual effect, aligning captions and fields on the same horizontal baseline.

```
FormLayout layout = new FormLayout(id, firstName, lastName,  
birthdate);
```



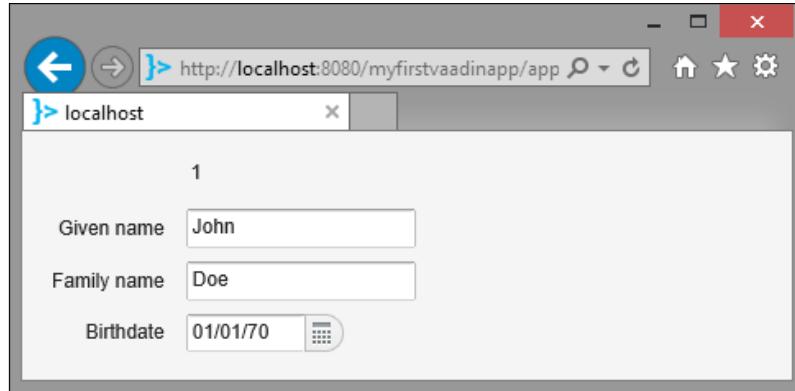
Changing captions

Field group default captions use the property name and convert it to uppercased spaced text.

If the caption is not desired, `FieldGroup` has a field `buildAndBind` overloaded method accepting a caption parameter. The following code removes the caption of the ID and sets alternate captions for `firstName` and `lastName`:

```
Field<?> id = group.buildAndBind(null, "id");  
Field<?> firstName = group.buildAndBind("Given name",  
                                         "firstName");  
Field<?> lastName = group.buildAndBind("Family name", "lastName");
```

Updating our code to reflect these changes gives us the following appearance:



Group commit/discard

In order to highlight commit/discard features, we just need to have the **Save** and **Discard** buttons, as in discussed in the *Handling changes* section. Let us implement this feature in a nicer way than before:

```
abstract class AbstractCommitDiscardClickListener
    implements ClickListener {

    private final String operation;

    public AbstractCommitDiscardClickListener(String operation) {
        this.operation = operation;
    }

    @Override
    public void buttonClick(ClickEvent event) {
        Notification.show("Before " + operation + ": "
            + group.getItemDataSource().toString(),
            TRAY_NOTIFICATION);

        execute();

        Notification.show("After " + operation + ": "
            + group.getItemDataSource().toString(),
            TRAY_NOTIFICATION);
    }
}
```

```

        protected abstract void execute();
    }

    Button commitButton = new Button("Commit");

    commitButton.addActionListener(
        new AbstractCommitDiscardClickListener("commit") {

            protected void execute() {

                try {

                    group.commit();

                } catch (CommitException e) {

                    throw new RuntimeException(e);
                }
            }
        );
    );

    Button discardButton = new Button("Discard");

    discardButton.addActionListener(
        new AbstractCommitDiscardClickListener("discard") {

            protected void execute() {

                group.discard();
            }
        );
    );

    FormLayout layout = new FormLayout(id, firstName, lastName,
        birthdate, new HorizontalLayout(commitButton, discardButton));

```



Notice that the `commit()` method throws a checked `CommitException` that we have to manage.

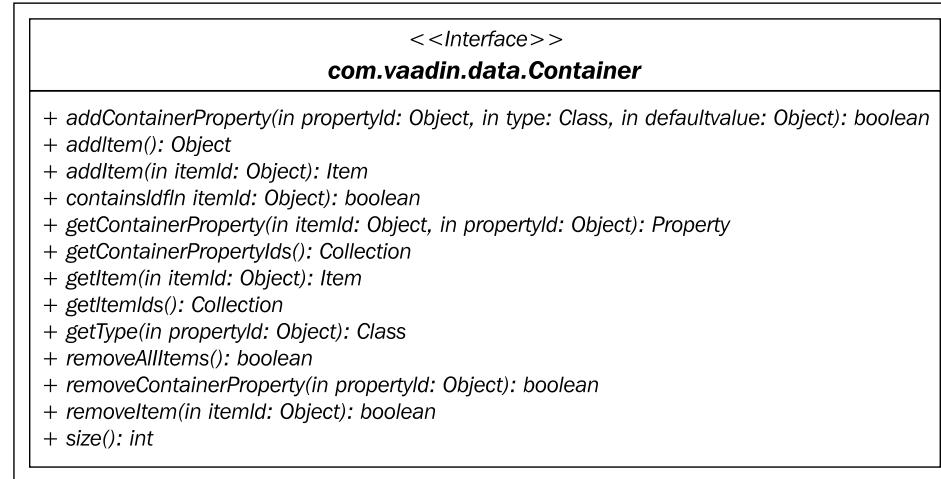
With this updated code, we can check that the changing of values do not happen until the field group is committed: it buffers the whole item it wraps.

This concludes the section about the use of field groups.

Container

In the previous section, we learned how to display a single object to a structured object. The next step is to learn how to display a list of structured objects, and that is the realm of Vaadin's Container interface.

Containers bring a completely new dimension to data binding.



The best way to picture a container is to think either of a 2D matrix, where lines are items and columns are properties.

However, there are some constraints on items put in a container:

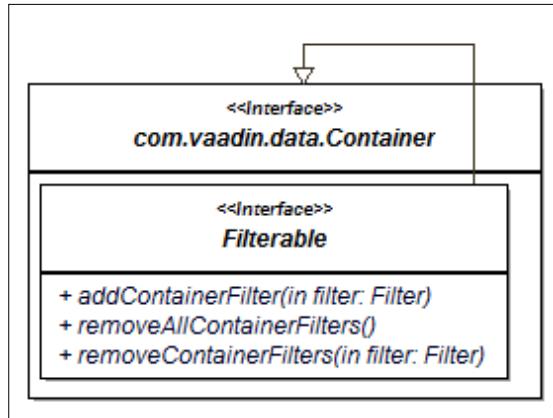
- All items in a container must have the same properties, meaning:
 - Properties must have the same ID
 - Properties must have the same data type
- Each item must be identified by a unique non-null identifier. Container enforces no particular condition on this ID, though children classes can. In essence, the ID is a key to access the corresponding item.

Filtering and sorting

Containers may also have additional capabilities.

Filterable

Filterable containers may display only some of its contained items, based on declared filters.



Filters can be either added or removed, and they are **additive**, meaning an item must meet all filter criteria to be displayed by the container.

Filter

Filters are based on the `Filter` interface that has the following two simple methods:

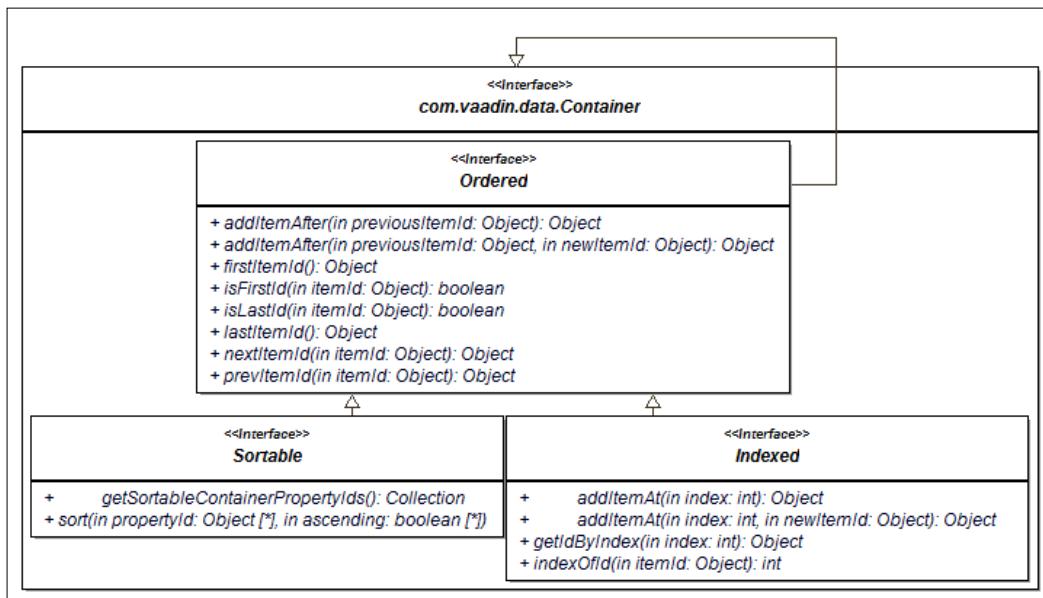
- `appliesToProperty(Object)` checks whether this filter applies to a specific property and returns a Boolean value accordingly. It's used as a first step in order to avoid possible algorithm overheads by the second method.
- `passesFilter(Object, Item)` applies the real filter to the object's ID and the object and also returns a Boolean value stating whether this object passes the filter and should be displayed.

Ordered

An ordered container lets us do the following:

- Insert an item after an already present item
- Get the first/last present item
- Get the next/previous item, given a present item ID

And that is it, it stops there. However, it has two interesting child interfaces.



Containers that also belong to the `Indexed` interface let us add items based on an index, as well as get the index of an object from its ID and vice versa. These features are seldom used; of much more interest is the `Sortable` interface.

Like its name implies, it allows us to sort the items found in the container.

The `sort()` method accepts the following two parameters:

- An array of property IDs. The sort is executed on the first property. If there's equality, sort continues with the second property, and so on.
- An array of Boolean values that refer to the sort order; `true` meaning ascending, `false` meaning descending.

Note that both arrays must have the same length. Moreover, we have to explicitly tell which properties are sortable with the `getSortableContainerPropertyIds()` method.

In order to be clearer with ordering, we will need some data. We will use the `Person` class defined earlier:

ID	First name	Last name	Birth date
1	John	DOE	01/01/1970
2	Jane	doe	01/01/1970
3	jules	winnfield	12/21/1948
4	vincent	Vega	02/17/1954

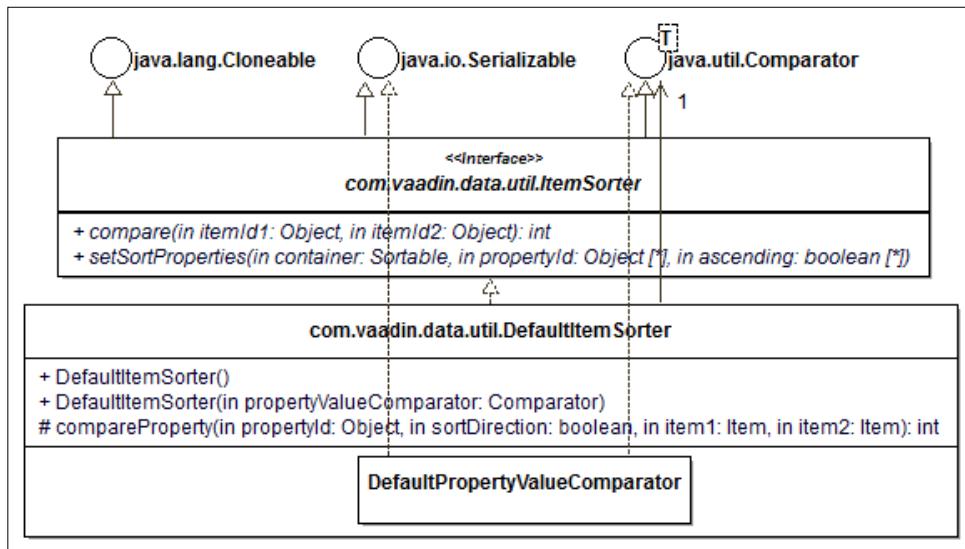
If we try to sort combinations on the sample data, the following are the results:

Property	Ascending	Lines order
<code>firstName</code>	<code>true</code>	2, 1, 3, 4 ("Jane", "John", "jules", "vincent")
<code>firstName</code>	<code>false</code>	4, 3, 1, 2 ("vincent", "jules", "John", "Jane")
<code>lastName</code>	<code>true</code>	1, 4, 2, 3 ("DOE", "Vega", "doe", "winnfield")
<code>birthdate</code>	<code>true</code>	3, 4, 1, 2

The first and the second sort seem to display the expected result. However, the third sort is case-sensitive and the fourth is based on the underlying `Date` value, which is desirable.

Item sorter

A concrete sortable container class (which we will see later in this chapter, bear with me for the moment) delegates sorting to an item sorter.



`DefaultItemSorter` is, guess what, the item sorter that is used if no other is set. In turn, it uses `DefaultPropertyValueComparator` in order to compare each property. Note that the latter implementation compares properties using `Comparable`.



If properties used for the set are not `Comparable`, Vaadin will throw a `ClassCastException`.



Now we understand the previous behavior. Both `Date` and `String` are `Comparable` and are sorted using the `compareTo()` method. In the case of dates, it does its job; in the case of strings, the method is case-sensitive.

In order to be case-insensitive, we would have to use `compareToIgnoreCase()`. Let us implement such a comparator and use it for a case-insensitive search.

As an example, we will create a property value comparator that will sort persons by names, either first or last, with no regards to case:

```
import java.util.Comparator;  
  
import com.vaadin.data.util.DefaultItemSorter.  
DefaultPropertyValueComparator;
```

```
public class CaseInsensitivePropertyComparator implements  
Comparator<String> {  
  
    @Override  
    public int compare(String prop1, String prop2) {  
  
        return string1.compareToIgnoreCase(string2);  
    }  
}
```

This new comparator does the same as the default one (in fact, it delegates to it if compared properties are not strings), but compares strings regardless of the case.

Then it is just a matter of passing the comparator as a constructor argument to the sorter, and then the sorter to the container implementation:

```
ItemSorter sorter = new DefaultItemSorter(  
    new CaseInsensitivePropertyComparator());  
  
container.setItemSorter(sorter);
```

Now, the third sort becomes 1, 2, 4, and 3 ("DOE", "doe", "Vega", "winnfield") which is the expected result.

Concrete indexed containers

Though container properties (`filterable`, `sortable`, and `ordered`) are designed so as to be independent, the vital `AbstractBeanContainer` implements all three. This suits our needs just fine nonetheless.

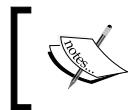
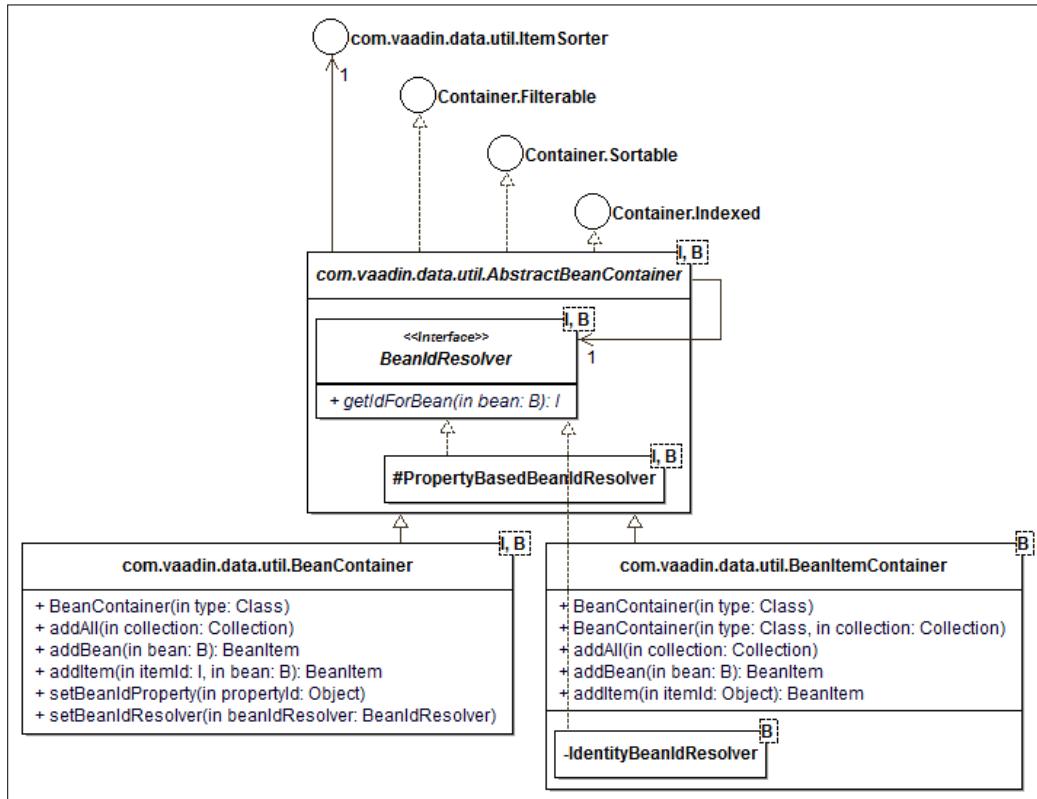
`AbstractBeanContainer` also introduces the following two important concepts:

- **Item sorter:** The one that we just talked about previously.
- **Bean ID resolver:** When we described container previously, we saw that an ID was just a key to a glorified hash map. Now, there must be some way to get the key: either pass it when adding an item or provide a way to compute the key from the bean. The latter is the responsibility of the bean ID resolver.

From there, the framework provides two simple implementations, which both use introspection on items to define what the container properties will be:

- `BeanItemContainer`, which uses the bean itself as the identifier. In order to do this, it redefines a very simple bean ID resolver named `IdentityBeanResolver`. Undercover, it uses `hashCode()` and `equals()`, just like standard `HashMap`.

- BeanContainer, which either enforces:
 - Passing an identifier along with the item to be added with the addItem() method
 - Using a bean ID resolver that computes an ID when adding a bean with the addBean() method



In order not to add even more complexity to the diagram, which is already dense enough, methods coming from `Ordered` and `Indexed` are not shown. Just remember that they exist.

As an example, let's create a bean container for our `Person` objects. This container will use the `Person` object's `id` as the key to the person itself.

The first step consists of creating the bean ID resolver, which is simple enough:

```
public class PersonIdResolver implements BeanIdResolver<Long, Person>
{
    @Override
    public Long getIdForBean(Person person) {
        return person.getId();
    }
}
```

Then, we can use our bean container as expected:

```
BeanContainer<Long, Person> container =
    new BeanContainer<Long, Person>(Person.class);

container.setBeanIdResolver(new PersonIdResolver());

container.addAll(...);

Person person = container.getItem(1L).getBean();
```

This example is trivial; however, how many times will we need an ID resolver that is not based on an identifier present in the bean? In order to prevent creating a bean ID resolver each time, Vaadin also provides a bean ID resolver based on a property. The previous code may be simplified with the following:

```
BeanContainer<Long, Person> container =
    new BeanContainer<Long, Person>(Person.class);

container.setBeanIdProperty("id");

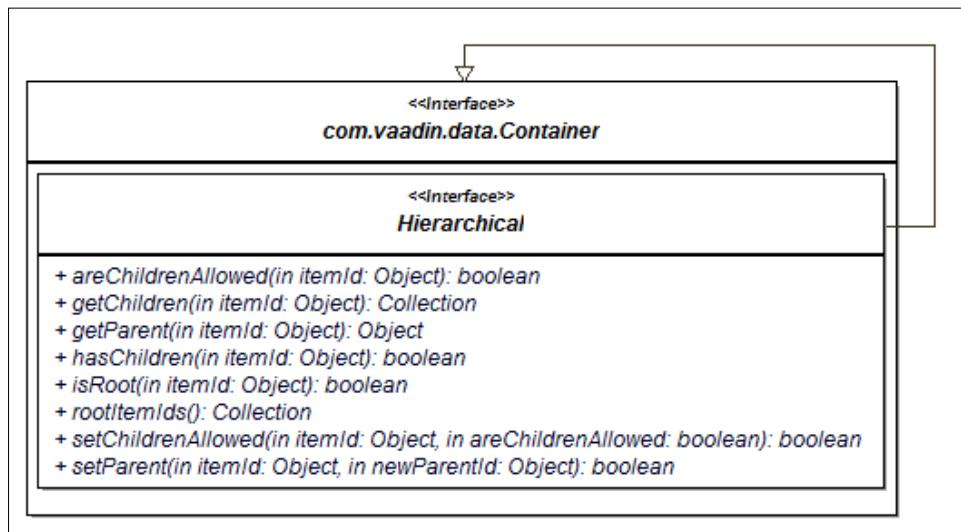
container.addAll(...);
```

We can now forget about our custom bean ID resolver.

Hierarchical

Simple tabular data management is addressed by the previous features and implementations of `AbstractBeanContainer`; however hierarchized data management is not.

Vaadin, however, provides another abstraction to manage it with the `Hierarchical` interface.



It provides some ways to organize data in a tree-like manner:

- Get/set leaf status of a node
- Get root status of a node
- Query for root nodes, note that multiple roots are possible
- Get the parent/children of a node
- Set a new parent for a node, thereby moving it around

Containers and the GUI

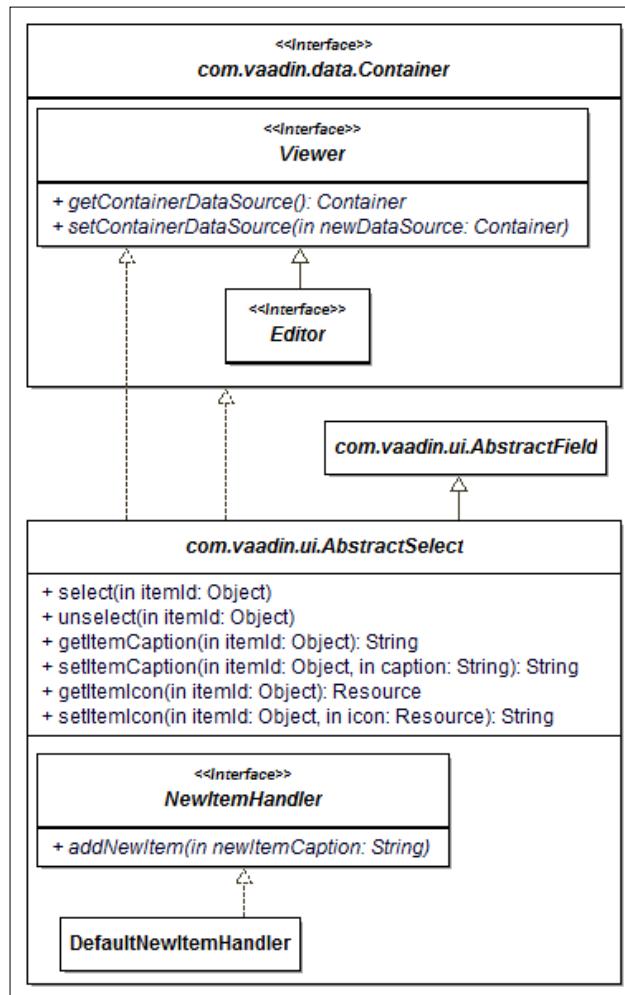
When we talked about `Property` earlier, it was a no-brainer to set it as a text field datasource in order to display it.

In the following section, we will be using `AbstractSelect` as an example.

Container datasource

In order to use Container as a component data source, there are still some details about it we have to understand:

1. First, Container mimics the structure of Property insofar as it encloses both a Viewer and an Editor interface.
 2. Second, there is a parallel between AbstractField and AbstractSelect, the parent classes for all components able to display a Container.



In addition to the previous methods, `AbstractSelect` also has some important properties accessible with getter/setter pairs. These are summarized in the following table:

Property	Type	Default value
<code>itemCaptionMode</code>	<code>ItemCaptionMode</code>	<code>EXPLICIT_DEFAULTS_ID</code>
<code>itemCaptionPropertyId</code>	<code>Object</code>	<code>null</code>
<code>itemIconPropertyId</code>	<code>Object</code>	<code>null</code>
<code>multiSelect</code>	<code>Boolean</code>	<code>false</code>
<code>newItemHandler</code>	<code>NewItemHandler</code>	<code>DefaultNewItemHandler</code>
<code>newItemsAllowed</code>	<code>Boolean</code>	<code>false</code>
<code>nullSelectionAllowed</code>	<code>Boolean</code>	<code>true</code>
<code>nullSelectionItemId</code>	<code>Object</code>	<code>null</code>

Displaying items

Components that display a single value per item (including combo-boxes and lists but excluding tables) may represent items by caption and/or by icon.

The simplest way to do that is to assign each specific item a caption and/or an icon with `setItemCaption()` and `setItemIcon()` methods respectively.

```
// select is an abstract select
select.addItem(person);
select.setItemCaption(person, person.getFirstName() + " " + person.
getLastName());
```

However, in this case, we have to add items one by one and lose the ability to initialize the component with a container datasource (which is the whole point).

`Abstract select` has a mode property that lets us manage it as such. Available values for this property are found in the `com.vaadin.ui.AbstractSelect`. `ItemCaptionMode` constant:

Constant	Computed caption value
<code>EXPLICIT</code>	None: We have to set it for each item explicitly, like in the previous example
<code>ICON_ONLY</code>	None: Only an icon is shown. An icon must be set explicitly for each item
<code>ID</code>	The <code>toString()</code> value of the item's ID
<code>ITEM</code>	The <code>toString()</code> value of the item

Constant	Computed caption value
INDEX	Item's index in the container
EXPLICIT_DEFAULTS_ID	By default, the <code>toString()</code> value of the item's ID, but individual items can be set a caption, thus overriding the default value
PROPERTY	An item property is used, which is specified with <code>setItemCaptionPropertyId()</code>

Note that setting the caption item by item, or using a general strategy is mutually exclusive, save the case of `EXPLICIT_DEFAULTS_ID` (which is the default). Also, be aware of the following:

- The defined strategy will take precedence over the manually set caption
- The framework won't say anything about it if we try to set it anyway

As an example, let's display our `Person` objects. We would like to show both the first and the last name, and nothing else really suits our needs; item IDs have a whole different purpose and redefining `toString()` seems a bad option. However, we could surely create a computed property from scratch as follows:

```
public String getDisplayName() {
    return firstName + " " + lastName;
}
```

Then using this property is just a matter of configuring the abstract select:

```
select.setItemCaptionMode(ItemCaptionMode. PROPERTY);
select.setItemCaptionPropertyId("displayName");
```



From a design point of view, it would have been cleaner to create a view object. Yet for clarity's sake, it is simpler this way.



Handling new items

New items may be added to concrete subclasses of `AbstractSelect`. The exact behavior is delegated to a `NewItemHandler` instance. The default one just checks whether the select is read-only and throws a `Property.ReadOnlyException` in this case.

Typical use cases of new item handlers include the following, among others:

- Inserting the new item as a row in the data tier (read database)
- Tracing the identity of the connected user

Null items

Use cases may/may not allow selecting null values. Such configuration can easily be achieved by calling the `nullSelectionAllowed()` method.

null values cannot be added to containers as such, but `AbstractSelect` can have items that contain null values. Just create a specific item ID (or object depending on the bean item container implementation) and call `setNullSelectionItemId()` with it as a parameter.

Now querying the value of the component when the dummy is selected will return null and not the real object:

```
BeanItemContainer<Person> container =
    new BeanItemContainer<Person>(Person.class);

// Define a person which cannot exist
Person nullPerson = new Person(-1L);

container.addItem(nullPerson);

final ListSelect select = new ListSelect("", container);

// Send events on directly when clicked
select.setImmediate(true);

// Handle the value of the person as null
select.setNullSelectionItemId(nullPerson);

select.addValueChangeListener(new ValueChangeListener() {

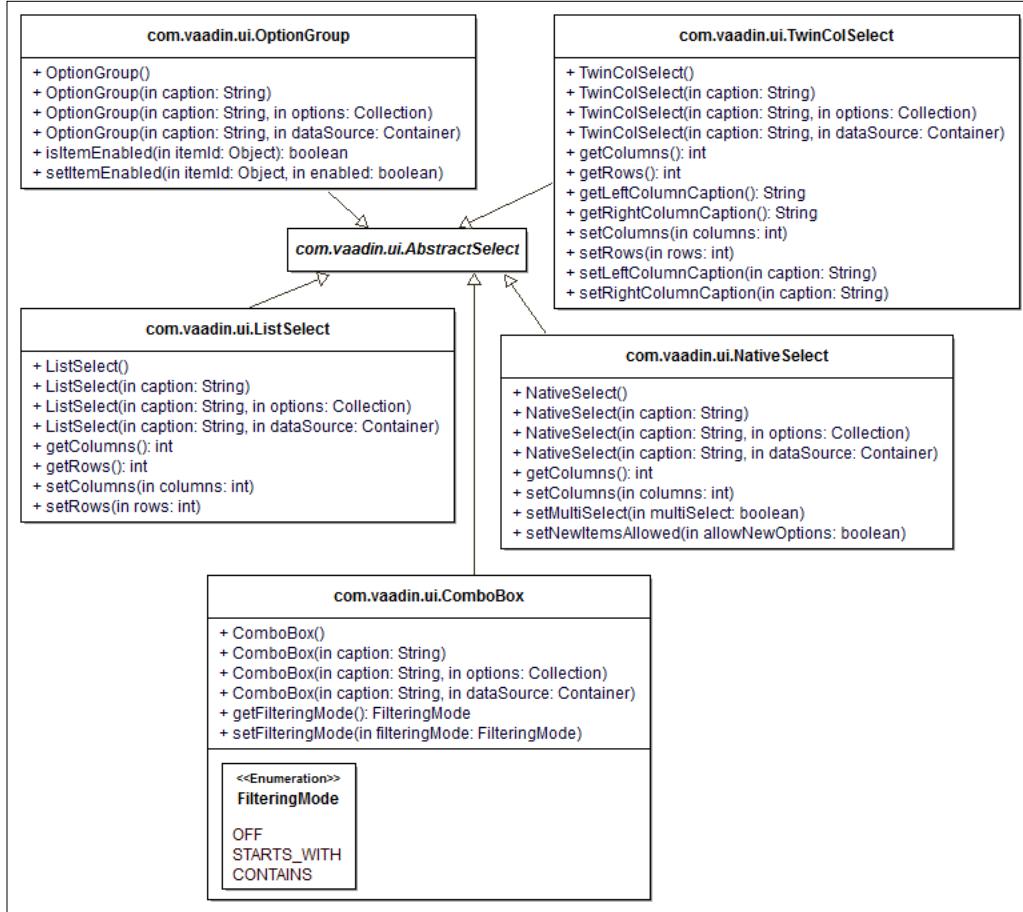
    @Override
    public void valueChange(ValueChangeEvent event) {

        System.out.println(select.getValue());
    }
});
```

Container components

Inherited from `AbstractSelect` are some concrete components that may display a container's content.

Depending on the properties described earlier and the implementation type, we can get virtually any result we need.



Note that all subclasses of `AbstractSelect` have the following constructors:

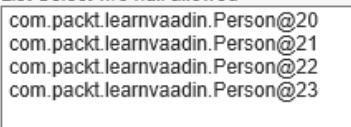
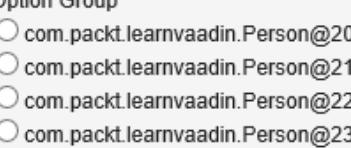
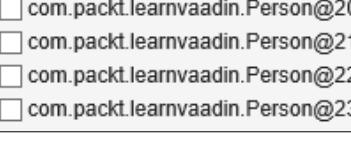
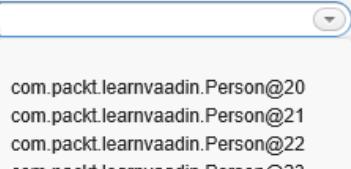
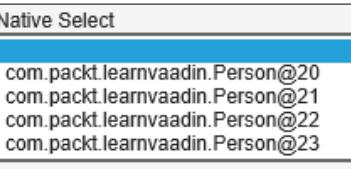
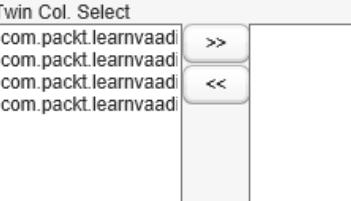
- A constructor with no parameters that comes in handy when we have no idea of the content at the time of instantiation.
- A constructor with a `String` parameter. It is the same as the previous constructor, only with a caption.
- A constructor with both `String` and `Container` parameters. The `String` references the component's caption and the `Container` references the items to be displayed by the component. If there is no caption, just use `null` for the parameter.
- A constructor with both the caption as a `String` and `Collection` parameters that will populate the underlying container.

Since a picture is worth a thousand words, here is a sample of different configurations of the preceding component, all setting the same `Person` object's `Collection`:

```
Person person1 =
    new Person(1L, "John", "DOE", new Date(70, 0, 1));
Person person2 =
    new Person(2L, "Jane", "doe", new Date(70, 0, 1));
Person person3 =
    new Person(3L, "jules", "winnfield", new Date(48, 11, 21));
Person person4 =
    new Person(4L, "vincent", "Vega", new Date(54, 1, 17));

List<Person> persons =
    Arrays.asList(person1, person2, person3, person4);
```

Code	Representation
<pre>ListSelect select = new ListSelect("List Select", persons);</pre>	<p>List Select</p> <div style="border: 1px solid black; padding: 10px;"><pre>com.packt.learnvaadin.Person@20 com.packt.learnvaadin.Person@21 com.packt.learnvaadin.Person@22 com.packt.learnvaadin.Person@23</pre></div>

Code	Representation												
<pre>ListSelect select = new ListSelect("List Select w/o null allowed", persons); select.setNullSelection Allowed(false);</pre>	<p>List Select w/o null allowed</p>  <ul style="list-style-type: none"> com.packt.learnvaadin.Person@20 com.packt.learnvaadin.Person@21 com.packt.learnvaadin.Person@22 com.packt.learnvaadin.Person@23 												
<pre>OptionGroup select = new OptionGroup ("Option Group", persons);</pre>	<p>Option Group</p>  <ul style="list-style-type: none"> <input type="radio"/> com.packt.learnvaadin.Person@20 <input type="radio"/> com.packt.learnvaadin.Person@21 <input type="radio"/> com.packt.learnvaadin.Person@22 <input type="radio"/> com.packt.learnvaadin.Person@23 												
<pre>OptionGroup select = new OptionGroup("Option Group w/ multiselect", persons); select. setMultiSelect(true);</pre>	<p>Option Group w/ multiselect</p>  <ul style="list-style-type: none"> <input type="checkbox"/> com.packt.learnvaadin.Person@20 <input type="checkbox"/> com.packt.learnvaadin.Person@21 <input type="checkbox"/> com.packt.learnvaadin.Person@22 <input type="checkbox"/> com.packt.learnvaadin.Person@23 												
<pre>ComboBox select = new ComboBox("Combo Box", persons);</pre>	<p>Combo Box</p>  <ul style="list-style-type: none"> com.packt.learnvaadin.Person@20 com.packt.learnvaadin.Person@21 com.packt.learnvaadin.Person@22 com.packt.learnvaadin.Person@23 												
<pre>NativeSelect select = new NativeSelect("Native Select", persons);</pre>	<p>Native Select</p>  <ul style="list-style-type: none"> com.packt.learnvaadin.Person@20 com.packt.learnvaadin.Person@21 com.packt.learnvaadin.Person@22 com.packt.learnvaadin.Person@23 												
<pre>TwinColSelect select = new TwinColSelect("Twin Col. Select", persons);</pre>	<p>Twin Col. Select</p>  <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">com.packt.learnvaadin.Person@20</td> <td style="padding: 5px; text-align: center;">>></td> <td style="padding: 5px;"></td> </tr> <tr> <td style="padding: 5px;">com.packt.learnvaadin.Person@21</td> <td style="padding: 5px; text-align: center;"><<</td> <td style="padding: 5px;"></td> </tr> <tr> <td style="padding: 5px;">com.packt.learnvaadin.Person@22</td> <td style="padding: 5px;"></td> <td style="padding: 5px;"></td> </tr> <tr> <td style="padding: 5px;">com.packt.learnvaadin.Person@23</td> <td style="padding: 5px;"></td> <td style="padding: 5px;"></td> </tr> </table>	com.packt.learnvaadin.Person@20	>>		com.packt.learnvaadin.Person@21	<<		com.packt.learnvaadin.Person@22			com.packt.learnvaadin.Person@23		
com.packt.learnvaadin.Person@20	>>												
com.packt.learnvaadin.Person@21	<<												
com.packt.learnvaadin.Person@22													
com.packt.learnvaadin.Person@23													

Notice that in the previous snippets, configuring the same component in different ways gets us different graphical representations.

For a practical example, consider we want the user to select a single Person object. Space requirement constrains us to use the smallest space possible. Moreover, there may be many Person objects available: it would be a good idea to let the user type some characters in order to filter out choices. In this case, our component of choice is ComboBox, as it does not use much space and lets us filter options:

```
import java.util.Arrays;
import java.util.Date;

import com.vaadin.data.Property.ValueChangeEvent;
import com.vaadin.data.Property.ValueChangeListener;
import com.vaadin.data.util.BeanItemContainer;
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.AbstractSelect;
import com.vaadin.ui.ComboBox;
import com.vaadin.ui.Notification;
import com.vaadin.ui.UI;

@SuppressWarnings("serial")
public class SelectPersonUI extends UI {

    @SuppressWarnings("deprecation")
    @Override
    protected void init(VaadinRequest request) {

        Person person1 =
            new Person(1L, "John", "DOE", new Date(70, 0, 1));
        Person person2 =
            new Person(2L, "Jane", "doe", new Date(70, 0, 1));
        Person person3 =
            new Person(3L, "jules", "winnf", new Date(48, 11, 21));
        Person person4 =
            new Person(4L, "vincent", "Vega", new Date(54, 1, 17));

        BeanItemContainer<Person> container =
            new BeanItemContainer<Person>(Person.class);

        container.addAll(Arrays.asList(
            person1, person2, person3, person4));

        ComboBox combo = new ComboBox("", container);
```

```
combo.setImmediate(true);
combo.setNullSelectionAllowed(true);
combo.setItemCaptionPropertyId("lastName");

combo.addValueChangeListener(new ValueChangeListener() {

    public void valueChange(ValueChangeEvent event) {

        AbstractSelect combo =
            (AbstractSelect) event.getProperty();

        Person selected = (Person) combo.getValue();

        if (selected == null) {

            Notification.show("null");

        } else {

            Notification.show(selected.getId() + " (" +
                + selected.getFirstName() + ")");
        }
    }
});;

setContent(combo);
}
}
```

There are some important points in the preceding code:

- `setItemCaptionPropertyId("lastName")` lets us use the `lastName` property of the `Person` item as its caption. We could have used any other property, of course, but it fits for a label.

If there would have been probable duplicates (as is the case in databases), we should probably have used a computed property (for example, the aggregation of both first and last names).

- In `ComboBox`, filtering is enabled by default in with the "starts with" pattern. There is nothing more to code to make it work.
- Last but not the least, getting the select's value returns the item itself! This means that getting and setting the selected value is a breeze.

Value type

The return type in the signature of the `getValue()` method is `Object`. We can easily cast it to the item type, in our example, `Person`. Be aware however, that when multiselection is enabled, the returned value is a set of all selected item IDs. When multiselection can be enabled or disabled, take extra care when casting the return value.

Tables

Tables merit their own section as they display multiple columns, something the components in the previous section are not meant to handle.

Besides specific event-listener pairs, tables add the following important features to a simple select:

- Computed columns, that is, columns not found in the underlying container. Does the `displayName` property ring any bells?
- Configurable columns, in order to display exactly what we want. It includes displaying date values with the right format but also using checkboxes for Boolean values, and so on.
- Drag-and-drop; tables are eligible as both source and target.
- A viewpoint around a very, very high number of items.

Consistence of table hierarchy

In essence, tables are abstract selects, of sorts. This means that they add methods and behaviors of their own, but some defined in their parent class have no meaning, for example; for `Table` instances, the `itemCaptionMode` property makes no sense.

Table structure

The first thing to understand for Vaadin tables is how they are structured.

	Prop.1 header	Prop.2 header	Prop.n header	Gen. col.1 header
item1.row header	item1.property1	item1.property2	item1.propertyn	item1.gencol1
Item2.row header	item2.property1	item2.property2	item2.propertyn	Item2.gencol1

	Prop.1 header	Prop.2 header	Prop.n header	Gen. col.1 header
Itemn.row header	Itemn.property1	itemn.property2	itemn.propertyn	Itemn.gencol1
	Prop.1 footer	Prop.2 footer	Prop.n footer	Gen. col.1 footer

Columns

Each table's column is referenced by a property ID. Those IDs may be explicitly set, but in most cases, it is the property's name of the bean item type stored in the underlying container. For our Person type, they are `firstName`, `lastName`, and so on.

Column properties can be set in either of the following ways:

- Globally, where the method expects an array of the right type values as a single parameter or varargs
- Column by column, where parameters are respectively the property ID and a value of the desired type

For example, should we want to set column headers in one line, we could do the following:

```
table.setColumnHeaders(new String[] { "First Name", "Last Name",
    "Birth date", "ID" });
```

Alternatively, the same result could be achieved with the following:

```
table.setColumnHeader("firstName", "First Name");
table.setColumnHeader("lastName", "Last Name");
table.setColumnHeader("birthdate", "Birth date");
table.setColumnHeader("id", "ID");
```

Global column properties are summed up in the following table:

Property	Type	Default value
<code>columnAlignments</code>	<code>Align...</code>	
<code>columnHeaders</code>	<code>String[]</code>	
<code>columnIcons</code>	<code>Resource[]</code>	
<code>visibleColumns</code>	<code>Object[]</code>	

The following table recaps single column "properties":

Property	Type	Default value
columnAlignment	String	Align.LEFT
columnCollapse	boolean	
columnCollapsible	boolean	
columnExpandRatio	float	
columnFooter	String	null
columnHeader	String	
columnIcon	Resource	null
columnWidth	int	-1

Most are self-describing; however, a little explanation on how table width works in Vaadin would be in order.

Width

In essence, it boils down to how the framework integrates with HTML: when the column width is set to `-1`, no width is set in it and thus, width is based on both CSS and available space in the page; if not, width is just set by the code.

Collapsing

Vaadin allows us to hide some columns at first, but provides us with the means to display them later. This is known as a **collapsing column**. First, we have to call `setColumnCollapsingAllowed(true)` in order to enable the feature.

Then, individual columns may be collapsed with the `setColumnCollapsed()` method or manually by the user. This code collapses the unnecessary columns:

```
table.setColumnCollapsed("firstName", true);
```

A screenshot of a Vaadin table component. The table has three visible columns: ID, LASTNAME, and BIRTHDATE. The fourth column is hidden and represented by a dropdown menu. The menu is titled with a downward arrow icon and contains four items: 'id' (selected), 'lastName', 'birthdate', and 'firstName'. The table data is as follows:

ID	LASTNAME	BIRTHDATE	
1	DOE	Jan 1, 1970 12:00:00 AM	• id
2	doe	Jan 1, 1970 12:00:00 AM	• lastName
3	winnfield	Dec 21, 1948 12:00:00 AM	• birthdate
4	Vega	Feb 17, 1954 12:00:00 AM	firstName

In the preceding screenshot, see how Vaadin displays a selector to choose columns to be shown.

 **Table width and collapsing**

Beware that collapsed columns are not used when computing the total table width. It is advised to explicitly set the table width in order to ensure that uncollapsed columns will get enough space to be put in view with no need to scroll.

Header and footer

Header and footer are structuring elements of tables.

The former code snippet showed us how to set headers explicitly, even if most of the time, the strategy used for creating headers (which is the same as the one for creating captions, see the *Changing captions* section of this chapter) from item properties is good enough.

Like captions, we can change the strategy used with `setColumnHeaderMode` (`ColumnHeaderMode`):

Constant	Computed row header value
HIDDEN	No column header is shown
EXPLICIT	None: We have to set each column header explicitly
EXPLICIT_DEFAULTS_ID	Default: A spaced uppercased property is used, but individual columns can be set a header, thus overriding the computed value
ID	Spaced uppercased property

As an illustration, let us update our table with more user-friendly headers:

```
table.setColumnHeader("firstName", "Given name");
table.setColumnHeader("lastName", "Family name");
table.setColumnHeader("birthdate", "Birth Date");
```

ID	GIVEN NAME	FAMILY NAME	BIRTH DATE
1	John	DOE	Jan 1, 1970 12:00:00 AM
2	Jane	doe	Jan 1, 1970 12:00:00 AM
3	jules	winnfield	Dec 21, 1948 12:00:00 AM
4	vincent	Vega	Feb 17, 1954 12:00:00 AM

Note that column headers are more than what we would expect as regular users.

Footers *must* be set independently as they are empty by default. Besides, we have to call `setFooterVisible(true)` to display the entire footer bar, as it is hidden otherwise.

Row header column

The row header column is a special column hidden by default. Think of it as a summary of the row item, made comprehensible for mere humans. In fact, it works exactly the same as the caption mode of `AbstractSelect`, only the method is `setRowHeaderMode(RowHeaderMode)` and the constants are as follows:

Constant	Computed row header value
HIDDEN	Default value: No header column is shown
EXPLICIT	None: We have to set it for each item explicitly
ICON_ONLY	None: Only an icon is shown. An icon must be set explicitly for each item
ID	The <code>toString()</code> value of the item's ID
ITEM	The <code>toString()</code> value of the item
INDEX	The item's index in the container
EXPLICIT_DEFAULTS_ID	By default, it is the <code>toString()</code> value of the item's ID, but individual items can be set a header, thus overriding the default value
PROPERTY	An item property is used, which is specified with <code>setItemCaptionPropertyId()</code>

In our current table, we could use the `displayName` computed property as the row header:

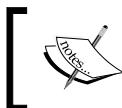
```
table.setRowHeaderMode(RowHeaderMode.PROPERTY);  
table.setItemCaptionPropertyId("displayName");
```

This would output the following:

	ID	FIRSTNAME	LASTNAME	BIRTHDATE
John Doe	1	John	Doe	Jan 1, 1970 12:00:00 AM
Jane Doe	2	Jane	Doe	Jan 1, 1970 12:00:00 AM
Jules Winnfield	3	Jules	Winnfield	Dec 21, 1948 12:00:00 AM
Vincent Vega	4	Vincent	Vega	Feb 17, 1954 12:00:00 AM

Ordering and reordering

By default, column ordering is random. In this regard, it is always better to explicitly set column ordering with the `setVisibleColumns(String[])` method that takes an array of item properties as an argument.



Note that Vaadin will check that column names exist inside the container. That is why we should only call this method after setting the datasource.



Users cannot change the column ordering. By calling `setColumnReorderingAllowed(true)`, this feature can be enabled.



Note that none of these changes affect the row header column.



In our example, the following order suits our needs just fine (and has been used in the previous screenshots):

```
table.setVisibleColumns(new String[] {"id", "firstName", "lastName",
"birthdate"});
```

Formatting properties and generated columns

Formatting table properties can be achieved through the following three different means:

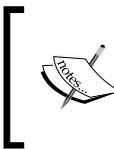
- Using a property formatter (see the *Property formatter* section of this chapter). It would be particularly unwieldy as it would require wrapping the to-be-formatted property of each item individually.
- Overriding the protected `formatPropertyValue(Object, Object, Property)` method. For example, let's format Person birth dates in our table with the French locale:

```
Table table = new Table("", container) {

    @Override
    protected String formatPropertyValue(Object itemId,
        Object propId, Property property) {

        Object value = property.getValue();
```

```
if (value instanceof Date) {  
  
    Date date = (Date) value;  
  
    DateFormat format = DateFormat.getDateInstance(  
        DateFormat.SHORT, Locale.FRENCH)  
  
    return format.format(date);  
}  
  
return super.formatPropertyValue(  
    itemId, propId, property);  
};  
};
```



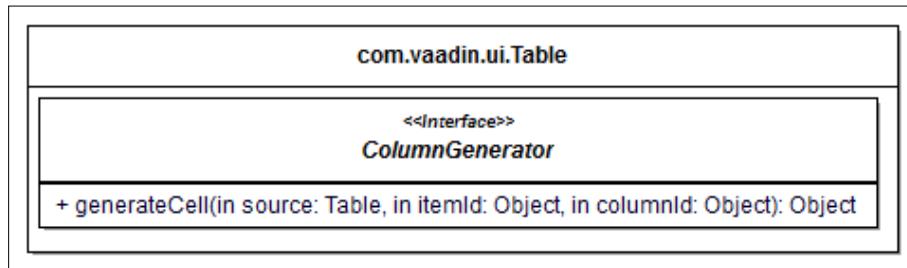
Alternatively, we could have checked on the property's ID. With our choice, if other date attributes are added to the underlying JavaBean, they will benefit from the formatting as well (which may / may not be desirable).

This approach not only defeats separation of concerns, it also renders the code less readable.

- In abstract selects, we had to explicitly create the `displayName` computed property on `Person`. It was a bad design, but alternatives were either complex or unsatisfying.

Table offers a feature that `AbstractSelect` does not: **generated columns**. These columns are the answer to the previous quandary and implements it in a clean way.

In order to achieve this, Vaadin introduces the `ColumnGenerator` interface: it overrides the wanted property's name, if it already exists, or creates a new one if it does not.



 **Generated cell return value**

The method return type is `Object`: in effect, we can return either a `Component` instance, or a `String`. In the latter case, Vaadin will wrap the `String` value inside `Label` to display.

Once implemented, the column generator can be added to the table with a specific property ID. Let's replace the `displayName` property on the `Person` with an equivalent generated column implementation:

```
import com.vaadin.data.Item;
import com.vaadin.data.Property;
import com.vaadin.ui.Table;
import com.vaadin.ui.Table.ColumnGenerator;

@SuppressWarnings("serial")
public class DisplayNameColumnGenerator implements ColumnGenerator {

    @Override
    @SuppressWarnings("unchecked")
    public Object generateCell(Table source,
        Object itemId, Object columnId) {

        // item is the Person (or the line of the table)
        Item item = source.getItem(itemId);

        Property<String> firstName =
            item.getItemProperty("firstName");
        Property<String> lastName =
            item.getItemProperty("lastName");

        return firstName.getValue() + " " + lastName.getValue();
    }
}
```

Now, it's just a matter of using the generated column:

```
table.addGeneratedColumn("displayName",
    new DisplayNameColumnGenerator());
table.setColumnHeader("displayName", "Display name");
table.setVisibleColumns(
    new String[] { "id", "displayName", "birthdate" });
```



Notice that generated columns cannot be sorted (see below) as there is no underlying property the container may be aware of.



Additionally, we can use generated columns as a way to change (for example, format) existing property columns. Let us create a column generator that will format dates according to the French locale:

```
import static java.text.DateFormat.SHORT;
import static java.util.Locale.FRENCH;

import java.text.DateFormat;
import java.util.Date;

import com.vaadin.data.Item;
import com.vaadin.data.Property;
import com.vaadin.ui.Table;
import com.vaadin.ui.Table.ColumnGenerator;

@SuppressWarnings("serial")
public class DateColumnGenerator implements ColumnGenerator {

    @Override
    @SuppressWarnings("unchecked")
    public Object generateCell(Table source, Object itemId,
        Object columnId) {

        // item is the Person (or the line of the table)
        Item item = source.getItem(itemId);

        // property is the date property
        // (or the intersection between the line and the column)
        Property<Date> property = item.getItemProperty(columnId);

        Date value = property.getValue();

        DateFormat format =
            DateFormat.getDateInstance(SHORT, FRENCH);

        return format.format(value);
    }
}
```

Now, it's just a matter of using the generated column:

```
table.addGeneratedColumn("birthdate", new DateColumnGenerator());
```

As the `birthdate` property already exists in the container, the date column generator will replace it. As an icing on the cake, the implemented column generator is able to decorate *any date property on any table*.

Sorting

By default, Vaadin tables are sortable: users can choose a column to sort from, by clicking on the column header.



Note that sorting is executed on the underlying property value, and not on its string representation.



User sorting

In order to prevent the user sorting, there are the following two options:

- Calling the `setSortDisabled(true)` method
- Hiding all column headers with `setColumnHeaderMode(ColumnHeaderMode.HIDDEN)`, as seen previously

Programmatic sorting

Alternatively, developers can use server-side sorting using two parameters: the property's ID and an indicator hinting at whether the sort is ascending (which is the default).

For example, the following snippet sorts the table from the last name, in the descending order:

```
table.setSortContainerPropertyId("lastName");
table.setSortAscending(false);
```

This approach has a strong limitation; it can only be used for sorting on a single column. In order to overcome this, remember that `Container.Sortable` provides the `sort(Object[] propertyId, boolean[] ascending)` method to use multiple columns at once.

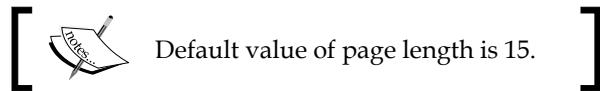
Also, it lets us sort in a single method call, as the following snippet proves, which is equivalent to the former:

```
table.sort(new String[] {"lastName"}, new boolean[] {false});
```

Viewpoint

Containers can hold a very high number of items and their associated table components are still reactive, as not all are displayed to the user. In order to achieve this, tables use the concept of viewpoint: the number of rows shown is constrained by the table (and fewer than the number of items).

First, we can set the number of visible rows with the `setPageLength(int)` method.



Notice that in combination to row height, it sets the default table height. If set to 0, the table will adjust its height to display all of the items in the container: be sure that it is the desired behavior, as it can have a great impact on both performance and ease of use.

On the contrary, not setting the page length to 0 means that the table will not adhere to the `setHeight()` contract. This is of utmost importance when trying to fit the table component in a specific place, for example, when `setSizeFull()` must scale the table accordingly.

Note that the page length will be updated on the client side if the table height is defined.

Additionally, we can programmatically scroll to the first item in the list, either by the item's ID or by its index. This is done with the `setCurrentPageFirstItemId(Object)` and `setCurrentPageFirstItemIndex(int)` methods respectively.

Viewpoint change event

Table does not come with an event model around scrolling visible items. Out of the box, the framework displays the range of visible items when the user scrolls at the top of the table, just below the column headers. If we need to be informed about scrolling, we have to override the protected `refreshRenderedCells()` method in order to implement the desired behavior (and still call the parent method, of course).

Improving responsiveness

Scrolling the viewpoint in order to show different items will make the table fetch newly visible items from the underlying container. As such, there would be a noticeable waiting time for the user if not for a nifty feature of the framework.

Vaadin table fetches more items than it is needed to be displayed and caches the superfluous items in memory. The number of such items is `pageLength` times the value of a property named `cacheRate`, above and below the table. Therefore, if users complain about response time, increasing the cache rate could be a good idea.

Of course, if the user scrolls too fast, it won't do anything. In most cases however, it just increases the responsiveness with no side effect.

Editing

Until now, we have learned how to configure how we want to display data, and that is no mean feat. Nonetheless, displaying is one thing, editing is another.

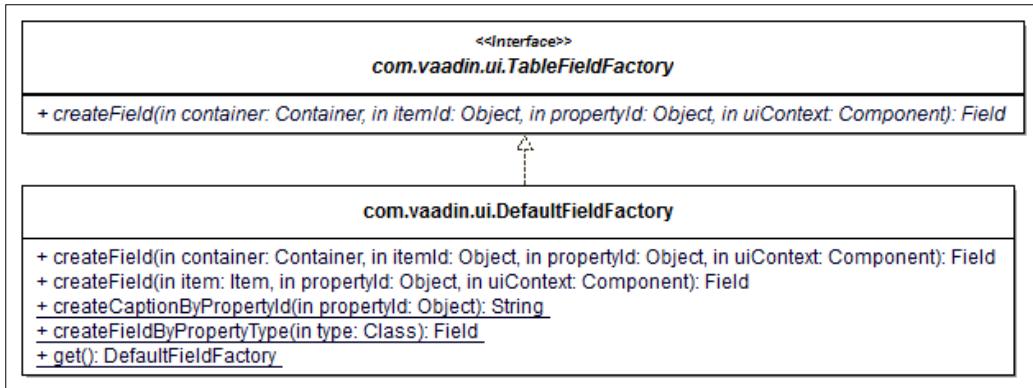
In Vaadin, making a table editable is as simple as calling `setEditable(true)` on it. Behold the result: in just one line, we have a fully-editable table:

ID	DISPLAYNAME	FIRSTNAME	LASTNAME	BIRTHDATE
1	John DOE	John	DOE	1/1/70 
2	Jane doe	Jane	doe	1/1/70 
3	jules winnfield	jules	winnfield	12/21/48 
4	vincent Vega	vincent	Vega	2/17/54 

The following things are worth noticing:

- Editable fields are specifically tailored to the property type and in exactly the same way as for single fields (see the *Configuring field types* section of this chapter).
- Computed properties, such as the `displayName` property, are still shown as simple labels (and not as a field).
- This is also the case for properties with no setter, such as the ID.

In fact, like forms, tables delegate field generation to a factory; a `TableFieldFactory` interface and `DefaultFieldFactory` also implements it, as presented in the following figure:



Tweaking the field of our editable table is just a matter of implementing the right table field factory.

Let us pretend that we want to display inline calendars instead of popups to edit dates (not a really useful idea in fact). It is just a matter of coding the desired implementation:

```
public class InlineDateTableFieldFactory
    implements TableFieldFactory {

    @Override
    public Field<?> createField(Container container,
        Object itemId, Object propertyId, Component uiContext) {

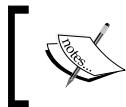
        Item item = container.getItem(itemId);

        Property<?> property = item.getItemProperty(propertyId);

        if (Date.class.isAssignableFrom(property.getType())) {

            return new InlineDateField(property);
        }

        return DefaultFieldFactory.get().createField(
            container, itemId, propertyId, uiContext);
    }
}
```



As for the previous field factory, remember to favor composition over inheritance (http://wikipedia.org/wiki/Composition_over_inheritance)

Then, it is just a matter of setting our factory on the table:

```
table.setTableFieldFactory(new InlineDateTableFieldFactory());
```

Notice that it is very similar to our previous custom form field factory. Well, it is even simpler as column header is not handled (and should not be).

Selection

On the client side, items in standard tables are just a bunch of characters put one next to another.

Just calling the `setSelectable(true)` method on the table will make each row appear as a single object to the user and selectable as such.

By default, only a single row can be selected at a time. In order to select multiple rows, we need to call `setMultipleSelect(true)` on the table. When multiple selections are enabled, users can select a range of rows with the *Shift* key and individual rows can be added or removed from the selection with the *Ctrl* key.

Selected rows can be retrieved by invoking the `getValue()` method on the table. The returned value is of runtime type:

- `Set<?>` if multiple selection is enabled
- The `Item` type item if it is not (in our running example, that would be `Person`)

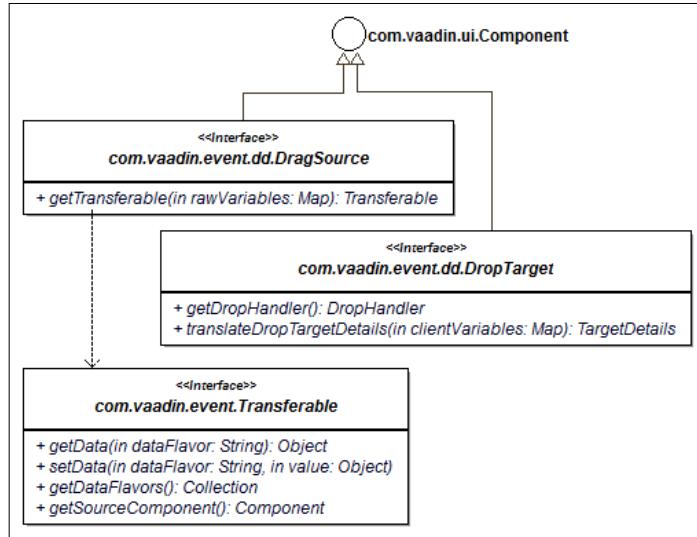
Drag-and-drop

Tables are a good entry point into Vaadin's drag-and-drop capabilities.

The framework uses the following abstractions in order to accomplish drag-and-drop:

- `Transferable`: This represents the transferred data
- `DragSource`: This stands for the source component

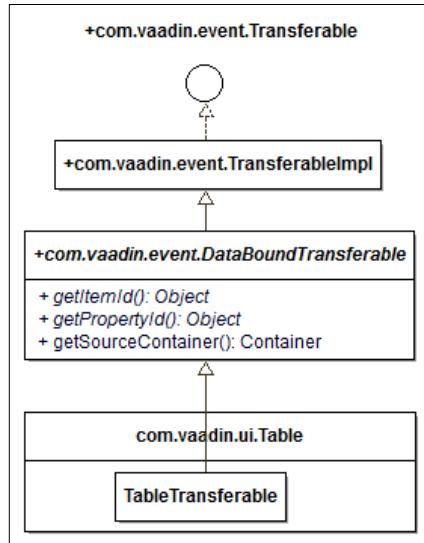
- DropTarget: This denotes the target component:



Transferable

`Transferable` wraps the data to be dragged-and-dropped between components.

`TransferableImpl` is just a straightforward implementation of `Transferable`. As `Transferable` may apply to a great number of differently structured data, it is built around a hash map:



Keys are called **data flavors** and vary, depending on the concrete type of transferable. We can query for all data flavors of a particular transferable with the `getDataFlavors()` method. `DataBoundTransferable` is a specialized transferable designed for containers.

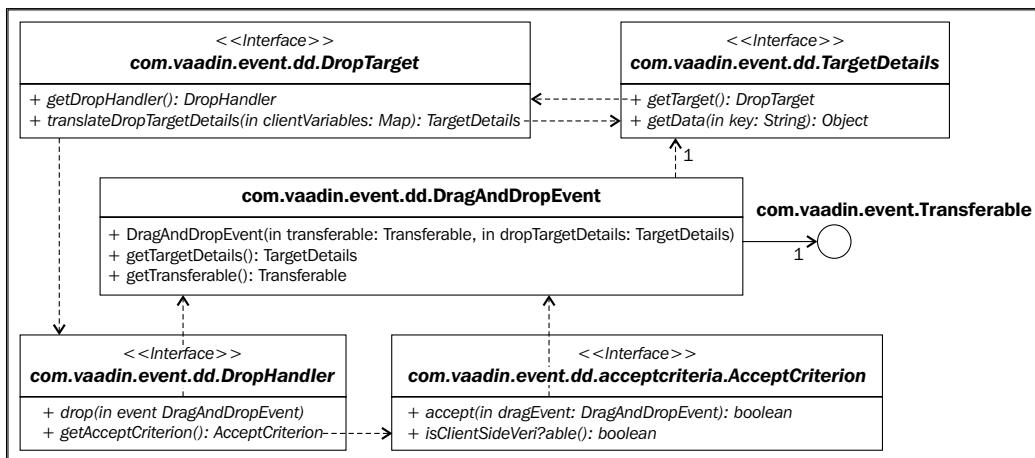
At the table level, drag sources are cells, so `TableTransferable` is an implementation where we can get both a cell's item ID and its property ID.

Drag source

`DragSource` knows how to create transferable objects. Concrete classes implement the single `getTransferable(Map<String, Object>)` method that return `Transferable` instances.

Drop target

Drop target model design is somewhat more complex than drag source, for the wanted behavior may need to be provided by collaborating classes.

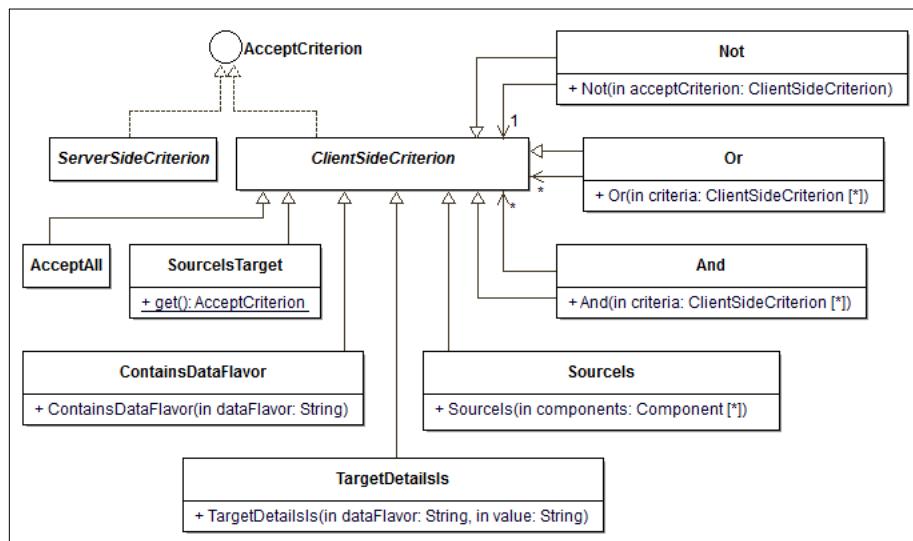


- At the heart of the model lies `DragAndDropEvent`, which is not a real event *per se* (it does not inherit from `EventObject`) but is still sent by the framework. It encapsulates both a `Transferable` and a `TargetDetails` instance.
- `TargetDetails` in turn wraps the drop target as well as all information contained in the aforementioned `Transferable`. Concrete target details classes are provided throughout Vaadin by components, including tables (and trees).
- `DropTarget` represents the target of the drag-and-drop operation. It delegates the drop itself to a `DropHandler`.

- DropHandler is the class responsible for managing what really happens in the drop through the `drop()` method. Drop handlers have to detail under what conditions the drop is valid.
- AcceptCriterion wraps possibly many criteria in order to determine whether to drop the transferable on the target or to abort the operation.

Accept criterion

Accept criterion can be separated into two main groups: criteria that can work solely on the client side and criteria that need server-side access.



In order to ease the understanding of the diagram, package was not represented. It is `com.vaadin.event.dd.acceptcriteria`.

The different types of criteria are fairly self-describing.

Table drag-and-drop

Table is a component which is drag-and-drop ready, having concrete implementations of all previous interfaces, save `DropHandler`, which has to be application-specific anyway.

- In order to enable dragging from the table, we just have to call `setDragMode(TableDragMode)` with the right value: either `ROW` or `MULTIROW`. It is `NONE` by default, meaning the drag is disabled. Note that in order for multi-row to work, the table must be selectable with multi-selection enabled.

- To enable dropping to a table, we just have to implement `DropHandler` and associate a new instance of it with the table.

The following `DropHandler` example creates a new person from the dragged person and drops it at the end of the table:

```
public class DuplicatePersonDropHandler implements DropHandler {

    @Override
    public void drop(DragAndDropEvent event) {

        TableTransferable transferable =
            (TableTransferable) event.getTransferable();

        Object itemId = transferable.getItemId();

        Table table = transferable.getSourceComponent();

        @SuppressWarnings({ "unchecked", "rawtypes" })
        BeanItem<Person> item = (BeanItem) table.getItem(itemId);

        Person originalPerson = item.getBean();

        Person newPerson = new Person(null);

        newPerson.setFirstName(originalPerson.getFirstName());
        newPerson.setLastName(originalPerson.getLastName());
        newPerson.setBirthdate(originalPerson.getBirthdate());

        table.addItem(newPerson);
    }
}
```

Then, we set it on the table and we do not forget to make it selectable with single row selection:

```
table.setSelectable(true);
table.setDragMode(TableDragMode.ROW);

table.setDropHandler(new DuplicatePersonDropHandler());
```

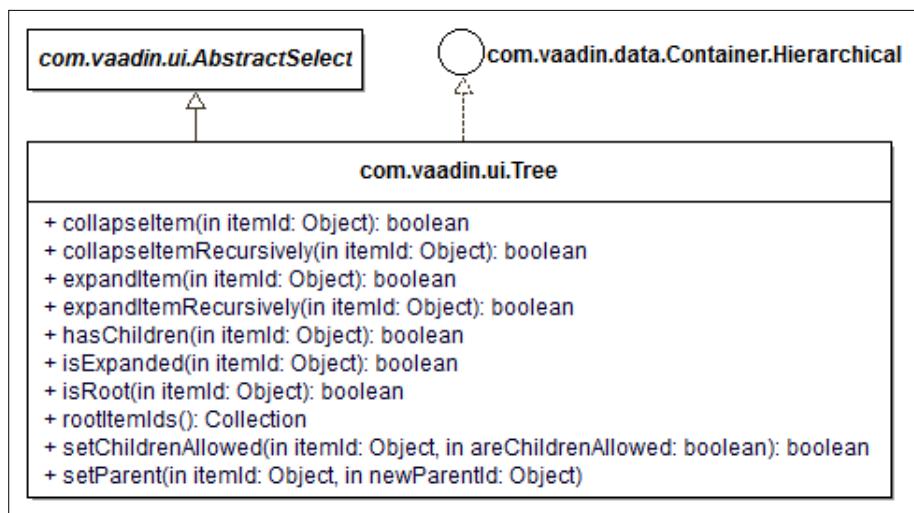
- We set the drag mode to `ROW`, in order for the table to be a drag source
- We created a drop handler and set it on the table, so as to make the latter a drag source too
- Finally, the accept criterion constrains the drop source so that it can only be the table itself

Trees

Trees are another component that has the capability to use a container datasource. Tables are meant to display flat containers; whereas trees are meant to do the same for hierarchical ones (see the section named *Hierarchical* of this chapter for a reminder).

From a UI point of view, trees are about nodes and parent-child relationships between them.

[ Note that a node can have a single parent (or none at all for root nodes).]



Tree is much simpler than Table as there are no columns involved: only nodes are displayed, albeit in a hierarchical fashion.

Collapse and expand

One of features of Tree is the ability to let us either collapse a node (that is, hide its children) or expand it (that is, display its children) programmatically.

The following two method flavors are available:

- One that just acts upon the node and its direct children, `collapseItem(Object)` and `expandItem(Object)`
- The other proceeds recursively from the node parameter, `collapseItemsRecursively(Object)` and `expandItemsRecursively(Object)`

One can also query for the expanded status of a particular node with the `isExpanded(Object)` method.

In all these methods, the `Object` parameter is the item's ID.

Parent and child

As trees are all about parent and child nodes, Vaadin provides the following methods in order to manage relationships:

- With the `rootItemIds()` method, we can get the root item IDs. This implies that there can be more than a single root node.
- For a particular item, `isRoot()` returns the root status and `hasChildren(Object)` returns whether it has children. Both methods are pretty self-explanatory.

Then, a node leaf status can be set with the help of the `setChildrenAllowed()` method.

Finally, one can change the entire node structure by using `setParent()`. For example, the following snippet simply rearranges the second and the fourth items respectively under the first and third items for a 4-items tree:

```
// Assume that the container is the same as in the table examples
Tree tree = new Tree("", container);

Iterator<?> iterator = tree.getVisibleItemIds().iterator();

tree.setParent(iterator.next(), iterator.next());
tree.setParent(iterator.next(), iterator.next());
```

Item labels

Labels are handled the same way for trees as for selects, meaning we should use the same solutions as seen in the *Displaying items* section, earlier in this chapter.

Refining Twaattin

We are now ready to connect Twaattin to the Twitter API. We will use Twitter4J for this purpose, available at <http://twitter4j.org/>. Twitter4J is a Java API facade over HTTP REST JSON requests.

It is open source, free, and well designed so it just fits our needs. As this book is not about it, detailed information can be found on the relevant website <http://twitter4j.org>.

Prerequisites

In order to use Twitter for authentication, the first thing to do is to register a client application using the form at <https://dev.twitter.com/apps/new>.

This will get us a consumer key and a consumer secret that will have to be passed to Twitter to authenticate (using the OAuth protocol under the cover).



Detailed configuration of Twitter4J is well beyond the scope of this book. Suffice it to say, we need to write the previous key/secret pair in `twitter4j.properties` at the root of the application classpath or pass them as system properties on the command line. For complete instructions, please go to <http://twitter4j.org/en/configuration.html>.

In order to use Twitter4J API, we need a dependency on its binary library. As dependencies are already managed by Ivy, add the following line to the `ivy.xml` file in the `dependencies` section:

```
<dependency org="org.twitter4j" name="twitter4j-core" rev="3.0.3" />
```

Adaptations

The biggest adaptation to our application is that since the Twitter API uses OAuth for authentication, we can log in to Twitter rather than Twaattin. This blatantly violates Twitter API guidelines, but is the best example in the scope of this book.



OAuth is an open protocol that delegates authentication to so-called OAuth providers. More information is available at <http://oauth.net/>.

When authenticated, Twitter will return a PIN that will be used to verify our credentials in latter calls. The login screen will have to be changed in order to provide a link to Twitter's authentication window and a PIN field.

Sources

Here are the Twaattin updated sources. Only the important code is reproduced; for the complete sources, refer to <https://github.com/nfrankel/twaattin/tree/chapter6>.

The login screen

Here is code for the login screen.

```
package com.packtpub.learnvaadin.twaattin.ui;

import com.packtpub.learnvaadin.service.TwitterService;
import com.packtpub.learnvaadin.twaattin.presenter.LoginBehavior;
import com.vaadin.server.ExternalResource;
import com.vaadin.ui.Button;
import com.vaadin.ui.Link;
import com.vaadin.ui.TextField;
import com.vaadin.ui.VerticalLayout;

public class LoginScreen extends VerticalLayout {

    private static final long serialVersionUID = 1L;

    private Link twitterLink = new Link();
    private TextField pinField = new TextField();
    private Button submitButton = new Button("Submit");

    public LoginScreen() {

        setMargin(true);
        setSpacing(true);

        twitterLink.setCaption("Get PIN");
        twitterLink.setTargetName("twitterauth");
        twitterLink.setResource(new ExternalResource(
            TwitterService.get().getAuthenticationUrl()));

        pinField.setInputPrompt("PIN");

        addComponent(twitterLink);
        addComponent(pinField);
        addComponent(submitButton);

        submitButton.addClickListener(new LoginBehavior(pinField));
    }
}
```

Noteworthy parts include the following:

- The declaration of a `Link` object. Links render as HTML links to a wrapped Resource. In our case, it is wrapped around Twitter's authentication page URL.
- The target's name property for `Link` directly translates to a `target` attribute for the `a` tag and lets us create a new pop-up window. When not specified, the URL is opened in the current window.
- Finally, we use `TwitterService`, a facade over Twitter4J that allows us to conveniently focus on the needed features of the API.

The login behavior

Update of the login behavior uses Twitter4J API to authenticate.

The timeline screen

Lorem ipsum label components are replaced with a simple table.

```
package com.packtpub.learnvaadin.twaattin.ui;

import static com.vaadin.server.Sizeable.Unit.PERCENTAGE;
import static com.vaadin.ui.Alignment.MIDDLE_RIGHT;
import static java.util.Locale.ENGLISH;

import java.security.Principal;

import com.packtpub.learnvaadin.presenter.LogoutBehavior;
import com.packtpub.learnvaadin.presenter.TweetRefresherBehavior;
import com.packtpub.learnvaadin.twaattin.ui.decorator.NameColumnGenerator;
import com.packtpub.learnvaadin.twaattin.ui.decorator.ProfileImageColumnGenerator;
import com.packtpub.learnvaadin.twaattin.ui.decorator.ScreenNameColumnGenerator;
import com.packtpub.learnvaadin.twaattin.ui.decorator.SourceColumnDecorator;
import com.packtpub.learnvaadin.twaattin.ui.decorator.TweetColumnDecorator;
import com.vaadin.server.VaadinSession;
import com.vaadin.ui.Button;
import com.vaadin.ui.HorizontalLayout;
```

```
import com.vaadin.ui.Label;
import com.vaadin.ui.Table;
import com.vaadin.ui.VerticalLayout;

public class TimelineScreen extends VerticalLayout {

    private static final long serialVersionUID = 1L;

    public TimelineScreen() {

        setMargin(true);

        Label label = new Label(VaadinSession.getCurrent()
            .getAttribute(Principal.class).getName());

        Button button = new Button("Logout");

        button.addClickListener(new LogoutBehavior());

        HorizontalLayout menuBar =
            new HorizontalLayout(label, button);

        menuBar.setWidth(100, PERCENTAGE);
        menuBar.setComponentAlignment(button, MIDDLE_RIGHT);

        addComponent(menuBar);

        addComponentAttachListener(new TweetRefresherBehavior());

        Table table = new Table();

        addComponent(table);

        table.addGeneratedColumn(
            "source", new SourceColumnDecorator());
        table.addGeneratedColumn(
            "screenName", new ScreenNameColumnGenerator());
        table.addGeneratedColumn(
            "name", new NameColumnGenerator());
        table.addGeneratedColumn(
            "profileImage", new ProfileImageColumnGenerator());
        table.addGeneratedColumn(
            "text", new TweetColumnDecorator());
    }
}
```

```
        table.setColumnHeader("source", "via");
        table.setColumnHeader("screenName", "Screen name");
        table.setColumnHeader("profileImage", "");
        table.setColumnHeader("text", "Tweet");

        table.setVisibleColumns(new Object[] { "text", "name",
            "screenName", "profileImage", "createdAt", "source" });
    }
}
```

Some new features appear in the updated timeline screen:

- The first highlighted line adds a new behavior (described below) as an attach listener. Attach listeners are triggered when a component is added (that is, attached) as a child component. Previously seen listeners were triggered by user actions. As we want to display tweets when the screen is displayed, using such a listener makes sense.
- We also add some column generators to display either additional info or format the existing one.

The tweets refresh behavior

`TweetRefreshBehavior` pulls tweets from the service layer, adds them to a bean item container, and adds then the container to a table.

It represents the core of what we've seen in this chapter:

```
package com.packtpub.learnvaadin.twaattin.presenter;

import java.util.List;

import twitter4j.Status;
import twitter4j.TwitterException;

import com.packtpub.learnvaadin.service.TwitterService;
import com.vaadin.data.util.BeanItemContainer;
import com.vaadin.ui.Component;
import com.vaadin.ui.HasComponents.ComponentAttachEvent;
import com.vaadin.ui.HasComponents.ComponentAttachListener;
import com.vaadin.ui.Table;

@SuppressWarnings("serial")
public class TweetRefresherBehavior implements ComponentAttachListener
{
```

```

@Override
public void componentAttachedToContainer(
    ComponentAttachEvent event) {

    Component component = event.getAttachedComponent();

    if (component instanceof Table) {

        Table table = (Table) component;

        try {

            List<Status> statuses =
                TwitterService.get().getTweets();

            BeanItemContainer<Status> container =
                new BeanItemContainer<Status>(Status.class);

            container.addAll(statuses);

            table.setContainerDataSource(container);

        } catch (TwitterException e) {

            throw new RuntimeException(e);
        }
    }
}

```

Notice that before doing anything, the method checks whether the child component is of type `Table`. This is mandatory, for the parent component container has more than one child.

Then, we just add the whole status collection to the container.



We used `BeanItemContainer` (the whole bean is its item ID) instead of `BeanContainer`. Alternatively, we could have used the latter and employed the `id` property as the unique item ID.

Column generators

Column generators have been discussed in this chapter. As an example, let us have a look at the screen name and the image profile column generators. Both inherit from an abstract column generator that makes the underlying User easily accessible:

```
package com.packtpub.learnvaadin.twaattin.ui.decorator;

import twitter4j.User;

import com.vaadin.server.ExternalResource;
import com.vaadin.ui.Link;
import com.vaadin.ui.Table;

@SuppressWarnings("serial")
public class ScreenNameColumnGenerator
    extends AbstractUserColumnGenerator {

    private static final String TWITTER_USER_URL =
        "https://twitter.com/";

    /**
     * @return Screen name of the underlying {@link User} as a
     * {@link Link} component.
     */
    @Override
    public Object generateCell(Table source, Object itemId,
        Object columnId) {

        User user = getUser(source, itemId);

        ExternalResource resource = new ExternalResource(
            TWITTER_USER_URL + user.getScreenName());

        Link link = new Link('@' + user.getScreenName(), resource);

        link.setTargetName("screenname");

        return link;
    }
}
```

The `getUser()` method of the `Status` object returns a structured `User` object and not a simple `String`, so we need a column generator to extract relevant data and possibly augment it. In our case, we get the screen name (the @ handle) and add a link to the user Twitter profile.

We developed a name column generator to just extract the name property of a `User`.

 It would have been much nicer to just call the `setVisibleColumns()` methods with a `user.name`. Unfortunately, this feature is not implemented at the time of this writing. Keep this under watch, it can reduce your code size!

```
package com.packtpub.learnvaadin.twaattin.ui.decorator;

import twitter4j.User;

import com.vaadin.server.ExternalResource;
import com.vaadin.ui.Image;
import com.vaadin.ui.Table;

@SuppressWarnings("serial")
public class ProfileImageColumnGenerator extends AbstractUserColumnGenerator {

    /**
     * @return Profile image of the underlying {@link User} as an
     * {@link Image} component
     */
    @Override
    public Object generateCell(Table source, Object itemId,
        Object columnId) {

        User user = getUser(source, itemId);

        String url = user.getMiniProfileImageURL();

        if (url != null) {

            ExternalResource resource = new ExternalResource(url);

            return new Image("", resource);
        }

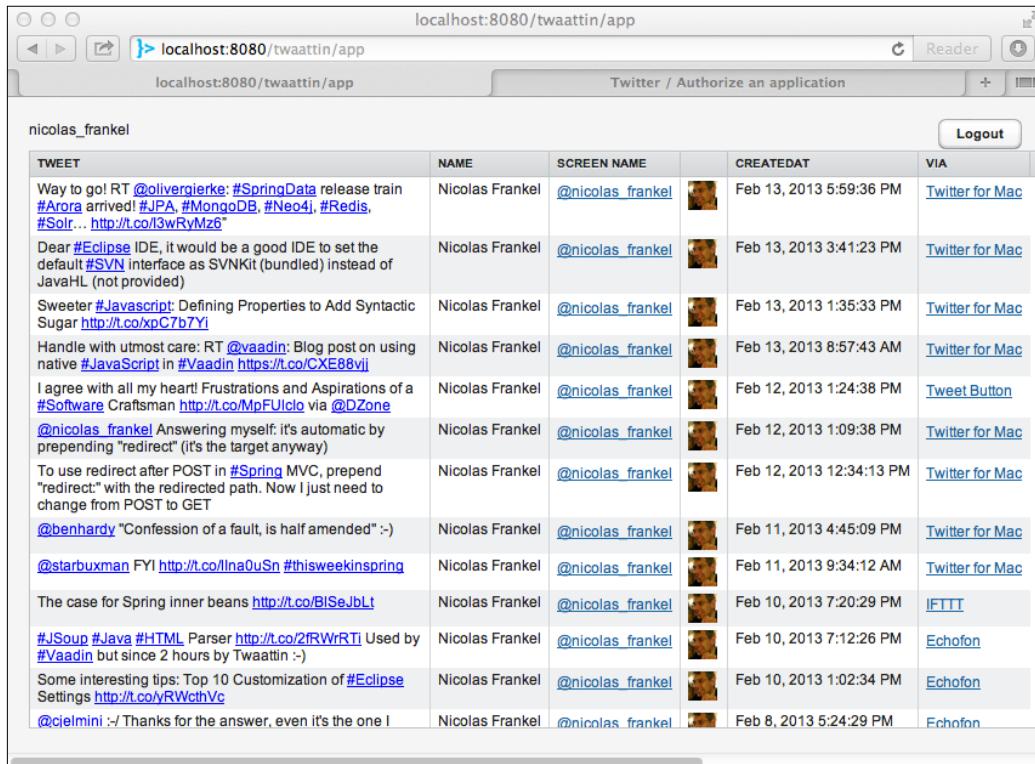
        return null;
    }
}
```

The profile column generator makes use of the `Image` class we did not see before. There is nothing special with this class; it is the class of choice to embed images in the rendered screen.

External resource performance

Be aware that external resources are rendered on the client-side; in our particular case, images are not read server-side and then sent byte by byte to the browser, rather is directly displayed by the browser. This means that we can display a large group of images without taxing our server(s).

The following screenshot is the final result of our improved Twaatin application:



TWEET	NAME	SCREEN NAME	CREATEDAT	VIA
Way to go! RT @olivergierke: #SpringData release train #Arora arrived! #JPA, #MongoDB, #Neo4j, #Redis, #Solr ... http://t.co/3wRVMz6"	Nicolas Frankel	@nicolas_frankel	Feb 13, 2013 5:59:36 PM	Twitter for Mac
Dear #Eclipse IDE, it would be a good IDE to set the default #SVN interface as SVNKit (bundled) instead of JavaHL (not provided)	Nicolas Frankel	@nicolas_frankel	Feb 13, 2013 3:41:23 PM	Twitter for Mac
Sweeter #Javascript: Defining Properties to Add Syntactic Sugar http://t.co/xpC7b7YI	Nicolas Frankel	@nicolas_frankel	Feb 13, 2013 1:35:33 PM	Twitter for Mac
Handle with utmost care: RT @vaadin: Blog post on using native #JavaScript in #Vaadin https://t.co/CXE88yjI	Nicolas Frankel	@nicolas_frankel	Feb 13, 2013 8:57:43 AM	Twitter for Mac
I agree with all my heart! Frustrations and Aspirations of a #Software Craftsman http://t.co/MpFUlclo via @DZone	Nicolas Frankel	@nicolas_frankel	Feb 12, 2013 1:24:38 PM	Tweet Button
@nicolas_frankel Answering myself: it's automatic by prepending "redirect:" (it's the target anyway)	Nicolas Frankel	@nicolas_frankel	Feb 12, 2013 1:09:38 PM	Twitter for Mac
To use redirect after POST in #Spring MVC, prepend "redirect:" with the redirected path. Now I just need to change from POST to GET	Nicolas Frankel	@nicolas_frankel	Feb 12, 2013 12:34:13 PM	Twitter for Mac
@benhardy "Confession of a fault, is half amended" :-)	Nicolas Frankel	@nicolas_frankel	Feb 11, 2013 4:45:09 PM	Twitter for Mac
@starbuxman FYI http://t.co/lIna0uSn #thisweekinspring	Nicolas Frankel	@nicolas_frankel	Feb 11, 2013 9:34:12 AM	Twitter for Mac
The case for Spring inner beans http://t.co/BISeJbLt	Nicolas Frankel	@nicolas_frankel	Feb 10, 2013 7:20:29 PM	IFTTT
#JSoup #Java #HTML Parser http://t.co/2fRWrtI Used by #Vaadin but since 2 hours by Twaatin :-)	Nicolas Frankel	@nicolas_frankel	Feb 10, 2013 7:12:26 PM	Echofon
Some interesting tips: Top 10 Customization of #Eclipse Settings http://t.co/yRWcthvC	Nicolas Frankel	@nicolas_frankel	Feb 10, 2013 1:02:34 PM	Echofon
@cilemini :-/ Thanks for the answer, even it's the one I	Nicolas Frankel	@nicolas_frankel	Feb 8, 2013 5:24:29 PM	Echofon

Summary

In this chapter, we were made aware of two important parts of many Vaadin applications. The first part described the following three levels of data wrapping in Vaadin:

- **Property:** This represents a simple object such as a `String` or a `Date`. Properties can be tweaked in order to display the encapsulated data the way we really want.
- **Item:** This wraps around a single structured object. We used our `Person` class to show how it could be encapsulated in `Item`, and finally displayed in a form. Forms are a nice façade over a single item, whether in a read-only or read-write mode.
- **Container:** This is the topmost level and lets us wrap around a collection of items. Containers may have additional properties such as filterable, sortable, and hierarchical, each one bringing features to the container.

Then, we saw components that can connect to container datasources: tables and trees. Remember that where trees are hierarchical containers, tables are flat containers. However, they can be filterable and sortable at the same time.

As tables are such useful components, Vaadin provides many configuration features around them and we spent some time looking at those capabilities:

- Collapsing and ordering of columns, as well as headers and footers
- Sorting, both on the user-side and programmatically
- Table viewpoint and items caching to enhance the user experience
- Selection of table rows and editing of table data
- Finally, the drag-and-drop feature was a good entry point into the more general drag-and-drop API in Vaadin

At this point, we have seen all the basics of the framework. You should now be able to create simple applications from scratch. The next chapter details some advanced features of the framework, both coming out of the box and from third-party add-ons.



This chapter is one of the biggest in the book and conveys important information. Be sure to have mastered all that was described before going further (or be sure to bookmark it to reread it later).

7

Core Advanced Features

In this chapter we will go beyond simple features to tackle what will make our applications well thought out and professional. Other core features are out-of-the-box capabilities provided by the Vaadin framework and include:

- Accessing JavaEE objects from within the Vaadin framework
- Using the Navigation API, letting us bookmark application state
- Embedding Vaadin components in third-party applications
- Default error handling and overriding it
- Connecting components to databases
- Server push

Accessing the JavaEE API

In some cases, we will want to get a handle on the underlying request-response model. This could be motivated by the following requirements:

- The servlet context; for example, integrating with some third-party framework such as Spring requires data stored in the servlet context
- Managing cookies, either to read or write them, for example, to pass cookie-based authentication reverse-proxies
- The session context, to get data stored there by a legacy part of the application

For all these cases, we need access to the underlying Java EE Servlet API, which until this point was completely hidden. Basically, we need to be able to get a handle on `HttpServletRequest`, `HttpServletResponse`, and `HttpSession` objects.

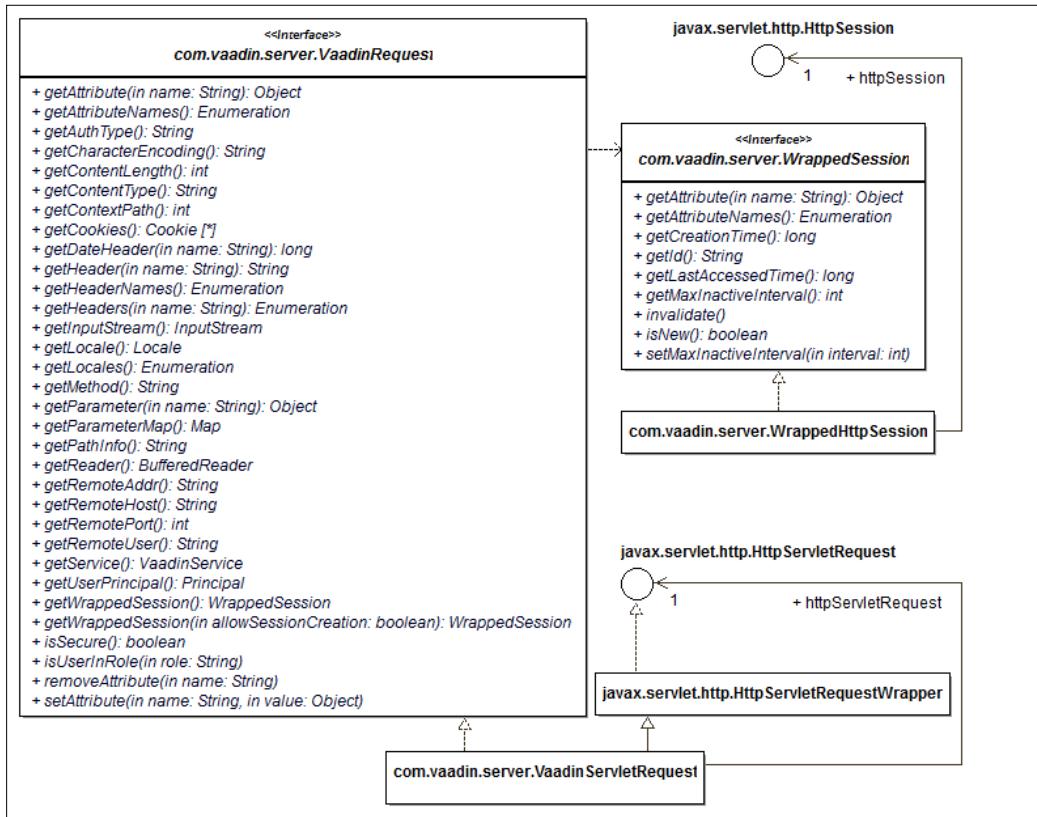
Servlet request

There are three entry points for getting a reference on the servlet request:

- The first is the `init (VaadinRequest)` UI method.
- The second is `com.vaadin.server.VaadinService.getCurrentRequest ()`, which returns a `com.vaadin.server.VaadinRequest`. The `com.vaadin.server.VaadinServletService.getCurrentServletRequest ()` method returning `javax.servlet.http.HttpServletRequest` is to be used in servlet containers.
- Finally, `com.vaadin.server.VaadinPortletService.getCurrentPortletRequest ()` method that returns `javax.portlet.PortletRequest` is to be used in portlet containers.

The last two methods respectively return a Servlet API object and a Portlet API one while the others return a Vaadin object. Why so? Because they are designed to run in both servlet and portlet containers. Therefore, the first two methods do not make assumptions in which environment the code will be deployed and those are the ones we should use if we plan to also provide agnostic code.

That is the reason why Vaadin provides a façade over both servlet and portlet requests in the form of the `VaadinRequest`. This interface offers a subset of methods found in both `HttpServletRequest` and `PortletRequest`. The following diagram offers an overview of the class hierarchy we are discussing:



In the preceding class hierarchy, the Vaadin implementation class tends to follow this pattern:

- It implements the Vaadin interface
- It encapsulates the corresponding Java EE Servlet API interface and offers a read-only accessor to it
- It extends the Java EE Servlet API wrapping class
- It delegates calls to the encapsulated interface



Most of the time, it is enough to get a handle on the native request object. It offers more than its lot of methods. It also let you run regardless of the container; you never know when a port of your Vaadin application to another container type will be necessary!

Servlet response

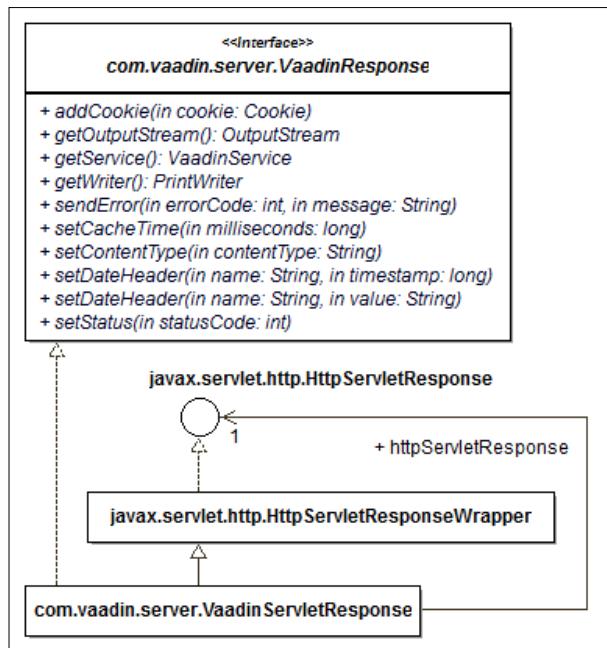
Servlet response is accessible through the following methods:

- The first way is `VaadinService.getCurrentResponse()`, returning a `VaadinResponse` object
- Then, `VaadinServletService.getCurrentResponse()` method that returns a `VaadinServletResponse` object
- The `VaadinPortletService.getCurrentResponse()` method returns a `VaadinPortletResponse` object



Please note the previous classes belong to the `com.vaadin.server` interface.

The following class diagram displays the big picture:



Vaadin shows there its design regularity: the response class hierarchy is similar to the preceding previous request.

With a reference on a `VaadinResponse` object, we can take many actions. Those include adding cookies, for example:

```
import java.util.Random;

import javax.servlet.http.Cookie;

import com.vaadin.server.VaadinRequest;
import com.vaadin.server.VaadinResponse;
import com.vaadin.server.VaadinService;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

@SuppressWarnings("serial")
public class VaadinResponseUI extends UI {

    @Override
    protected void init(VaadinRequest request) {

        Label label = new Label("Check cookie under 'random' key");

        Random generator = new Random();

        long random = generator.nextLong();

        Cookie cookie = new Cookie("random", String.valueOf(random));

        VaadinResponse response = VaadinService.getCurrentResponse();

        response.addCookie(cookie);

        setContent(label);
    }
}
```

Wrapped session

The final integration of Java EE Servlet API concerns `WrappedSession`. One can get a handle on it through the following methods:

- `findVaadinSession(VaadinRequest)` in `VaadinService`
- `getSession()` in `VaadinSession`

Vaadin session

 `VaadinSession` is completely unconnected to `WrappedSession`. The first relates exclusively to the Vaadin API itself while the latter is oriented towards the Servlet/Portlet API. We used Vaadin session to store the logged in Twattin when we used login/password authentication in *Chapter 5, Event Listener Model*.

As is the case for Vaadin request and response, wrapping session is a facade over both Servlet and Portlet API.

Navigation API

We discussed in *Chapter 1, Vaadin and its Context*, about Vaadin's approach opposite to traditional page flow paradigm, and how it is a good thing since applications are about screens and not pages. However, this single URL thing is precisely what prevents us from bookmarking individual pages.

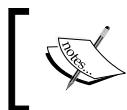
In some cases, however, this is a business requirement. Consider for example, the following use-cases:

- A large retail store application offers many items dispatched in the same screen. Such a screen displays details about items, and the vendor really wants customers to bookmark specific items. In fact, any such catalogue application would probably have the same requirements.
- In the same vein, a forum application could probably host many different subjects. It would be a real asset to this application to let users bookmark a specific subject or even a particular thread.

URL fragment

HTML provides a nice feature in the form of the URL fragment. Quoting the W3C:

Some URIs refer to a location within a resource. This kind of URI ends with "#" followed by an anchor identifier (called the fragment identifier).



Refer the following URL for more details:

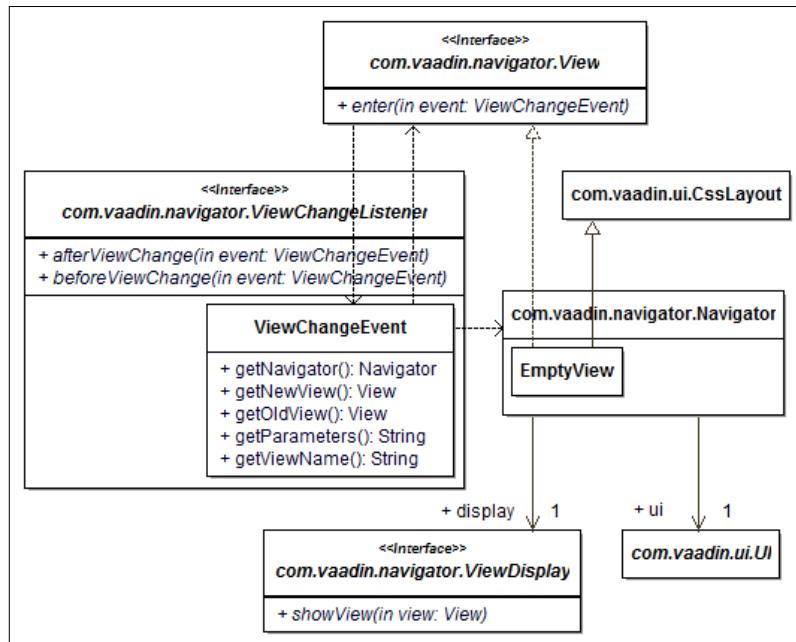
<http://www.w3.org/TR/html401/intro/intro.html#h-2.1.2>



Therefore, this lets us to stay on the same page, and yet reference different states, that can be read or written on the server side. It is up to the application developer to bridge between the string fragment and the whole state. Taking our previous use-cases as examples, the fragment can be an item's, a forum's, or a thread's ID. Let us implement this with Vaadin!

Views

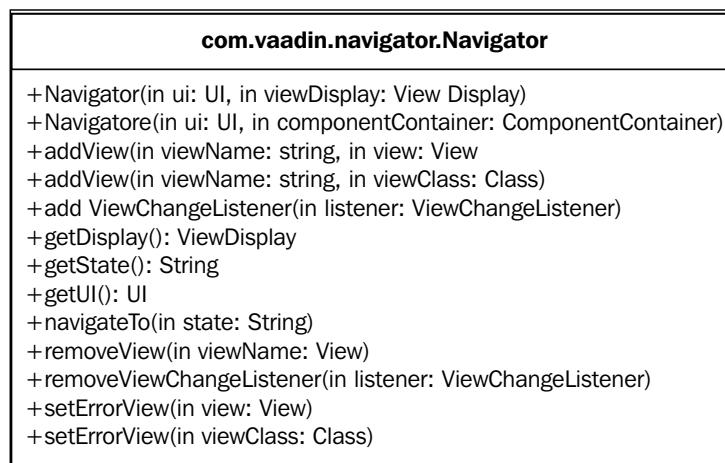
Views form the Navigator API root, as shown in the following diagram. Basically, a view is just a place to display a bag of components. A view provides a single method, `enter(ViewChangeEvent)` called when the view is displayed by a `Navigator` object. Though there is no requirement in the API, views are generally components.



Navigator

A navigator is a class that manages views switching. This is how it works: first, we register a view under a specific name and then we can tell the navigator to display the view. Each time a view is changed, it fires a view change event. Navigators offer a way to subscribe to those events.

Each navigator needs a UI and a place to display views, aptly named `ViewDisplay`. View displays have two implementation classes: `SingleComponentContainerViewDisplay` and `ComponentContainerViewDisplay`. The following diagram displays the methods of the `Navigator` class:



Methods to add views to the navigator either take a view instance or a view class. In the latter case, the navigator takes care of instantiating the view.

Let us create a very simple application as an illustration, where users can switch views. With the following code, clicking on a button displays a view registered at initialization:

```
@PreserveOnRefresh
public class ViewUI extends UI {

    private static final String[] VIEW_NAMES =
        { "AView", "AnotherView", "AThirdView" };

    @Override
    protected void init(VaadinRequest request) {

        Layout viewPlaceholder = new VerticalLayout();
```

```
ViewDisplay display =
    new ComponentContainerViewDisplay(viewPlaceholder);

final Navigator navigator = new Navigator(this, display);

navigator.addView("", new LabelView("Void"));

BeanItemContainer<String> container =
    new BeanItemContainer<String>(String.class);

for (String viewName : VIEW_NAMES) {

    container.addBean(viewName);

    LabelView newView = new LabelView(viewName);

    navigator.addView(viewName, newView);
}

ComboBox combo = new ComboBox("View name", container);

combo.setImmediate(true);

combo.addValueChangeListener(new ValueChangeListener() {

    @Override
    public void valueChange(ValueChangeEvent event) {

        String viewName =
            (String) event.getProperty().getValue();

        navigator.navigateTo(viewName);
    }
});

FormLayout layout = new FormLayout(viewPlaceholder, combo);

layout.setMargin(true);

setContent(layout);
}

private class LabelView extends Label implements View {
```

```
public LabelView(String label) { super(label); }

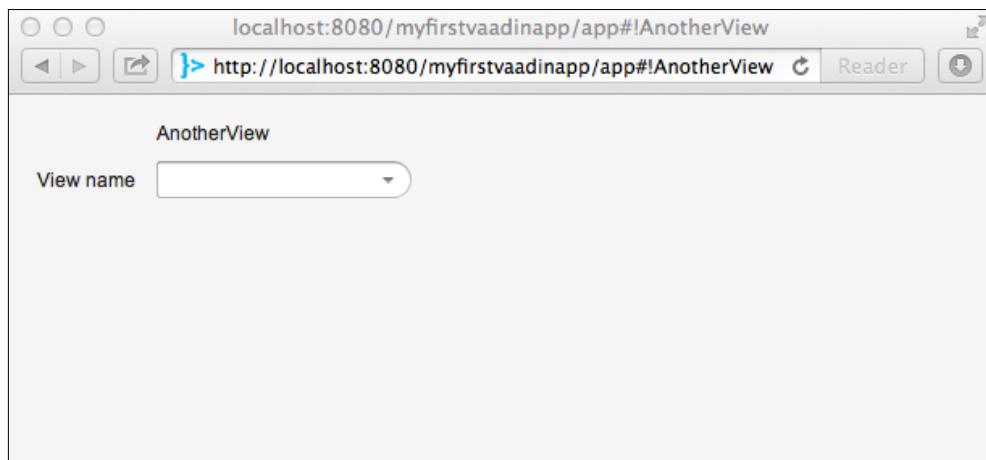
@Override
public void enter(ViewChangeEvent event) {}
}

}
```

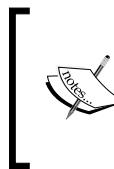
The UI has only two components: a placeholder for the view and a combo-box to hold view names.

When the combo-box changes value, we just tell the navigator to navigate to the view name and the navigator takes care to display it in the placeholder location.

This code produces the following display:



From a user point of view, nothing exceptional happens. We could easily have obtained the same result without views. Yet, notice how the URL appends the view name. Even better, we can navigate to a view by directly typing the URL appended with #! and the view name!



Of course, more realistic examples would load data corresponding to the entity. Using the entity's ID directly is good enough for catalogue applications. Depending on the required level of security, it may be a bad idea to expose it; in this case, symmetric encoding may be the solution.

Initial view

Notice how we first registered a view under the empty string:

```
navigator.addView("", new LabelView("Void"));
```

If this is not done, Vaadin will loudly complain with a **java.lang.IllegalArgumentException: Trying to navigate to an unknown state "and an error view provider not present** error.

Registering an initial view under the empty string is thus mandatory.

Error view

Users may likely type invalid URL fragments and provoke errors on their own. In order to prevent throwing raw stack traces to their face, it is well advised to create a specific error view and register it in the navigator by using:

```
navigator.setErrorView(myErrorView);
```

Dynamic view providers

In the previous example, views were static and were added during UI initialization. Sometimes, this cannot be done. For example, what if the views had to display n labels where n comes from user input? n is not bound so we cannot create views at initialization; we definitely have to take another route.

This route is to delegate to an interface called view provider. View providers know how to:

- Translate from view name and optional parameters to raw view name. Of course, if there are no parameters, this is a no-brainer. It is the responsibility of `getViewName(String)`.
- Get a view object from a view name with the help of the `getView(String)` method. Using the same view over and over or creating a new view each time the method is called with the same key is dependent on the implementation.

Then, a view provider can be added to and respectively removed from the navigator with `addProvider(ViewProvider)` and `removeProvider(ViewProvider)`.

Given these, implementation of our n labels problem is easily achieved:

```
@PreserveOnRefresh
public class ViewProviderUI extends UI {

    @Override
    protected void init(VaadinRequest request) {

        Layout viewPlaceholder = new VerticalLayout();

        ViewDisplay display =
            new ComponentContainerViewDisplay(viewPlaceholder);

        final Navigator navigator = new Navigator(this, display);

        navigator.addProvider(new NumberViewProvider());

        HorizontalLayout layout =
            new HorizontalLayout(viewPlaceholder);

        setContent(layout);
    }

    private class NumberViewProvider implements ViewProvider {

        @Override
        public String getViewName(String viewAndParameters) {

            int indexQuestionMark = viewAndParameters.indexOf('?');
            int indexSlash = viewAndParameters.indexOf('/');

            int index = Math.max(indexQuestionMark, indexSlash);

            return index == -1 ? viewAndParameters :
                viewAndParameters.substring(0, index);
        }

        @Override
        public View getView(String viewName) {

            GridView gridView = new GridView();

            if (!"".equals(viewName.trim())) {
```

```

        int number = Integer.parseInt(viewName);

        for (int i = 0; i < number; i++) {

            Label label = new Label(String.valueOf(i));

            gridView.addComponent(label);
            gridView.setComponentAlignment(label, MIDDLE_RIGHT);
        }
    }

    return gridView;
}
}

private class GridView extends GridLayout implements View {

    public GridView() {

        setMargin(true);
        setSpacing(true);
        setColumns(5);
    }

    @Override
    public void enter(ViewChangeEvent event) {}
}
}

```

Now we can append #! and a positive number to the URL's end: the application will get the view name from the URL and it creates a brand new view from the name, adding as many labels as specified.

This is achieved by using adding a view provider that parse the view name and creates a view with so many labels. Of course, we have to account for the fact that at application launch, there is a view name.

In truth, the same result could have been attained by inheriting from Navigator and overriding the `navigateTo(String)` method. But remember the golden principle from object-oriented programming:

Favor composition over inheritance.

View provider is the way composition is addressed regarding this issue in Vaadin.



Point is: for static views, create them and simply register them.
For dynamic views, use a (or many) view provider(s).



Event model around the Navigation API

The Navigation API brings a standard event model based on view changes; events are generated when views are switched either programmatically or by the user. View change listeners are provided so that we are able to take action in this case for example, switches may be vetoed.

The event/listener pair follows the standard Observer pattern seen in *Chapter 5, Event Listener Model*; we will use it at the end to enhance Twaattin.

Final word on the Navigator API

The good news about the Navigator API is that it uses Google standards to make **Single Page Interface (SPI)** applications crawlable by search engines. In standard websites, a single URI maps to specific content. In SPI, the application URI only maps to many contents, as it is reloaded through asynchronous requests, AJAX in one word.

Google provides the way to crawl and index AJAX applications with the use of hashbang, the #! used in the Navigator API. By using the latter, you automatically make relevant application pages appear in Google search results. Isn't life sweet?



Developers interested in going down this road should probably get acquainted with **Search Engine Optimization (SEO)** first.



Embedding Vaadin

In our previous examples, the whole application was Vaadin-based. This may not be desirable for a variety of reasons:

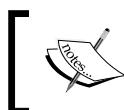
- Legacy applications may have to be upgraded one part at a time, thus having one part managed by the old application framework (if any) and the other part by the Vaadin framework
- Even though Vaadin-based applications can easily be integrated with other frameworks, some may be unsuitable for such integration, thus creating the need for embedding either the Vaadin or the other application

- Finally, we may need to display more than one Vaadin application at the same time in the same HTML page without having to go as far as installing a portal

In addition, sometimes there is a need to use a single Vaadin component as a part of a larger, static web page where the content is better made with something else than Vaadin. That is why it is great that it is so easy to embed Vaadin.

Basic embedding

At the simplest level, Vaadin can be embedded using a simple HTML `iframe` tag. In this case, the embedding page and the iframe Vaadin servlet have to come from the same domain.



More information on iframes can be found at the following URL:

<http://www.w3.org/TR/html4/present/frames.html#h-16.5>



In this case, we need a dedicated servlet mapping in the web application deployment descriptor to serve Vaadin and use it as the `iframe src` attribute. The servlet mapping would look something akin to the following:

```
<servlet-mapping>
  <servlet-name>VaadinServlet</servlet-name>
  <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

On the HTML page, the `iframe` is referenced as follows:

```
<iframe src="/app/">[Cannot display Vaadin]</iframe>
```

This technique is very easy to use and requires no specific knowledge at all. Nonetheless, using iframes brings some huge disadvantages, including:

- Making the bookmarking feature discussed earlier impossible.
- Increasing complexity of page-iframe communications.
- Decreasing accessibility of pages as screen readers have two different structures to analyze. Depending on the specific reader, it may make the application completely unusable for vision-impaired users.
- Finally, iframes represent a security hole in that they are susceptible to **Cross-Site Scripting**. More information on security in general and issues related to iframes can be found on the OWASP website: <https://www.owasp.org/>.

Nominal embedding

Ever wondered how a Vaadin application is kick started after calling a URL?

The first request to the Vaadin servlet downloads a page with only a few data, but including GWT bootstrap code. Subsequent requests are then only client-sided AJAX ones. Nominal embedding only replicates this process.

This approach embeds Vaadin directly in a `div` tag on the desired page, without the need for further artifacts. However, whereas Vaadin took care of providing loading of necessary components and initializing them, it is now the developer's responsibility.

Page headers

In order to enable Vaadin 7 applications to run on Internet Explorer 6 or 7, the Google Chrome Frame plugin: a way of running Chrome inside IE, has to be enabled on the user's computer.



For more information on Google Chrome Frame, please see
<https://developers.google.com/chrome/chrome-frame/>.



Then, encoding must be set to UTF-8. This is achieved through HTML page headers:

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=9;chrome=1" />
</head>
```

The div proper

The placeholder `div` only has to have the `v-app` CSS class attribute and a unique `id` attribute.

In order to provide the best user-experience, you may also want to:

- Provide a loading indicator to inform the user that application resources are being loaded. This is done with adding an embedded `div` with CSS class `v-app-loading`.
- Add alternative text if the user has deactivated JavaScript (which to be frank should be seldom seen nowadays).

The final result would look something like the following:

```
<div id="embedded-app" class="v-app">
    <div class="v-app-loading"></div>
    <noscript>You should enable javascript to use this
    app</noscript>
</div>
```

The bootstrap script

A requirement of embedding Vaadin in a div tag is to call the JavaScript initialization script. It is called `vaadinBootstrap.js` and is available under the `/VAADIN` subcontext, as all other Vaadin static resources. Calling this script must be made before UI initialization.

```
<script type="text/javascript" src="/VAADIN/vaadinBootstrap.js"></
script>
```

By using GWT client-side, we also inherit from GWT requirements. One of such requirement is the GWT history `iframe` tag, needed to support URL fragments (as seen in the previous section named *URL fragment*).

```
<iframe tabindex="-1" id="__gwt_historyFrame" src="javascript:false"
style="position: absolute; width: 0; height: 0; border: 0;
overflow: hidden"></iframe>
```

UI initialization call

The final step in the embedding process is to call the method that will initialize the UI. It is only possible if the previous bootstrap script has been referenced properly and is finished loading. To ensure this is the case, try getting a reference on the `window.vaadin` object.

The method itself is `vaadin.initApplication()` and it expects two parameters, the first being the `id` of the aforementioned placeholder `div`, the second a full-fledged JSON data structure:

```
{
  "browserDetailsUrl" : string,
  "widgetset" : string,
  "theme" : string,
  "versionInfo" : {
    "vaadinVersion" : string
    "applicationVersion" : string
  },
}
```

```
"vaadinDir" : string,
"heartbeatInterval" : int,
"debug" : boolean,
"standalone" : boolean,
"authErrMsg" : {
    "message" : string,
    "caption" : string
},
"comErrMsg" : {
    "message" : string,
    "caption" : string
},
"sessExpMsg" : {
    "message" : string,
    "caption" : string
}
}
```

Here is the description of the mandatory attributes:

- `browserDetailsUrl`: The Vaadin servlet's context root, configured in the web deployment descriptor.
- `widgetset`: Fully qualified widgetset (see *Chapter 8, Featured Add-ons*, for detailed information on widgetsets). If no custom widgetset is developed, use `com.vaadin.DefaultWidgetSet`.
- `theme`: Specifies the theme name. Pick a theme offered by Vaadin out-of-the-box (`liferay`, `chameleon`, `reindeer`, or `runo`) or choose your own custom theme name if you installed one.
- `vaadinDir`: The subcontext for serving Vaadin static resources (such as widgetsets and themes). Has to be set to `VAADIN/`.
- `heartbeatInterval`: The interval between heartbeats in seconds. Heartbeats are asynchronous requests sent to the Vaadin application. In the absence of such signals, the servlet cleans the user session to reclaim memory space.
- `vaadinVersion`: Specifies the Vaadin version, in major, minor and bugfix dotted notation (for example, `7.0.0`).

Additional optional attributes include `authErrMsg`, `comErrMsg` and `sessExpMsg`, each being composed of a caption and a detailed text. The latter can use `<u>` start and `</u>` end tags to create a link pointing to the application root. Defaults should be enough in most cases.

The following is an example of a working HTML 5 page:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=9;chrome=1" />
<title>Embedding page</title>
</head>
<body class="v-generated-body">
<div id="embedded-app" class="v-app"
      style="height:200px;width:400px"></div>
<script type="text/javascript" src="VAADIN/vaadinBootstrap.js">
</script>
<script type="text/javascript">//<![CDATA[
  if (!window.vaadin) {
    alert("Failed to load the bootstrap JavaScript");
  }
  vaadin.initializeApp("embedding", {
    "browserDetailsUrl": "app",
    "theme": "reindeer",
    "versionInfo": {
      "vaadinVersion": "7.0.0"
    },
    "widgetset": "com.vaadin.DefaultWidgetSet",
    "vaadinDir": "VAADIN/",
    "heartbeatInterval": 300
  });
}<![CDATA]]>
</script>
</body>
</html>
```

An important limitation of this solution is that it does not allow for embedding applications hosted on domains different from the page since browser security models tend to view AJAX requests made across third-party domains as security breaches.

Real-world error handling

Up to this point, we managed checked exceptions by wrapping them under runtime exceptions and re-throwing those. This way of handling error produces the following output:



A screenshot of a terminal window showing a Java stack trace. The stack trace is enclosed in a light orange box. It starts with a warning message: "com.vaadin.server.ServerRpcManager\$RpcInvocationException: Unable to invoke method click in". Below this, the full stack trace is shown, starting from "com.vaadin.shared.ui.button.ButtonServerRpc at" and ending at "at javax.servlet.http.HttpServlet.service(HttpServlet.java:728) at". The terminal window has a dark background with light-colored text.

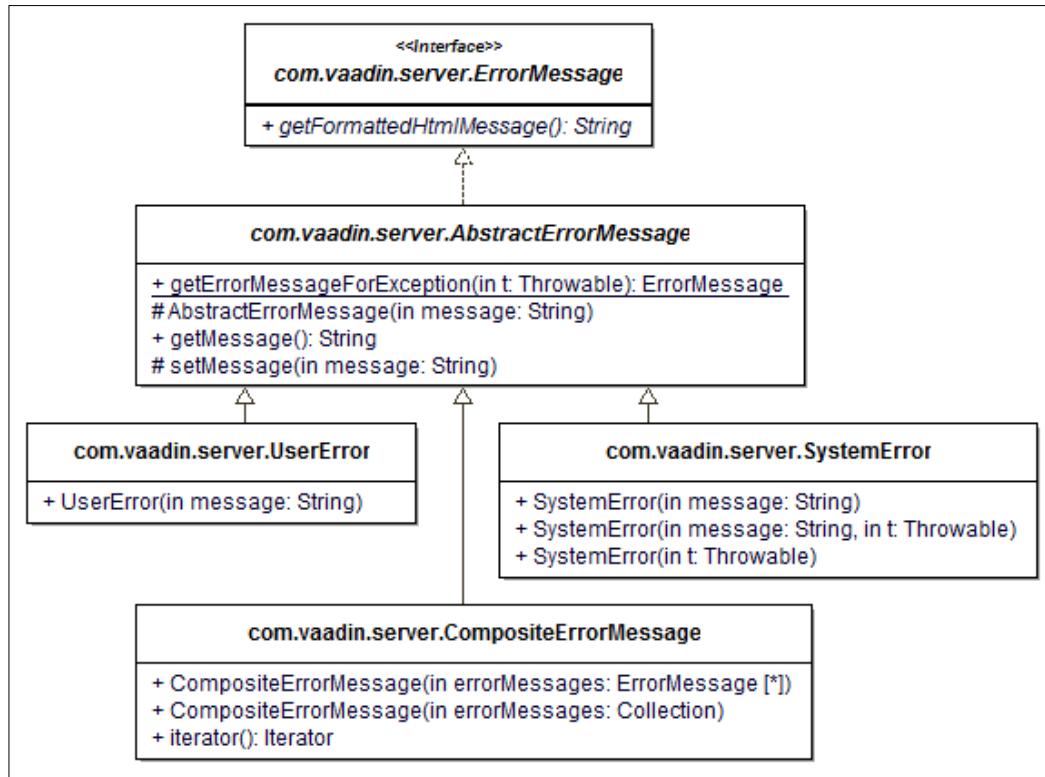
```
com.vaadin.server.ServerRpcManager$RpcInvocationException:  
Unable to invoke method click in  
com.vaadin.shared.ui.button.ButtonServerRpc at  
com.vaadin.server.ServerRpcManager.applyInvocation(ServerRpcMar  
at  
com.vaadin.server.ServerRpcManager.applyInvocation(ServerRpcMar  
at  
com.vaadin.server.AbstractCommunicationManager.handleBurst(Abs  
at  
com.vaadin.server.AbstractCommunicationManager.handleVariables(/  
at  
com.vaadin.server.AbstractCommunicationManager.handleUidlRequest  
at com.vaadin.server.VaadinServlet.service(VaadinServlet.java:315)  
at com.vaadin.server.VaadinServlet.service(VaadinServlet.java:201)  
at javax.servlet.http.HttpServlet.service(HttpServlet.java:728) at
```

The error messages

A simple hierarchy of error messages is available out-of-the-box in Vaadin:

- User errors represent errors provoked by the user. They are expected during the course of the application, and are intended as a guide to the user.
- System errors, on the other hand, are unexpected errors. They inherit from `RuntimeException` in order to convey contextual information about these abnormal conditions.
- Finally, `CompositeErrorMessage` allows us to encapsulate one or more error messages should the need arise.

The hierarchy of error messages is as shown in the following diagram:



Component error handling

Once the right type of error message has been chosen, it is easy as pie to set the error message on the desired component through the `setComponentError(ErrorMessage)` method: Vaadin takes care of displaying the small error icon.

In order to remove it, just call `setComponentError(null)` on the component.

The following code presents a combo box to let use set an error flag on the desired text field. Then, setting the focus on the text field removes it.

```
import java.util.ArrayList;
import java.util.Collection;

import com.vaadin.data.Property.ValueChangeEvent;
import com.vaadin.data.Property.ValueChangeListener;
import com.vaadin.event.FieldEvents.FocusEvent;
import com.vaadin.event.FieldEvents.FocusListener;
import com.vaadin.server.UserError;
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.ComboBox;
import com.vaadin.ui.FormLayout;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.TextField;
import com.vaadin.ui.UI;

@SuppressWarnings("serial")
public class ComponentErrorUI extends UI {

    @Override
    protected void init(VaadinRequest request) {

        FormLayout layout = new FormLayout();

        layout.setMargin(true);

        Collection<Integer> values = new ArrayList<Integer>();

        final TextField[] fields = new TextField[5];

        for (int i = 0; i < fields.length; i++) {

            final TextField textField =
                new TextField(String.valueOf(i));

            textField.addFocusListener(new FocusListener() {
```

```
    @Override
    public void focus(FocusEvent event) {

        textField.setComponentError(null);
    }
});

layout.addComponent(textField);

values.add(i);

fields[i] = textField;
}

ComboBox combo = new ComboBox("Choose component", values);

combo.setImmediate(true);
combo.setNewItemAllowed(false);

combo.addValueChangeListener(new ValueChangeListener() {

    @Override
    public void valueChange(ValueChangeEvent event) {

        Integer i = (Integer) event.getProperty().getValue();

        if (i != null) {

            fields[i].setComponentError(
                new UserError("An error"));
        }
    }
};

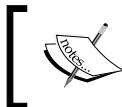
setContent(new HorizontalLayout(layout, combo));
}
}
```

This way, we can choose the component the error is displayed on, but it may not be enough when an exception occurs from within the code's depths. Fortunately, Vaadin provides a more general error handling solution.

General error handling

The root component in the exception handling chain, the one tasked to manage behavior, is the servlet as in most other web applications.

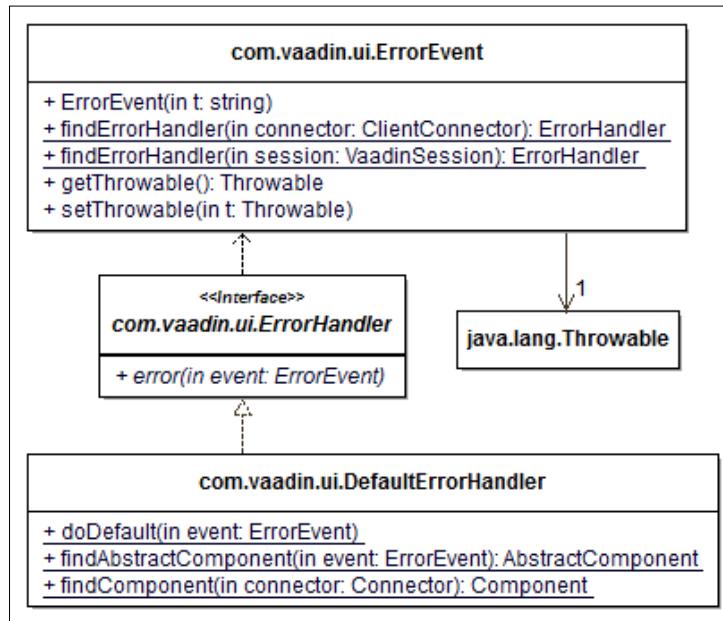
The servlet delegates error handling to an `ErrorHandler` instance, which is a `DefaultErrorHandler` concrete class by default. Default error handler is what displays the red warning icon and the associated stack trace.



Note that error handler is located in the Vaadin session so that error handling can be customized per user, if the need be, and the servlet queries the session to provide the error handler.



The following diagram shows the collaborating classes in the default exception handling mechanism:



Making use of our newfound knowledge of Vaadin error handling, we can design a solution to emulate the way of displaying errors in classical web applications: reserve a location at the screen's top and show the error message there. Sometimes, a button also let us consult additional details. This is exactly what the following code does:

```

import java.io.PrintWriter;
import java.io.StringWriter;
  
```

```
import com.vaadin.data.Property.ValueChangeEvent;
import com.vaadin.data.Property.ValueChangeListener;
import com.vaadin.server.DefaultErrorHandler;
import com.vaadin.server.ErrorHandler;
import com.vaadin.server.VaadinRequest;
import com.vaadin.server.VaadinSession;
import com.vaadin.ui.Button;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.Button.ClickListener;
import com.vaadin.ui.CheckBox;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Label;
import com.vaadin.ui.TextArea;
import com.vaadin.ui.UI;
import com.vaadin.ui.VerticalLayout;

@SuppressWarnings("serial")
public class ErrorHandlingUI extends UI {

    @Override
    protected void init(VaadinRequest request) {

        final ErrorBar errorBar = new ErrorBar();

        Button button = new Button("Throw error");

        button.addClickListener(new ClickListener() {

            @Override
            public void buttonClick(ClickEvent event) {

                throw new RuntimeException("A bad thing happened");
            }
        });

        CheckBox check = new CheckBox("Set custom error handler");

        check.addValueChangeListener(new ValueChangeListener() {

            @Override
            public void valueChange(ValueChangeEvent event) {

                boolean checked =

```

```
(Boolean) event.getProperty().getValue();

VaadinSession session = VaadinSession.getCurrent();

ErrorHandler handler = checked
    ? new CustomErrorHandler(errorBar)
    : new DefaultErrorHandler();

errorBar.setVisible(false);

session.setErrorHandler(handler);
}

);

VerticalLayout layout =
new VerticalLayout(errorBar, check, button);

layout.setMargin(true);
layout.setSpacing(true);

setContent(layout);
}
```

The custom error handler is the "meat" of the code: it is where the error managing code is implemented. In our case, it displays the stack into an error bar custom component.

```
private class CustomErrorHandler implements ErrorHandler {

    private final ErrorBar errorBar;

    public CustomErrorHandler(ErrorBar errorBar) {

        this.errorBar = errorBar;
    }

    @Override
    public void error(com.vaadin.server.ErrorEvent event) {

        Throwable throwable = event.getThrowable();

        errorBar.displayError(throwable);
    }
}
```

The error bar is a component aggregating multiple other components. Its responsibility is to display a throwable stack trace.

```
private class ErrorBar extends VerticalLayout {  
  
    private final Label staticLabel =  
        new Label("An error occurred:");  
    private final Label errorLabel = new Label();  
    private final TextArea stackTextArea = new TextArea();  
    private final Button collapseButton =  
        new Button("Show details");  
  
    public ErrorBar() {  
  
        HorizontalLayout upperBar =  
            new HorizontalLayout(staticLabel, errorLabel,  
                collapseButton);  
  
        upperBar.setSpacing(true);  
  
        addComponent(upperBar);  
        addComponent(stackTextArea);  
  
        stackTextArea.setVisible(false);  
        stackTextArea.setWidth("100%");  
  
        addStyleName("error");  
  
        setVisible(false);  
  
        collapseButton.addClickListener(new ClickListener() {  
  
            @Override  
            public void buttonClick(ClickEvent event) {  
  
                boolean visible = stackTextArea.isVisible();  
  
                stackTextArea.setVisible(!visible);  
                collapseButton.setCaption(visible  
                    ? "Show details" : "Hide details");  
            }  
        });  
    }  
}
```

```
public void displayError(Throwable throwable) {  
  
    errorLabel.setValue(throwable.getMessage());  
  
    StringWriter stringWriter = new StringWriter();  
  
    PrintWriter printWriter = new PrintWriter(stringWriter);  
  
    throwable.printStackTrace(printWriter);  
  
    stackTextArea.setReadOnly(false);  
    stackTextArea.setValue(  
        stringWriter.getBuffer().toString());  
    stackTextArea.setRows(throwable.getStackTrace().length);  
    stackTextArea.setReadOnly(true);  
  
    setVisible(true);  
}  
}  
}
```

Play with the application created by the preceding code (or variations of it). Notice how error handler is tied to the Vaadin session by opening two different browser windows: they can manage different error handlers, because they are seen as different clients by the server (and thus generate two separate session spaces).

Key points include:

- Creation of a custom error component to allow for the display of exceptions
- Laying out an instance of error component on the screen
- Creation of an error handler concrete class to bind received exceptions to said instance

SQL container

In *Chapter 6, Containers and Related Components*, we looked at the table component but we left how to fill the container with data aside. If we were in a standard layered architecture, we would explicitly call the service layer in order to call the persistence layer which would itself query the database.

If our requirements are limited to CRUD operations, this is overkill. In order to manage this, Vaadin provides a SQL container implementation able to directly connect to a SQL database using JDBC.

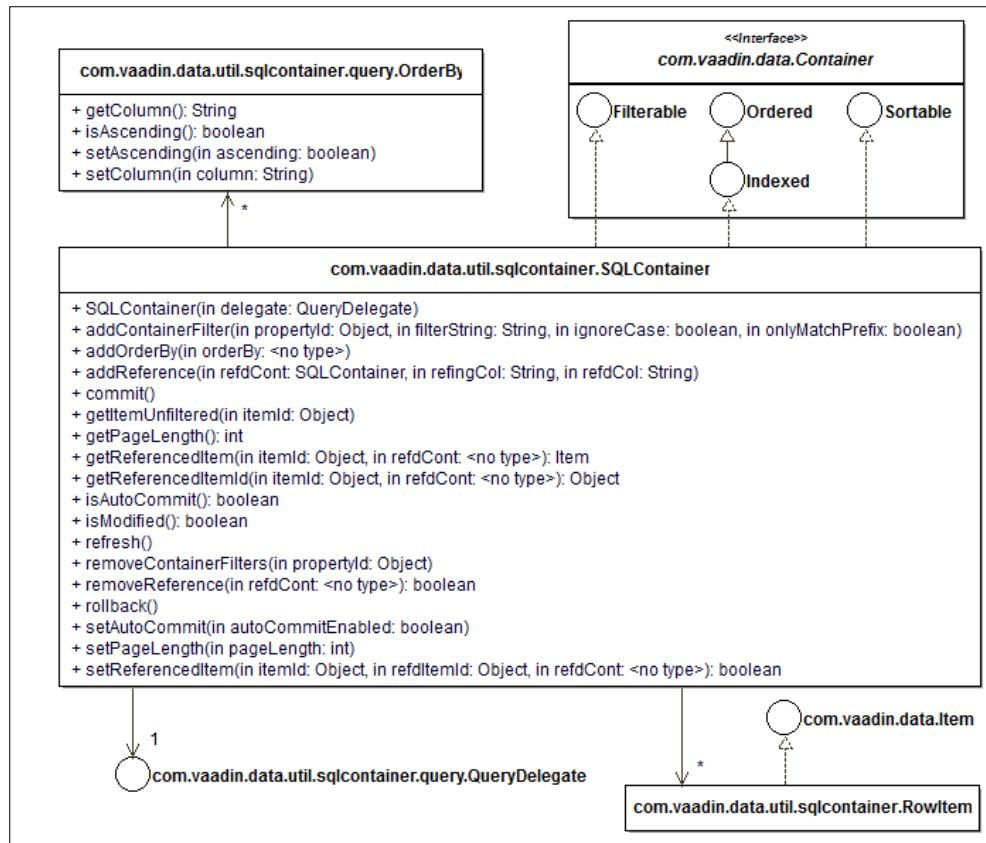


The add-ons directory provides some additional database-oriented containers, each using a specific API (JPA and Hibernate).



Architecture

Vaadin SQL Container is built around the `SQLContainer` class, which has all the nice properties we expect from a container: it is indexed, sortable, and filterable.



Items in the container are specialized instances, namely `RowItem`. This type is not exposed and should not be manipulated directly by the developer, even though it is part of the API.

Features

Features of SQL container include:

- **Transaction management:** Changes made to the container can be either committed or rollbacked to the data tier. Alternatively, we can set each operation to the container to be committed automatically. Note that the default behavior doesn't use autocommit.
- **Programmatic filtering:** Uses Vaadin's filter API already seen in the *Filter* section of *Chapter 6, Containers and Related Components*.
- **Programmatic ordering:** Ordering is as simple as calling `addOrderBy(OrderBy)` on `OrderBy` instances.
- **Initialization:** The container can be initialized with data from the underlying data tier with the `refresh()` method.
- **Paging:** The container uses a page length attribute in order to optimize performance. As in tables (see the *Viewpoint* section in *Chapter 6, Containers and Related Components*), SQL containers use a page length, as well as a cache ratio. Unlike table, cache ratio is set to 2 and cannot be changed, at least not without serious hacking. This means that requests made through query delegate (see the diagram in the next section) limit the number of occurrences to two times the page length.

Queries and connections

SQL container's responsibilities are those of a container. Real interaction with the database is delegated to a `QueryDelegate` instance, which in turn delegates connection management to a `JDBCConnectionPool` instance.

JDBC connection pool can either wrap a direct connection to the database, driver manager style, or a data source retrieved from the application server, depending on the concrete class type. Note that in the former case, the class creates a **pseudo-shareable** pool.

For more information on driver manager, visit the following URL:

<http://download.oracle.com/javase/6/docs/api/java/sql/DriverManager.html>

For more information on data source, visit the following URL:

<http://download.oracle.com/javase/6/docs/api/javax/sql/DataSource.html>

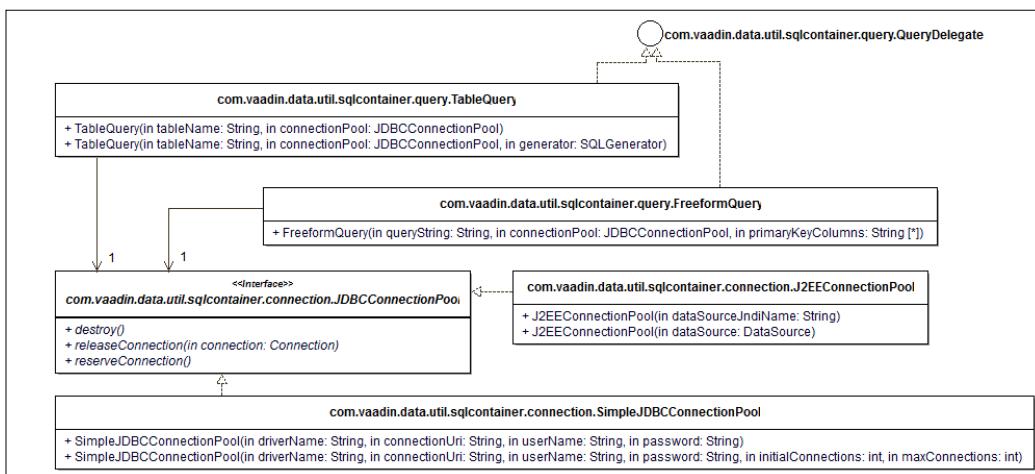


Query delegates come in two flavors:

- For simple table display, just use a `TableQuery`, passing the table's name and the JDBC connection pool to use. This kind of delegate will retrieve all occurrences from the table. The only thing left to do is to customize the appearance of the table as was done in *Chapter 6, Containers and Related Components*.
- Alternatively, when we need to go beyond this, we can use `FreeformQuery`. It allows us to pass the query, as well as the JDBC connection pool and optionally all primary key columns.

Note that without the last parameter, only `SELECT` orders can be executed, thus rendering the wrapping table read-only.

In both cases, programmatic order bys and filters can be added in order to further refine the results.



Note that table query is relatively straightforward, and does not require much effort in order to create an editable table widget backed by a SQL database table.



NoSQL backends are currently not compatible with SQL container, which would be strange, anyhow.

Database compatibility

As table query delegates much of the SQL generation to a dedicated `SQLGenerator` class, using another RDBMS is just a matter of creating the right implementation (inheriting from `DefaultSQLGenerator` seems like a good starting point). By default, Vaadin provides:

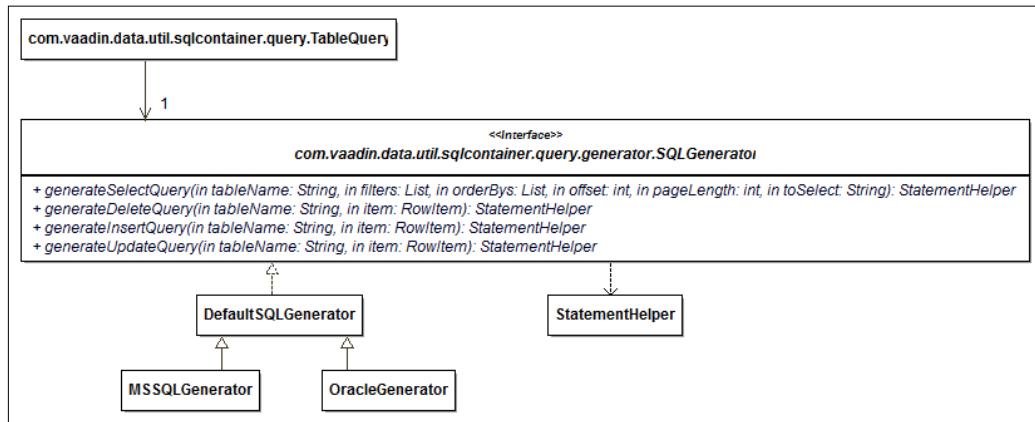
- A default implementation compatible with Hypersonic SQL, MySQL, and PostgreSQL databases
- A specific Microsoft SQL implementation
- And a dedicated Oracle implementation



The default implementation of `SQLContainer` doesn't work with IBM DB2 as it does not support `LIMIT/OFFSET` per default. It can be explicitly set, but when working with existing tables that cannot be changed, one will have to update some implementation.



All these classes and interfaces are located in the `com.vaadin.data.util.sqlcontainer.query.generator` package.



Note that `SQLGenerator` represents an abstraction over the RDBMS product whereas `StatementHelper`, as its name implies, is a helper whose responsibility is to help creating SQL statements and should generally not be used directly.



Armed with our newfound knowledge, it is easy to create a generic UI meant to display any data table (with a Primary Key constraint) within an editable table widget, along with two global commit/rollback buttons. Each row also shows a delete button as generated columns (see *Chapter 6, Containers and Related Components*, for a refresher on generated columns):

```
@SuppressWarnings("serial")
public class DatabaseTableScreen extends VerticalLayout {

    private SQLContainer container;
    private Table table;

    public DatabaseTableScreen() {

        setMargin(true);

        table = new Table();

        table.setPageLength(10);
        table.setEditable(true);
        table.setSizeFull();

        table.addGeneratedColumn("", 
            new RemoveItemColumnGenerator());

        HorizontalLayout buttonBar = new HorizontalLayout();

        buttonBar.setMargin(true);
        buttonBar.setSpacing(true);

        Button commit = new Button("Commit");
        commit.addClickListener(new ClickListener() {

            @Override
            public void buttonClick(ClickEvent event) {

                try {
                    container.commit();

                    Notification.show("Changes committed");

                } catch (SQLException e) {

```

```
        Notification.show("Unable to commit",
                           ERROR_MESSAGE);
    }
}
});
buttonBar.addComponent(commit);

Button rollback = new Button("Rollback");
rollback.addActionListener(new ClickListener() {

    @Override
    public void buttonClick(ClickEvent event) {

        try {
            container.rollback();

            Notification.show("Changes rollbacked");

        } catch (SQLException e) {

            Notification.show(
                "Unable to rollback", ERROR_MESSAGE);
        }
    }
});
buttonBar.addComponent(rollback);

addComponent(table);
addComponent(buttonBar);
}

public void populate(String tableName,
                     JDBCConnectionPool connectionPool) {

    QueryDelegate query =
        new TableQuery(tableName, connectionPool);

    try {

        container = new SQLContainer(query);

        table.setContainerDataSource(container);
    }
}
```

```

    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

The following column generator can be reused to provide a generic remove item feature in Vaadin tables:

```

public class RemoveItemColumnGenerator
    implements ColumnGenerator {

    @Override
    public Component generateCell(Table source,
        Object itemId, Object columnId) {

        Button button = new Button("Delete");

        button.setData(itemId);

        button.addClickListener(new ClickListener() {

            @Override
            public void buttonClick(ClickEvent event) {

                Object itemId = event.getButton().getData();

                container.removeItem(itemId);
            }
        });
    }

    return button;
}
}

```



Storing arbitrary data

The two highlighted lines are of particular importance: the first lets us store the item's Primary Key in the button while the second enable us to get it. Both `setData()` and `getData()` methods are available on `AbstractComponent`.

In order to use this window, we just have to provide a JDBC pool instance, for example, by tasking the UI to create it during initialization as follows:

```
package com.packt.learnvaadin.container;

import java.sql.SQLException;

import com.vaadin.data.util.sqlcontainer.connection.JDBCConnectionPool;
import com.vaadin.data.util.sqlcontainer.connection.SimpleJDBCConnectionPool;
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.UI;

@SuppressWarnings("serial")
public class DatabaseTableUI extends UI {

    @Override
    protected void init(VaadinRequest request) {

        DatabaseTableScreen screen = new DatabaseTableScreen();

        try {

            JDBCConnectionPool connectionPool =
                new SimpleJDBCConnectionPool(
                    "org.h2.Driver", "jdbc:h2:~/learnvaadin",
                    "SA", "");

            screen.populate("PERSON", connectionPool);

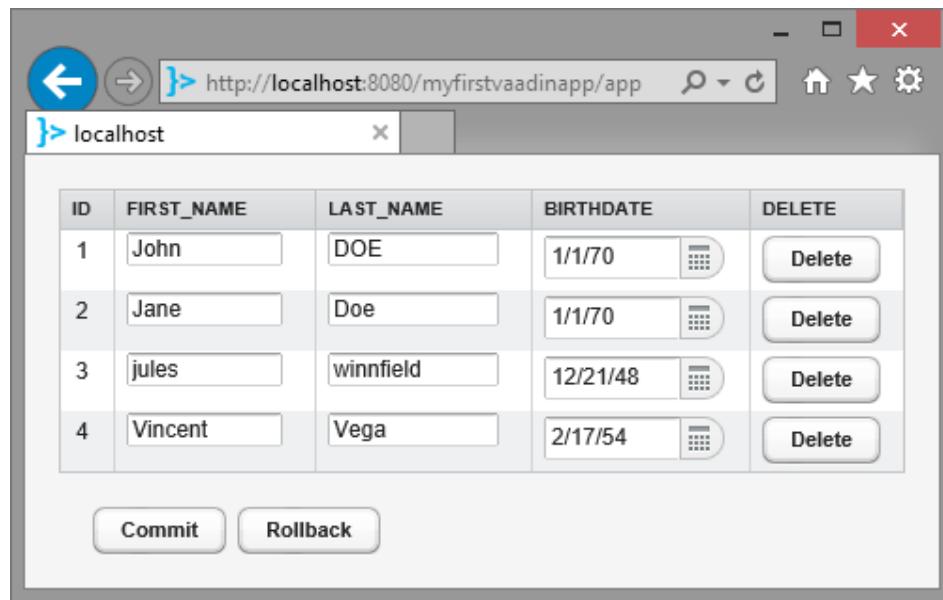
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

        setContent(screen);
    }
}
```

As an example, let us use a table that maps our `Person` entity from *Chapter 6, Containers and Related Components*:

PERSON			
PK	ID	LONG	VARCHAR
	FIRST_NAME	VARCHAR	VARCHAR
	LAST_NAME		
	BIRTHDATE	DATE	

If we use the database component on the previous table, we get the following display:



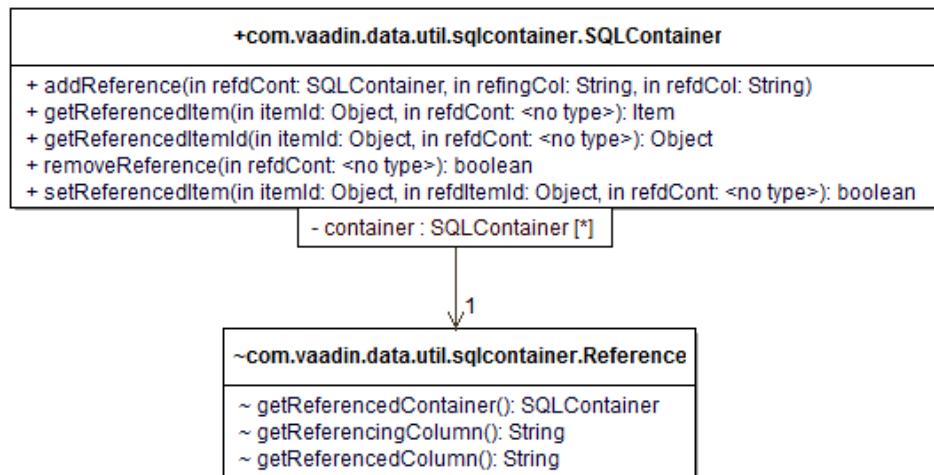
Vaadin automatically detects that the `ID` column is the Primary Key and displays it as a label (instead of a field).

Joins

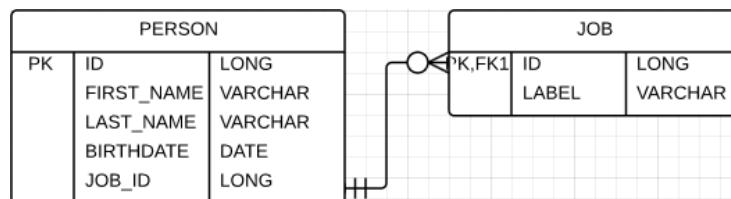
Table queries are enough when viewing/updating data from a single table. However, most of the time, data is scattered through more than one table. That is why SQL provides `JOIN`, a way to get results from multiple tables in a single select query.

References

SQL containers know how to reference other SQL containers so Vaadin can create relationships of sort without using a single line of SQL. In order to do that, the framework internally uses Reference instances. Such references are relationships between a referencing container (and its referencing column) and a referenced container (and its referenced column).



As an example, let's change our former table diagram to create n-to-1 job relationships for each person:



We are going to change the previous code to be able to view this new structure. In this case, this makes no sense to be generic:

```

public void populate(JDBCConnectionPool connectionPool) {

    QueryDelegate personsQuery =
        new TableQuery("PERSON", connectionPool);
  
```

```

QueryDelegate jobsQuery = new TableQuery("JOB", connectionPool);

try {

    personsContainer = new SQLContainer(personsQuery);
    jobsContainer = new SQLContainer(jobsQuery);

    /*1*/    personsContainer.addReference(jobsContainer, "JOB_ID",
    "ID");

    table.setContainerDataSource(personsContainer);

    table.addColumn("Job", new ColumnGenerator() {

        @Override
        public Object generateCell(Table source, Object itemId,
        Object columnId) {

            Item person = personsContainer.getItem(itemId);

            /*2*/    if (person.getItemProperty("JOB_ID").getValue() != null) {

                /*3*/    Item job = personsContainer.getReferencedItem(
                itemId, jobsContainer);

                @SuppressWarnings("unchecked")
                Property<String> property =
                job.getItemProperty("LABEL");

                return property.getValue();
            }

            return null;
        }
    });

    /*4*/    table.setVisibleColumns(new Object[] {
        "ID", "FIRST_NAME", "LAST_NAME", "BIRTHDATE", "Job", "" });

    } catch (SQLException e) {

        table.setComponentError(new SystemError(e));
    }
}

```

The most important lines in the preceding code are:

1. Create the relationship between the person's job ID in the PERSON container and the job ID in the JOB container.
2. Check whether the join is not made on a null value.
3. Retrieve the job as an item without coding a single line of SQL.
4. Finally, we should not forget to add the newly job generated column to the visible columns set.

The result looks like the following screenshot:

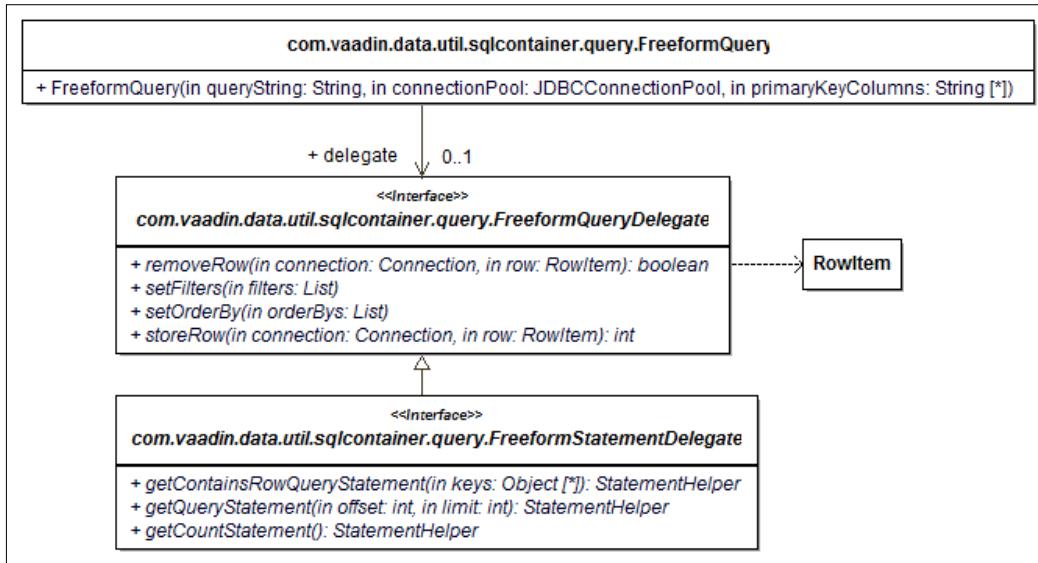
A screenshot of a web browser window displaying a Vaadin application. The address bar shows the URL `http://localhost:8080/myfirstvaadinapp/app`. The main content area contains a table with four rows of data. The columns are labeled `ID`, `FIRST_NAME`, `LAST_NAME`, `BIRTHDATE`, `JOB`, and an empty column. The `JOB` column contains the values "Hitman" for rows 3 and 4. Each row has a "Delete" button in the last column. At the bottom of the table are two buttons: "Commit" and "Rollback".

ID	FIRST_NAME	LAST_NAME	BIRTHDATE	JOB	
1	John	DOE	1/1/70		<button>Delete</button>
2	Jane	Doe	1/1/70		<button>Delete</button>
3	jules	winnfield	12/21/48	Hitman	<button>Delete</button>
4	Vincent	Vega	2/17/54	Hitman	<button>Delete</button>

Free form queries

As references, free form queries allow us to display data from different SQL tables into a single table component.

They offer finer grained control at the cost of requiring developers to write SQL code themselves. Moreover, they also require more effort, at least when executing CRUD (CRUD without read: INSERT, UPDATE, and DELETE) statements.



As an exercise, we are to implement the same display as before, but with free form queries instead of container references:

```

public void populate(JDBCConnectionPool connectionPool) {

    QueryDelegate personsQuery = new FreeformQuery(
        "SELECT P.ID AS ID, FIRST_NAME, LAST_NAME, BIRTHDATE, LABEL " +
        "AS JOB FROM PERSON AS P LEFT OUTER JOIN JOB AS J ON " +
        "P.JOB_ID = J.ID", connectionPool, "ID");

    try {
        personsContainer = new SQLContainer(personsQuery);
    }
}
  
```

```
table.setContainerDataSource(personsContainer);  
  
table.setVisibleColumns(new Object[] {  
    "ID", "FIRST_NAME", "LAST_NAME", "BIRTHDATE", "JOB", ""});  
  
} catch (SQLException e) {  
  
    table.setComponentError(new SystemError(e));  
}  
}
```

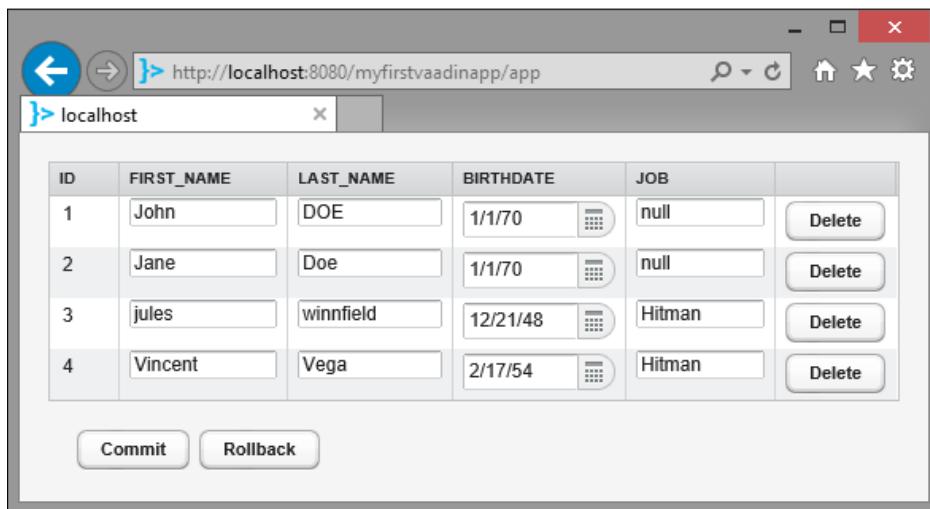
Pros and cons are readily visible; the code as a whole is much shorter but in exchange, we need to code the SQL select statement, which may be more or less easy depending on the relationships between tables.

Troubleshooting



Always be careful to use the right column name. When Vaadin complains about IDs must exist in the Container or as a generated column , missing id: xxx, the first thing to check is the visible column names case. As a first measure, it is advised to always use upper case when referencing column names from the database.

The preceding code produces the following output:



The only difference between using referenced containers and free form queries is that in the former case, referenced columns IDs are displayed as labels by default whereas in the latter, all non-identity columns are displayed as fields, thus read-write.

Using table queries, Vaadin can infer data types from the table metadata and create the adequate field. When using free form queries, it is impossible. As such, any CUD operation delegates to a free form statement query delegate that has to be set in order to execute such operations.

Troubleshooting

If Vaadin complains about `java.lang.UnsupportedOperationException: FreeFormQueryDelegate not set`, it is because setting a delegate to handle updates to the underlying data was forgotten.

Developing a free form query delegate may be done at two different levels:

- For simple implementations, `FreeformQueryDelegate` is enough to fit our needs.
- If we use `PreparedStatement` instead of regular statements, we need to implement `FreeformStatementDelegate`. However, remember that `PreparedStatement` is the preferred way to prevent SQL injection attacks.

The framework code is smart enough to adapt what is used to the underlying interface.

The case for prepared statements

Considering `PreparedStatement` is compiled once to the native RDBMS internal language and then reused over and over, and the prevention against SQL injection attacks, it is advised to always use it (see https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet for the detailed OWASP Cheat Sheet reference).

Related add-ons

Should SQL be a little too old-fashioned, some add-ons may be of interest (see *Chapter 8, Featured Add-ons*, on add-ons for a description of what they are):

- A container over a Hibernate backend, `HbnContainer` is provided by *Gary Piercy*. It is provided in Version 2.0.1 under the friendly Apache 2.0 license and is considered beta. More information can be found on the add-on homepage <https://vaadin.com/directory#addon/hbncontainer>.

- EclipseLink is not well represented as the associated container is only compatible with Vaadin 6 and has seen no update since late 2009: <https://vaadin.com/directory#addon/eclipselink-container>.
- Finally, JPACContainer is a certified add-on provided by Vaadin Ltd. It is also available under Apache 2.0 license (since v2.2.0) and provides a nice integration for components to use a Java Persistence API backend. It is described in *Chapter 8, Featured Add-ons*, in the relevant section.

Server push

Traditional HTML-based applications use the request-response model. Requests are initiated by the client, sent to the server, and the latter sends the response back to the client.

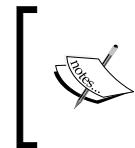
Nonetheless, some use-cases require the server to notify the client without the client initiating the sequence. Such use-cases would include chat applications, real-time trading platforms, and so-on.

Legacy techniques to emulate server push include:

- **Traditional polling:** At regular intervals, the client sends the server requests. If there is data to be communicated, it is sent back in the response; if not, the response is empty. The ratio of empty versus meaningful responses is generally an argument against this approach.
- **Long polling:** As in the previous strategy, the client sends requests at regular intervals. But instead of sending empty responses, the server keeps the connection until data becomes available. At this time, it is sent back to the client.
- **Long-lived HTTP connections:** This way is the same as the previous one, except that requests are not sent regularly but when they are necessary.

The modern way to do server push is to use WebSocket: bi-directional communication between client and server that is part of HTML5. This means we need to use relatively-modern browsers that support this feature.

Vaadin versions prior to 7.1 rely on external add-ons (see *Chapter 8, Featured Add-ons*, for a word on add-ons), but with 7.1, push comes with core and that is very good!

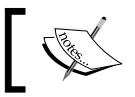


Users stuck with previous versions should migrate to 7.1, or take a look at the ICEPush add-on located at <https://vaadin.com/directory#addon/icepush>, based on the excellent ICEPush technology.



Push innards

Vaadin push is based on the Atmosphere framework, an open source project hosted on GitHub (<https://github.com/Atmosphere/atmosphere>).



Knowing these details is by no mean mandatory to use server push. Just skip this section to concentrate on usage, if you prefer.



Under the hood, Atmosphere uses WebSocket. However, for users using less state-of-the-art browser and version combinations, Atmosphere provides a fallback based on the Comet framework that aim to emulate server push through techniques described in the preceding section.

All in all, this means there are a couple of minimum versions for both server and browsers. Here is an excerpt for both:

Server	Minimum compatible version
Eclipse Jetty	7
Oracle Glassfish	3.1.2
Apache Tomcat	7.0.27

Browser	Minimum compatible version
Mozilla Firefox	9
Google Chrome	13
Microsoft Internet Explorer	10
Opera	11
Apple Safari (OS X)	5

The full compatibility matrix is available on Atmosphere's wiki <https://github.com/Atmosphere/atmosphere/wiki/Supported-WebServers-and-Browsers>.



[For Eclipse users, this means we have to use Tomcat instead of the J2EE Preview Server.]

Installation

In order to use server push, we need to add an additional dependency in Ivy.
The snippet to use is:

```
<dependency org="com.vaadin" name="vaadin-push" rev="&vaadin.version;"  
conf="default->default" />
```



[Vaadin version should at least be 7.1 for the library to be available.]

How-to

To push, annotating the UI with `com.vaadin.annotations.Push` is necessary.
That's it!

Yet if you just do that, chances are that the UI changes will not work; there is a single requirement to fulfill and that is that the pushed UI is locked to prevent concurrent changes. In order to do so, Vaadin provides the `access(Runnable)` method in the UI class. It is somewhat akin to a synchronized block in standard Java code.

As for synchronized blocks, code in `access()` methods should be as short as possible, in order to be executed as quickly as possible. Remember that locks are performance bottlenecks.

In any way, notice there is no change beside that; most importantly, the web deployment descriptor stays untouched.

Example

As an illustration, we will create a clock, refreshed by the server. Our first class is the runnable that will update the label:

```
import static java.text.DateFormat.MEDIUM;

import java.text.DateFormat;
import java.util.Calendar;
import java.util.Date;

import com.vaadin.ui.Label;

final class LabelUpdaterRunnable implements Runnable {

    private static final DateFormat DATE_FORMAT =
        DateFormat.getTimeInstance(MEDIUM);

    private Label timeLabel;

    LabelUpdaterRunnable(Label timeLabel) {

        this.timeLabel = timeLabel;
    }

    protected String getCurrentTime() {

        Date date = Calendar.getInstance().getTime();

        return DATE_FORMAT.format(date);
    }

    @Override
    public void run() {

        timeLabel.setValue("Time: " + getCurrentTime());
    }
}
```

Nothing very interesting here: we pass a label and update its value when the runnable is run. Code that does the real magic follows:

```
import com.vaadin.annotations.Push;
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

@SuppressWarnings("serial")
@Push
public class ServerPushUI extends UI {

    private Label timeLabel = new Label();

    @Override
    protected void init(VaadinRequest request) {

        HorizontalLayout layout = new HorizontalLayout(timeLabel);
        layout.setMargin(true);
        setContent(layout);

        new Thread(new EndlessRefresherRunnable()).start();
    }

    private class EndlessRefresherRunnable implements Runnable {

        @Override
        public void run() {

            while (true) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}

                access(new LabelUpdaterRunnable(timeLabel));
            }
        }
    }
}
```

The `EndlessRefresherRunnable` has the following features:

- Implements `Runnable` so it can be run as a thread.
- Loop over a combination of:
 - Waiting for one second.
 - Passing the previous runnable to the `access()` method. Vaadin takes care of locking the UI and launching the thread parameter.

Threads in web application



The preceding code should not be taken at face value and used in a production environment, as not only spawning new threads is frowned upon in a Java EE application, but this one has no end. Threads should not be started unless you deeply know what you do. Real-world application should probably use an event-based stack, such as JMS, an EJB timer, Java 7 thread pools, or a third-party framework like Quartz.

The whole application code can be found at <https://github.com/nfrankel/server-push>.

Twaattin improves!

Given the current state of Twaattin and our understanding of Vaadin, automatically refreshing the displayed timeline when new tweets are published seems like a natural improvement.

In this section, we will describe how to do that. Changes are needed to use server push but also Twitter4J streaming features. The latter is out of the scope of this book; just know that a specified Twitter4J module implements the Observer pattern so we can automatically be notified of new Twitter events.

Ivy dependencies

We need to do the following:

- Upgrade to at least version 7.1.0 of Vaadin
- Add two dependencies, `vaadin-push` and `twitter4j-stream`

```
<?xml version="1.0"?>
<!DOCTYPE ivy-module [
  <!ENTITY vaadin.version "7.1.0 ">
```

```
    ] >
<ivy-module version="2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/
  schemas/ivy.xsd">
  ...
<dependencies>
  ...
  <dependency org="org.twitter4j" name="twitter4j-core"
    rev="3.0.3" />
  <dependency org="org.twitter4j" name="twitter4j-stream"
    rev="3.0.3" />
</dependencies>
</ivy-module>
```

Twaattin UI

The meaty changes take place in `TwattinUI`; we of course have to annotate it with `@Push` to enable server push. We also need to make a couple of changes, particularly from a design point of view:

1. We learned previously of the `access()` method on the UI. Therefore, we will use this to our advantage and make the UI our observer, tasked with propagating events.
2. The tweet refresher behavior is already taking care of making changes. However, it should be the same instance used for a single user. It should have the same scope as the UI, so in the absence of dependency injection framework, we will manage it from there.
3. Finally, Twitter4J stream takes care of notifying us of new events. Therefore, we just have to wrap GUI updating code inside a `Runnable` interface and pass it as a parameter to `access()`.

```
@Push
public class TwattinUI extends UI implements UserStreamListener {

    private static final long serialVersionUID = 1L;

    private TweetRefresherBehavior tweetRefresherBehavior = new
    TweetRefresherBehavior();

    ...

}
```

```

public static TwaattinUI getCurrent() {

    return (TwaattinUI) UI.getCurrent();
}

public TweetRefresherBehavior getTweetRefresherBehavior() {

    return tweetRefresherBehavior;
}

@Override
public void onException(Exception ex) {

    setComponentError(new SystemError(ex));
}

@Override
public void onStatus(final Status status) {

    access(new Runnable() {

        @Override
        public void run() {
            tweetRefresherBehavior.updatedStatus(status);
        }
    });
}

// A bunch of methods to implement
...
}

```

The code being a little long, it has been abridged. The entire source can be found online at the following address: <https://github.com/nfrankel/twaattin/blob/chapter7/src/com/packtpub/learnvaadin/twaattin/ui/TwaattinUI.java>



Note that we only implemented methods to be notified of a new tweet and when something unexpected happens. Other methods can be left untouched or implemented, depending on one's taste.

Tweet refresher behavior

The current refresher behavior gets the user timeline and adds them to the table when the latter is attached to the UI.

We have to change that a little to only store the reference to the table when attached and add tweets each time it is called by the UI.

 As an improvement, we will store the container instead of the table, to ease our task. Otherwise, we would have to create the BeanItem object. Better let the framework do that for us.

```
package com.packtpub.learnvaadin.twaattin.presenter;

import twitter4j.Status;

import com.vaadin.data.util.BeanItemContainer;
import com.vaadin.ui.Component;
import com.vaadin.ui.HasComponents.ComponentAttachEvent;
import com.vaadin.ui.HasComponents.ComponentAttachListener;
import com.vaadin.ui.Table;

@SuppressWarnings("serial")
public class TweetRefresherBehavior
    implements ComponentAttachListener {

    private BeanItemContainer<Status> container;

    public void updatedStatus(Status status) {
        if (container != null) {
            container.addBean(status);
        }
    }

    @SuppressWarnings("unchecked")
    @Override
    public void componentAttachedToContainer(
        ComponentAttachEvent event) {
        Component component = event.getAttachedComponent();
```

```
if (component instanceof Table) {  
  
    Table table = (Table) component;  
  
    container = (BeanItemContainer<Status>)  
        table.getContainerDataSource();  
}  
}  
}
```

Twitter service

The final but mandatory touch is to use Twaattin4J Streaming API instead of the standard one. This means replacing TwitterFactory and Twitter by their respective counterparts TwitterStreamFactory and TwitterStream.

The whole codebase of Twaattin implementing server push can be found on GitHub for reference <https://github.com/nfrankel/twaattin/tree/chapter7>.

Summary

In this chapter, we learned about some advanced features of Vaadin that span a large perimeter. In particular, we saw the following use-cases and the way Vaadin could meet our needs out-of-the-box:

- Should we need to go deeper in order to access any of the available web contexts (servlet, application, and session), we just have to use the getCurrent() method on the different objects.
- With the help of Vaadin Navigation API, we can capture state and set it as a URL fragment on the address bar while staying on the same screen. Also, the reverse can be done: taking the fragment and using it to set data in the application.
- Moreover, we discovered how legacy applications and/or sites can be integrated with Vaadin application and how more than one Vaadin application can run in the same context, with just a touch of HTML and JavaScript configuration.

- If the default error handler mechanism does not suit us, we know how to override it to do exactly what we want with the `ErrorHandler` interface.
- When the need to integrate with a SQL backend comes up, we know how to put our newfound knowledge to good use with the `SQLContainer` API. With it, it is easy as pie to map database tables to UI tables and columns to lines.
- Last but not least, we learned how to push data from the server to the client.

At this point, Vaadin should be fairly well understood; important features are part of the core framework and are available out-of-the-box. In the next chapter, we will get on the next step to add additional features such as more widgets, more persistence features, themes, and others.

8

Featured Add-ons

As the Vaadin team wants to keep the framework as cohesive as possible, some very interesting capabilities are not integrated into the core framework, but are available as add-ons.

In this chapter, we will describe the Vaadin add-ons portal then detail some add-ons that bring new features or just make our life easier. These are as follows:

- Button group is a UI add-on, which is very easy to use yet a great introduction to understand the underlying GWT layer
- Clara is a way to declare UI (*a la Flex*) instead of coding them
- JPA container to connect to a JPA backend, so as to directly display data into Vaadin tables
- CDI Utils is an add-on to take advantage of CDI-enabled containers

In Vaadin, third-party modules are called add-ons. Some are supplied by Vaadin Ltd, but most are supplied by others (either individuals or companies).

Other add-ons of interest include: Vaadin Charts, a library to help visualize data in 2D charts, TestBench, an end-to-end testing tool, and TouchKit, a theme aimed entirely toward mobile users user-experience.

Vaadin add-ons directory

Vaadin provides an add-on store available online at <http://vaadin.com/directory>. Should you become interested in providing add-ons, know that this repository is accessible to do this.

Add-ons search

The store provides categorization in order to search for certain specific features. Those categorizations are available in different flavors.

Results obtained using the following criteria, either alone or in conjunction, can then be sorted. Sorting is possible by release date, number of downloads, and grade.

Typology

The available categories are as follows:

- **UI components:** These add-ons have graphical representations that are displayed to users when added to a window.
- **Data components:** This category groups add-ons containers that can connect to data tiers, such as SQL databases. These connectors provide a way to manage data (direct SQL, JPA, Hibernate, and so on).
- **Themes:** These add-ons offer a quick route to change an application's look and feel.
- **Tools:** These add-ons concern themselves with integration with third-party products (excluding data containers) such as Spring, languages such as Scala, build tools such as Gradle, and so on.
- **Miscellaneous:** Add-ons that are not part of another category are put in here.
- **Official:** This cross-cutting category lists all add-ons provided by Vaadin Ltd, whether available commercially or freely under an open source license.

Stability

Add-ons are classified into different stability levels as follows:

- **Certified:** These add-ons are provided by Vaadin Ltd, the company behind the framework, and guarantee the highest level of reliability and integration. Third-party add-ons can also be certified by the company, against a set fee.
- **Stable:** These add-ons have been subject to at least one whole release lifecycle. They can usually be trusted to function in an expected way.
- **Beta:** These add-ons are just that. It is advised not to use them in a production environment because they do not guarantee to be completely bug-free.
- **Experimental:** These add-ons should not be used beyond R&D: they are here just to give you a preview of what is to come. If interested, you should probably contact the author for help!

Add-ons presentation

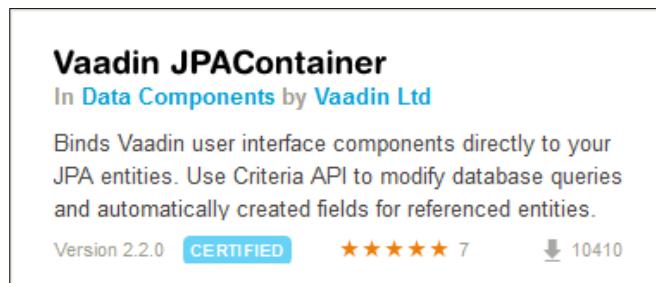
Add-on presentation is handled by the directory and thus highly standardized. Each add-on has two views: a summarized view (displayed in search results) and a detailed view.

Summarized view

Displayed information in the summarized view includes the following:

- Name
- Category
- Author
- A short description
- Version
- Maturity
- Rate (from 1 to 5)
- Number of downloads

As an example, here is a screenshot for the Vaadin JPAContainer summary:



Detailed view

After clicking on the add-on's name, the detailed view opens. In addition to the previous information, it displays the following data:

- Version; it is selectable so we can see data related to previous versions.
Note that some field values are dependent on the selected version.
- Maturity, it is version-dependent.
- License information, it is also version-dependent.

- Matrix of browsers compatibility including versions, it is version-dependent, too.
- Vaadin versions compatibility.
- Detailed description.
- One or more highlights area-it displays a pop-up window usually with either code example or screenshot.
- Release notes, it is version-dependent.
- History of ratings (and their associated comment) for each published version.
- Download link, of course it is version-dependent.
- Maven snippet to include in the POM, it is version-dependent.
- Depending of the license nature, a link to purchase the add-on or to subscribe to Vaadin Pro Account which includes the add-on use. More info on Vaadin Pro Account can be found at <https://vaadin.com/pro>.
- List of links; commons links include (but are not limited to):
 - Demo application
 - Source Control Manager repository
 - Afferent documentation
 - Discussion forum
 - Issue tracker
- Rate widget; rating may be associated with a comment.
- Page's permalink.

Here is the detailed page for the JPAContainer add-on:

Vaadin JPAContainer

In Data Components by Vaadin Ltd ★★★★★ 7 10410

[Report this add-on](#)

Version	3.0.0.beta1	▼
Maturity	CERTIFIED	Browser Compatibility
License	Apache License 2.0	Browser independent
Vaadin	7.0+ Available for 6	

Overview

Vaadin JPAContainer allows connecting Vaadin user interface components directly to the persistent model objects. It is an implementation of the container interface defined in the Vaadin core framework. You can use JPAContainer to display data in a table, tree, any other selection component, or edit the data in a form. JPAContainer uses 3rd party JPA 2.0 (JSR-317) standard supporting Object-Relational Mapping (ORM) libraries, such as EclipseLink or Hibernate, for storing and retrieving data from database.

Vaadin JPAContainer 2.0 supports the most common features required by Java EE applications out of the box, such as lazy loading, advanced filtering, nested property names and caching. JPAContainer utility classes makes writing CRUD functionality to your application fast and

Highlights



Setting up a ...



Relation fiel...



FieldFactory ...

Release notes

3.0.0.beta1:
 Vaadin 7 support
 License changed to Apache 2.0

Latest Ratings

Version 3.0.0-alpha2 Released — Dec 20, 2012 3:04 PM

★★★★★ Sergei Zver — Dec 12, 2012 12:44 PM

Download Now Version 3.0.0.beta1 (38 MB) 

Maven POM 

Related Links

- Add-on Homepage
- Tutorial
- Issue Tracker
- Source Code
- Discussion Forum

Rate this Add-on

★★★★★

Permalink to this add-on:

<http://vaadin.com/addon/vaadin-jpacontainer>

Noteworthy add-ons

At the time of writing this book, the Vaadin directory contained 100 add-ons, stable or certified (half are Vaadin 7-compatible) on a total of more than 350. It is well beyond the scope of this book to them all in detail. However, a few are worth describing, as they really enhance Vaadin capabilities.

Button group

Button group is one of many components available as add-ons. Not only does it have a nice visual display – reducing space between buttons in the same group to nothing, it is a good example of an add-on providing client-side code.

A dedicated page can be found at <https://vaadin.com/directory#addon/buttongroup>.

Prerequisites

The Vaadin plugin comes bundled with Ivy, the dependency management tool that we have already used when integrating Twitter4J in [Twaattin](#).

Use of button group is achieved by adding the following line into the `ivy.xml` file:

```
<dependency org="org.vaadin.addons" name="buttongroup" rev="2.3" />
```

Core concepts

If we naively add the dependency, then just try to add a `buttonGroup` component to some UI, we will get similar error messages:

Widgetset does not contain implementation for org.vaadin.peter.buttongroup.ButtonGroup. Check its component connector's @Connect mapping, widgetsets GWT module description file and re-compile your widgetset.

In case you have downloaded a Vaadin add-on package, you might want to refer to add-on instructions.

This means things are getting serious; up to this point, we could code only in Java. In order to make this add-on that includes client-side code work, we have to go deeper in to our GWT knowledge.

GWT modules

We learned in earlier chapters that GWT produces client-side code (HTML, JavaScript, and CSS). In order to address complexity and reusability issues brought by real-world applications, GWT provides a standardized way toward modularity. Essentially, modules are packages with reusable functionalities.

Each module must provide an XML descriptor file which ends with `gwt.xml`. The un-suffixed file name is the module's name. This descriptor has to be placed in a Java-like package structure.



Finer details about GWT modules are out of the scope of this book. Please refer to <http://code.google.com/p/google-web-toolkit-doc-1-5/wiki/DevGuideModules>.



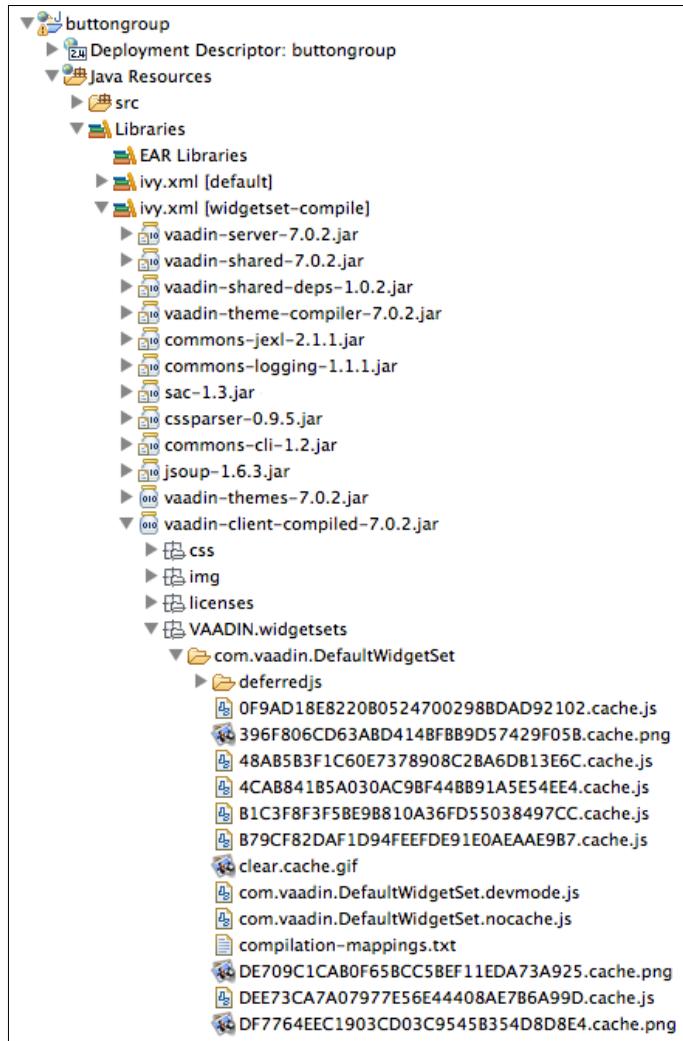
For now, suffice to know that modules offer a way to describe both module composition and an application single entry-point.

Widget sets

Vaadin make a restricted usage of modules with only composition and entry-point. To differentiate between full-blown modules and those limited modules, Vaadin-restricted modules are called widgetsets in Vaadin parlance.

To speak the truth, widgetsets have been used all throughout our previous examples, only without our knowledge. Just get your hands on whatever Vaadin project you have at the moment and search for a `vaadin-client-compiled-7.x.y.jar` file. Then look for a `VAADIN.widgetset` folder: GWT developers will not be surprised here, as it is the exact structure of a GWT compiled application.

The following screenshot displays a view of the project structure:



With this client-compiled JAR, Vaadin supplies GWT-compiled widgets. The main difference between widgets and components is that the former are client-side and compiled to JavaScript, while the latter are server-side and compiled to bytecode. All Vaadin components we used formerly have a widget counterpart inside the Java.



We will describe exact relations between components and widgets in *Chapter 9, Creating and Extending Components and Widgets*.



An important Vaadin advantage is that the default widgetset is already compiled. This means we do not need to compile it during development. Calling display-related setter methods (or passing constructor arguments) in Java code will send messages from the server to the client.

GWT has chosen another path: calling setter methods translates into compiled-code updates and causes significant coding freezes when they take place.

How-to

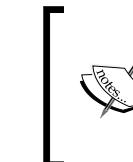
GWT can only accept a single entry-point, thus a single module. When there are multiple modules, we have to create a composition structure. In this case, the new module composing all others can be set as the application entry point.

The easiest way to achieve this is through the Vaadin Eclipse plugin and now we can do this more precisely, thanks to the compile Vaadin widgets button located in the toolbar, as shown in the following screenshot:



After clicking on this magic button with a selected Vaadin project and additional widgetsets, we will do the following:

1. Create a relevant `gwt.xml` file under a `widgetset` sub-package relative to the configured UI in the sources folder. It describes the composition described previously (references to all widgetsets, including the default one).



Do not change the generated widgetset filename!

Neither package name nor file name (nor extension) should be changed: the framework is will look for this specific file and no other.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc./DTD Google Web Toolkit
1.7.0//EN" "http://google-web-toolkit.googlecode.com/svn/
tags/1.7.0/distro-source/core/src/gwt-module.dtd">
```

```
<module>
  <inherits name="com.vaadin.DefaultWidgetSet" />
  <inherits name="org.vaadin.peter.buttongroup.
ButtonGroupWidgetset" />
</module>
```

2. Add a reference to this GWT file in the web deployment descriptor as a Vaadin servlet init-param. The code for performing the same is as follows:

```
<servlet>
  <servlet-name>Buttongroup Application</servlet-name>
  <servlet-class>com.vaadin.server.VaadinServlet</servlet-class>
  <init-param>
    <param-name>UI</param-name>
    <param-value>
      com.packtpub.learnvaadin.ButtonGroupUI
    </param-value>
  </init-param>
  <init-param>
    <param-name>widgetset</param-name>
    <param-value>
      com.packtpub.learnvaadin.widgetset.ButtonGroupWidgetset
    </param-value>
  </init-param>
</servlet>
```

3. Last but not least, it compiles client-side code into GWT-compatible JavaScript (look at the new VAADIN folder in WEB-INF).

This is an excerpt of GWT engine compiling output:

```
Executing compiler with parameters [/Library/Java/JavaVirtualMachines/
jdk1.7.0_10.jdk/Contents/Home/bin/java, -Djava.awt.headless=true,
-Xss8M, -Xmx512M, -XX:MaxPermSize=512M, -classpath, ... -Dgwt.
persistenunitcachedir=/var/folders/_s/4wk9w3fs6xs70t3sb8g
ptrpsb4lk97/T/widgetset_com.packtpub.learnvaadin.widgetset.
ButtonGroupWidgetsetdb4ceafdb4f50-afed-e47f849d5f52, com.
vaadin.tools.WidgetsetCompiler, -war, WebContent/VAADIN/widgetsets,
-deploy, /var/folders/_s/4wk9w3fs6xs70t3sb8gptrpsb4lk97/T/widgetset_
com.packtpub.learnvaadin.widgetset.ButtonGroupWidgetsetdb4ceafdb4f50-
afed-e47f849d5f52, -extra, /var/folders/_s/4wk9w3fs
6xs70t3sb8gptrpsb4lk97/T/widgetset_com.packtpub.learnvaadin.
widgetset.ButtonGroupWidgetsetdb4ceafdb4f50-afed-e47f849d5f52,
-localWorkers, 8, -logLevel, INFO, com.packtpub.learnvaadin.widgetset.
ButtonGroupWidgetset]

Updating GWT module description file...

Process output
```

```
Compiling
Compiling permutation 1...
Compile of permutations succeeded
Linking into /Users/nicolas.fraenkel/Google Drive/Learning Vaadin/
workspace/buttongroup/WebContent/VAADIN/widgetsets/com.packtpub.
learnvaadin.widgetset.ButtongroupWidgetset; Writing extras to /var/folder
s/_s/4wk9w3fs6xs70t3sb8gptrpsb4lk97/T/widgetset_com.packtpub.learnvaadin.
widgetset.ButtongroupWidgetsetdb4ceafad-a8ec-4f50-afed-e47f849d5f52/com.
packtpub.learnvaadin.widgetset.ButtongroupWidgetset
Link succeeded
Compilation succeeded -- 71.941s
```

Once this has been performed and updates have been published on the server, the application should run as expected. The following screenshot should be displayed:



Alternatively, we can manually replace preceding second part (the web.xml part) by a simple annotation on the UI:

```
@Widgetset("com.packtpub.learnvaadin.widgetset.ButtongroupWidgetset")
public class ButtonGroupUI extends UI {
    ...
}
```

Vaadin will first look if there is a @Widgetset on the UI, and if not found fallback on the web deployment descriptor. Real-life usage depends on one's needs and taste.

Conclusion

This specific add-on serves as a nice illustration of using add-ons embedding widgets. It also is a space-saver in some already tight-packed UIs.

Sources of the preceding example with tags for each configuration flavor (web.xml versus annotation) can be found at <https://github.com/nfrankel/buttongroup>.

Clara

Believe it or not, one the most heard complaint about Vaadin is that it does not use XML for UI design, as for Flex and Android, but only pure Java. After all you hear against XML on the web, it may seem strange, but it is still in popular demand.

Clara is an attempt to remedy to that.



Clara is developed by *Teemu Pöntelin*, a member of the Vaadin team, but is not endorsed by Vaadin Ltd (yet).



Details on the plugin may be found at <https://vaadin.com/directory#addon/clara>. Currently there is no similar add-on.

Prerequisites

As for button group, adding dependencies takes place in the `ivy.xml` file.

```
<dependency org="org.vaadin.addons" name="clara" rev="1.0.0" />
```

How-to

Here is a description on how to use Clara.

XML

XML files use two different schemas:

- One for UI components, including layouts, `urn:package:com.vaadin.ui`
- The other for layout alignment, `urn:vaadin:layout`

Each UI component (and layout) of the Vaadin framework can be displayed by using a tag corresponding with its exact unqualified class name.

```
<TextField />
```

Setting **simple** values on those components can be called by using the property name as a XML attribute.

```
<TextField caption="FirstName" />
```

Adding components to containers is achieved by using embedded tags.

```
<HorizontalLayout>
  <TextField caption="FirstName" />
</HorizontalLayout>
```

As an illustration, let us remake the Twaattin front page using Clara:

```
<?xml version="1.0" encoding="UTF-8"?>
<VBoxLayout xmlns="urn:package:com.vaadin.ui"
  xmlns:l="urn:vaadin:layout" margin="true" spacing="true">
```

```
<Link caption="Get PIN" targetName="twitterauth" />
<TextField inputPrompt="PIN" />
<Button caption="Submit" />
</VerticalLayout>
```

And this is it: everything is now in place, with only a few lines of XML!

Inflating

Android has popularized the term "inflate" to describe the process of passing binary stream containing XML to a factory to get *bytecode* made of graphical components.

Clara inflating is equally simple: just use `Clara.create(InputStream)`. The standard way to get an `InputStream` handle on files in a Servlet context is the following:

```
httpRequest.getServletContext().getResourceAsStream("/WEB-INF/
twaattin.xml");
```

Adding behavior

XML is a first-class citizen to define UI but is found lacking for defining behavior: it is the realm where programming languages such as Java rule. We just need to have a bridge between UI XML and Java event handlers.

Clara provides that through the `@UiHandler` annotation. It requires the `id` attribute of the XML component and has to be set on the handling Java method we want to associate the component with.

```
public class ClaraBehavior {

    @UiHandler("submitButton")
    public void handleSubmitButtonClick(ClickEvent event) {

        Notification.show("You're authenticated into Twaattin");
    }
}
```

We need to update the XML with an attribute `id` having the same value.

```
<Button id="submitButton" caption="Submit" />
```

Also, we need to pass the behavior class as a parameter to the `create()` method.

```
Component component = Clara.create(xml, new ClaraBehavior());
```

Data sources

XML simple grammar is a limitation: we can only create components and add some layout data. Most Vaadin applications require data sources. Clara provides an easy way to bind them to components with the `@UiDataSource` annotation.

It has to be set on a method whose return type is a data source. As seen in *Chapter 6, Containers and Related Components*, those are dependent on the component type:

- Property for `TextField` and `Label`
- Item for `Form`
- Container for `AbstractSelect`

`@UiDataSource` needs a value which corresponds to the component `id`.

The following snippet sets the value of the `pin` text field in a dynamic way:

```
@UiDataSource("pin")
public Property<String> getLabelProperty() {

    return new ObjectProperty<String>("1111", String.class);
}
```

Creating complex components

There are numerous requirements beyond data sources that XML cannot address, for example, the PIN link of our Twaattin example using `ExternalResource`.

In order to address this issue, Clara offers a way to retrieve any component it created, through the `Clara.findComponentById(Component, String)` method with:

- Component being the inflated XML
- String the value of the component `id` attribute

We will continue with our Twaattin example. Here is the rest of the code:

```
Link link = (Link) Clara.findComponentById(component, "twitterLink");
link.setResource(new ExternalResource("http://twitter.com"));
```

Limitations

There are limitations in the current Clara version, which definitely prevents us from porting Twaattin to Clara.

One of those limitations is that it is impossible to pass components to event handler class constructor, as for `Twaattin LoginBehavior`, which requires a PIN field. This is a chicken and egg problem: inflating the XML requires the behavior class, but the behavior requires a reference to field which can only be available if the XML is inflated!

Of course, there are workarounds for this problem: one could change the design to create a setter for the behavior, for example. Yet, those are just workarounds that break previous immutability.

Conclusion

Despite its "coolness" and ease of use, Clara is still experimental. It is OK to play with it and to conduct personal projects but beware before using it "for real".

However, just keep it under close watch—it has the potential to become one of the most popular plugins around.

Sources of the preceding example can be found at <https://github.com/nfrankel/clara>.

JPA Container

JPAContainer is an add-on that let us use a specific container bound to a predefined **Java Persistence API (JPA)** model.

The plugin itself can be found at <https://vaadin.com/directory#addon/vaadin-jpacontainer>. The plugin is both provided by and certified by Vaadin Ltd. It is released under the Apache 2.0 License.



Read carefully before using previous versions of the plugin (up to 2.1.0): those are released under a dual AGPL 3.0 and CVAL 2.0 licenses, take care.

Similar add-ons include the following:

- Though not an add-on anymore, SQL Container (seen in *Chapter 7, Core Advanced Features*) was once an add-on and let us directly connect to a database using SQL
- HbnContainer to use a predefined Hibernate entity model

Concepts

JPA is a Java EE standardized way to access SQL backends. JPA has been available since Java EE 5.

JPA version	Java EE version	JSR
1.0 (under umbrella EJB 3.0)	5	220
2.0	6	317
2.1	7	338

Describing JPA is way out of the scope of this book; we will assume we already have a working JPA model.

Prerequisites

Using JPA container requires some additional steps.

Dependency

As before, let us add the add-on dependency to our project:

```
<dependency org="org.vaadin.addons" name="jpacontainer"
rev="3.0.0.beta1" />
```

Model

We also need a pre-existing model. We will reuse the Person/Job model of *Chapter 7, Core Advanced Features*.

Here are the sources, annotated the JPA-way:

```
package com.packtpub.learnvaadin.jpa.model;

import static javax.persistence.TemporalType.DATE;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Temporal;
```

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "FIRST_NAME")
    private String firstName;

    @Column(name = "LAST_NAME")
    private String lastName;

    @Temporal(DATE)
    private Date birthdate;

    @ManyToOne
    private Job job;

    // Getters & setters
    // ...
    // Do not forget equals() & hashCode() !
    // ...
}

package com.packtpub.learnvaadin.jpa.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Job {

    @Id
    @GeneratedValue
    private Long id;
    private String label;

    // Getters & setters
    // ...
    // Do not forget equals() & hashCode() !
    // ...
}
```

Finally, here is the persistence descriptor to use:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="vaadindemo" transaction-type="JTA">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider
        </provider>
        <jta-data-source>jdbc/persons</jta-data-source>
        <!--Prevents:
            http://netbeans.org/bugzilla/show_bug.cgi?id=181068-->
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.ddl-generation"
                value="create-tables" />
        </properties>
    </persistence-unit>
</persistence>
```

Nothing fancy here. Mainly, we declare the `eclipselink` persistence provider, which is used by Glassfish as well as:

- The persistence unit name
- The JNDI name the datasource will be bound to

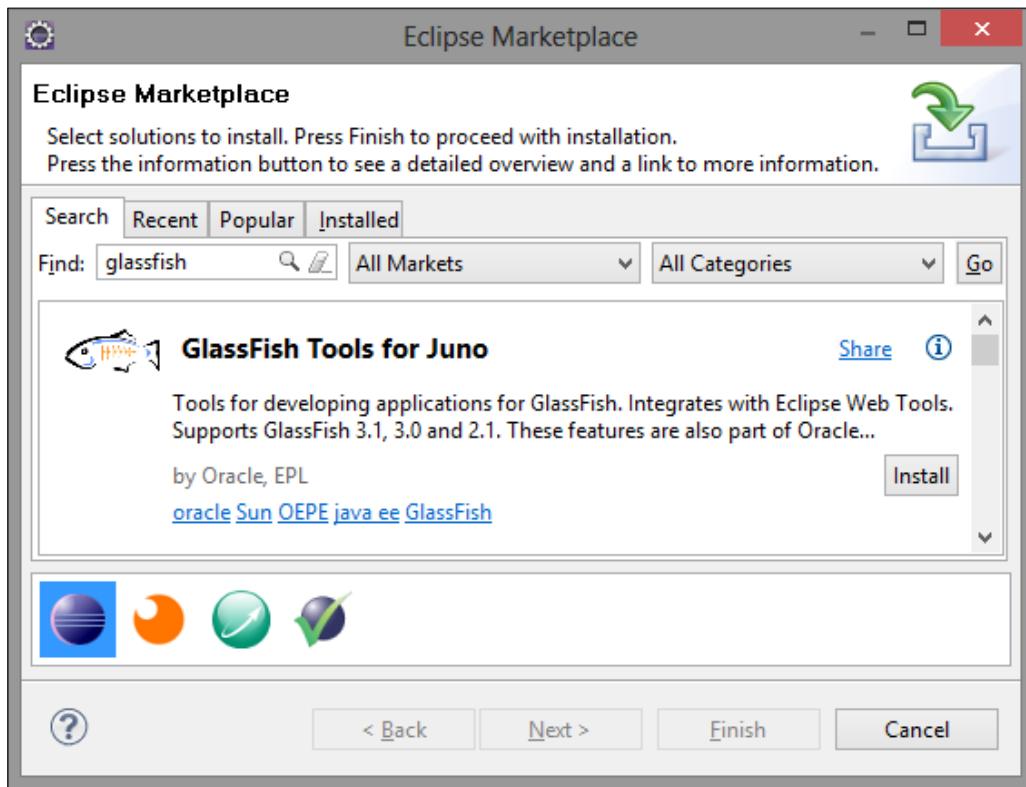
Application server

Using JPA requires both the API itself but also an implementation (for example, Hibernate or EclipseLink). It is entirely possible to run JPA-based applications on simple servlet containers like Tomcat, with both API and implementation embedded into the Web Application Archive libraries. However, using a full-blown application server guarantees JPA works exactly as expected as part of Java EE, and keeps the WAR size lower as well.

There are plenty of free open source application servers available on the market. Were only JPA in the balance, we would probably go for TomEE, a fully Java EE 6 compliant Tomcat (<http://tomee.apache.org/apache-tomee.html>). However, we will also need an OSGi container in later chapters, so it would be a good move to choose an application server that can do both. In other words, we are going to use GlassFish, Oracle's open source (and free!) application server.

Download, install, and integrate

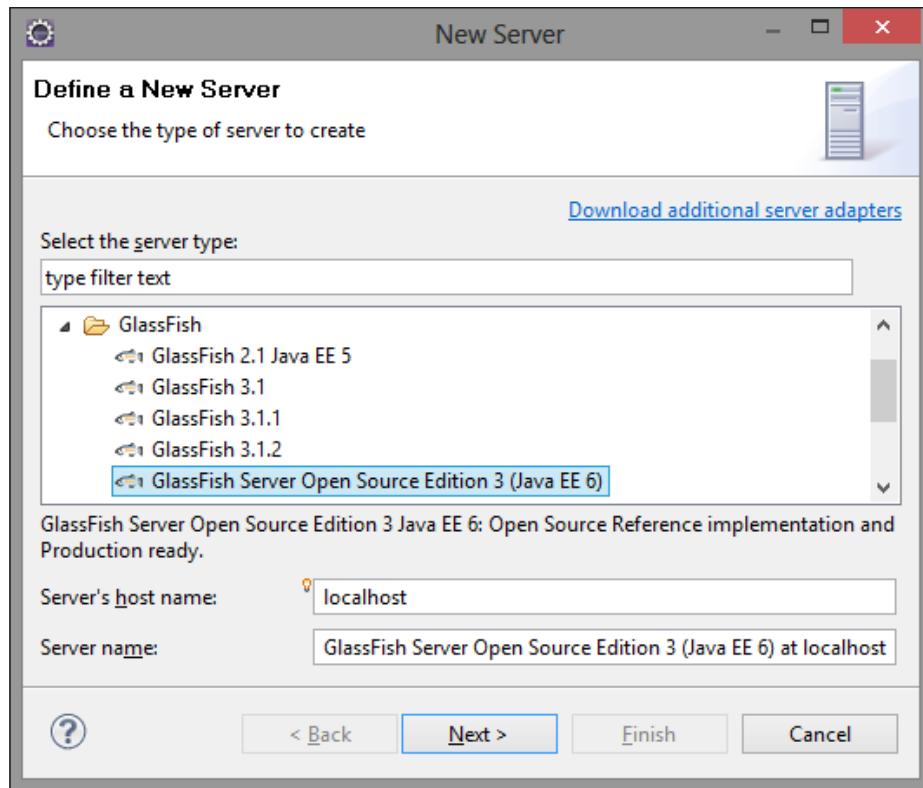
In order to integrate Glassfish within the Eclipse IDE, we have to add a plugin to the latter. In Eclipse, go to **Help | Eclipse Marketplace**. In the opened dialog box, search for `glassfish`. Choose the plugin adapted to the installed Eclipse version- if prior Eclipse installation instructions were followed, that would be **Glassfish Tools for Juno**.



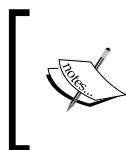
Click on the **Install** button. Follow the wizard by selecting the single component to install and accept the license agreement. Restart Eclipse as the end dialog asks.

Featured Add-ons

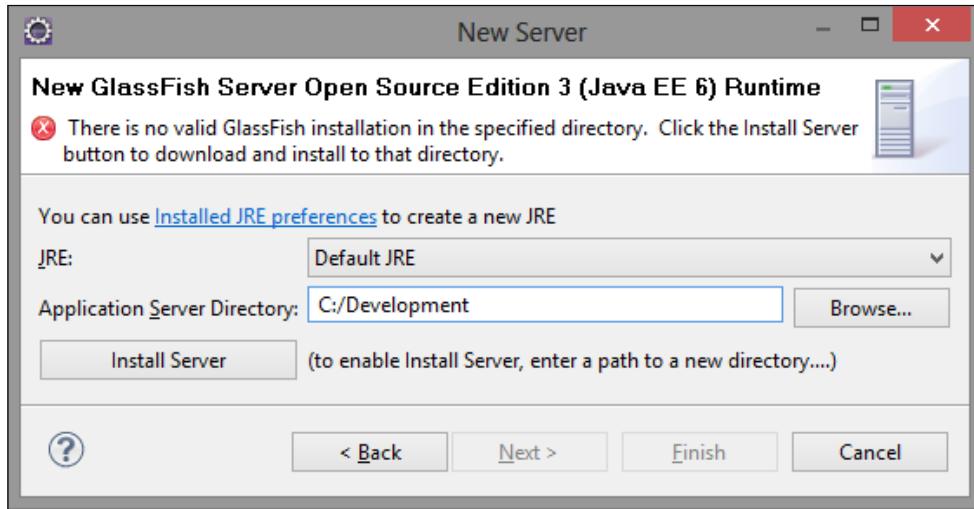
At this point, create a new server as seen in *Chapter 3, Hello Vaadin!*. A new entry is available under the **Glassfish** menu (and not under the Oracle one). Choose **GlassFish Server Open Source Edition 3 (Java EE 6)**.



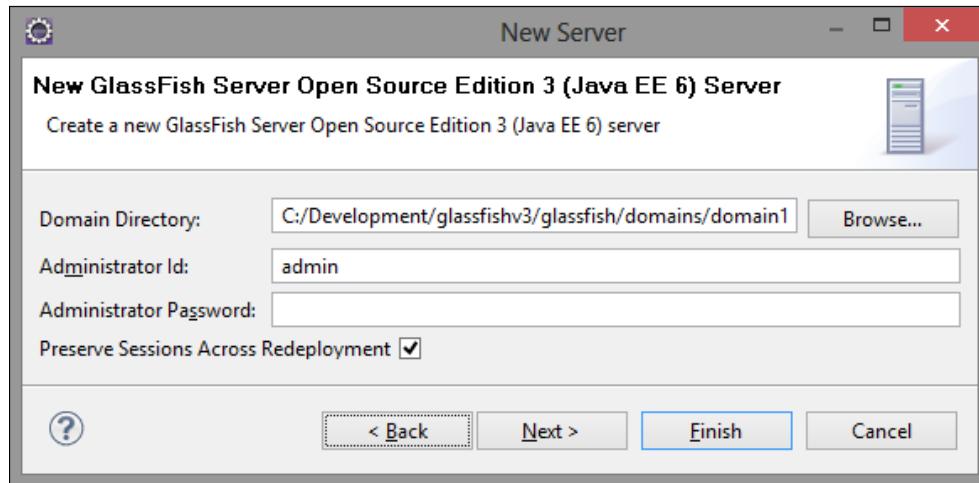
Click on **Next**. Select an installation path to the desired Glassfish directory and click on **Install** (you need to accept the license agreement).



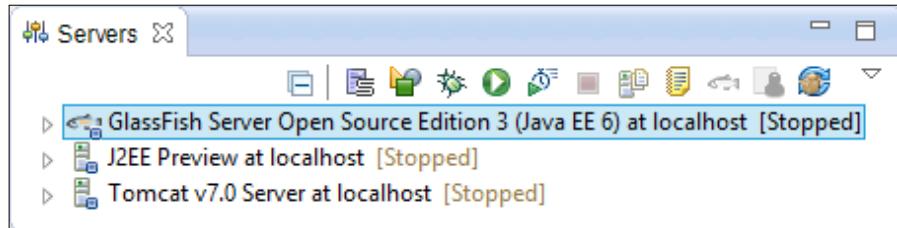
Note that Glassfish will be installed in a `glassfish3` directory inside the chosen folder, so that if you choose `C:/Development`, Glassfish will be installed in `C:/Development/glassfish3`.



Once the server is finished downloading, click on **Next**. The next wizard screen is about configuring the server instance. We are not using it for production purposes, so let us keep the defaults, also including the blank admin password (it will be easier to remember in the future).



Click on **Next** and then **Finish**. Congratulations, a new Glassfish server has appeared in the **Servers** tab, as shown in the following screenshot:



Create the data source

We used our own Vaadin connection in *Chapter 7, Core Advanced Features*. Whereas in this chapter we will use a server provided data source. This way, one can choose one's preferred way to connect to a database.

 In most real-life cases, it is advised to use a server data source, as it decouples the application from specific environments. As a developer, we can then work with the same JNDI resource while letting administrators concern themselves with connection parameters changing from environment to environment.

To begin with, we need to be able to connect to the data source. Since we will reuse H2, we already have the necessary JAR driver. Put it in the <GLASSFISH_HOME>/lib folder. This will make H2 data sources creation available for all domains of the same Glassfish installation.

 You can think of domain as a way to logically partition applications and resources. In this book, we will keep using the same default domain domain1 and will not need this kind of resources management.

Browse to Glassfish admin console, at <http://localhost:4848/>.



Go to **Resources | JDBC | Connection Pools** menu. In the right frame, click on **New**. Fill in the fields as shown in the following screenshot:

- **Name:** Vaadin Pool
- **Resource Type:** java.sql.Driver
- **Database Vendor:** H2

Click on **Next** (at the top-right).

New JDBC Connection Pool (Step 1 of 2) Next Cancel

Identify the general settings for the connection pool.

General Settings * Indicates required field

Name: *

Resource Type: ▼
Must be specified if the datasource class implements more than 1 of the interface.

Database Vendor: ▼ ×
Select or enter a database vendor

In the new screen, the following fields are present:

- **Driver Classname:** org.h2.Driver
- In the **Additional Properties** section:
 - **URL:** jdbc:h2:~/learnvaadin.
 - **user:** SA.
 - **password:** none. Note that this information is necessary, regardless of whether a password has previously been set.

Click on **Finish**. A new pool will appear in the JDBC Connection Pools screen list.

New JDBC Connection Pool (Step 2 of 2) [Previous](#) [Finish](#) [Cancel](#)

Identify the general settings for the connection pool.

General Settings * Indicates required field

Name:	Vaadin Pool
Resource Type:	java.sql.Driver
Database Vendor:	H2
Datasource Classname:	<input type="text"/>
Select or enter vendor-specific classname that implements the DataSource and/or XADataSource APIs	
Driver Classname:	<input type="text"/> org.h2.Driver
Select or enter vendor-specific classname that implements the java.sql.Driver interface.	
Ping:	<input type="checkbox"/> Enabled
When enabled, the pool is pinged during creation or reconfiguration to identify and warn of any erroneous values for its attributes	

Additional Properties (3)

Add Property Delete Properties	
Name	Value
<input type="checkbox"/> URL	jdbc:h2:~/learnvaadin
<input type="checkbox"/> password	none
<input type="checkbox"/> user	SA

[Previous](#) [Finish](#) [Cancel](#)

Once the connection pool is created, we need to make it available under a JNDI name. This is the goal of the **Resources | JDBC | JDBC Resources** menu.

In the opening frame, set fields accordingly:

- **JNDI Name:** `jdbc/persons`; it has to be exactly the same as in the provided `persistence.xml` file
- **Pool Name:** `Vaadin Pool`, as in the previously defined pool

Click on **OK**. A new resource now appears in the resources list screen.

Everything is now ready to go onward.

How-to

JPA itself has many features and some complexity, including a rich lifecycle for managed entities. A full-blown book could very well be dedicated on subject, so we will focus on a simple use-case and see how to make it work.

In this context, we will display a list of persons and their associated job, if any. Using JPAContainer, it can be easily achieved in only two steps as following:

1. To create a `JPAContainer` – a specialized Container fully integrated with JPA, we call the `JPAContainerFactory.make(Class, String)` factory method where `Class` is the JPA enhanced class and `String` is the name of the persistence unit in the persistence descriptor.
 - Given the previous data, this translates to:

```
JPAContainer<Person> container = JPAContainerFactory.make(Person.class, "vaadindemo");
```
2. The previous step only is enough to display persons, with only a single flaw: jobs are shown as their `String` equivalent (for example, `com.packtpub.learnvaadin.jpa.model.Job@xyztuv`) which hints that the `toString()` method is used. With our current knowledge, the first reflex will be to use a column decorator and that will work very well.
 - Yet, `JPAContainer` offers a simpler way to display nested property (for example, a job's label): just call the `addNestedProperty(String)` method, where the parameter is the dotted path to the wanted property.
 - In our example, this will display the job's label:

```
container.addNestedContainerProperty("job.label");
```

That's it! Of course, everything that applies to a `Container` applies to a `JPAContainer`, so the tricks we learned in *Chapter 6, Containers and Related Components* also apply, resulting in the following code:

```
public class JpaScreen extends VerticalLayout {

    public JpaScreen() {

        setMargin(true);

        JPAContainer<Person> container =
            JPAContainerFactory.make(Person.class, "vaadindemo");

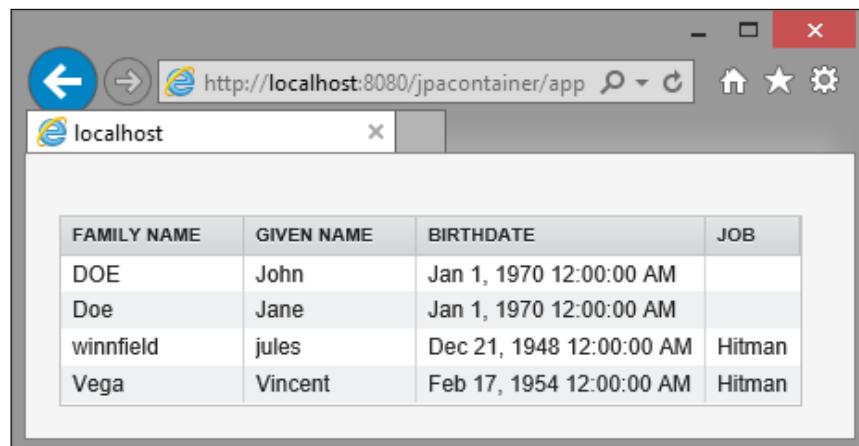
        container.addNestedContainerProperty("job.label");

        Table table = new Table("", container);

        table.setVisibleColumns(new Object[] {
            "lastName", "firstName", "birthdate", "job.label" });
        table.setColumnHeader("firstName", "Given name");
        table.setColumnHeader("lastName", "Family name");
        table.setColumnHeader("job.label", "Job");

        addComponent(table);
    }
}
```

Note that we do not use column names as for `SQLContainer`, but JPA property names. Here is what should be displayed:



Conclusion

SQLContainer is a good path to follow when needing to connect to a database. Yet, when a JPA model is already provided, it is a no brainer to use JPA Container to connect to the JPA backend. Even better, it handles nested properties.

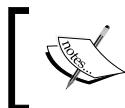
Remember to use this add-on when being in this situation, it is developed and endorsed by Vaadin itself!

CDI Utils

Context and Dependency Injection is a JSR that defines how Dependency Injection takes place in a Java EE compliant application server.

At the moment, Twaatin has come up with coupling issues, meaning classes are directly dependent on one another. The issues are as follows:

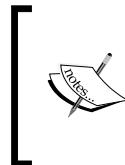
- Screens set their own behavior. It would be nice to be completely independent, in order for screens to be usable with different behaviors in different contexts.
- Likewise, authentication strategy implementation is dependent on the Twitter service. Since the latter is a class (with static methods), unit testing the strategy is impossible.
- Finally, the login behavior is also dependent on the authentication strategy. The former needs to create a new strategy instance.



For a more detailed description of coupling, please see
http://en.wikipedia.org/wiki/Coupling_%28computer_programming%29.



Similar inversion of control (see the following information box) add-ons include ways to integrate the Spring framework: search for Spring Vaadin Integration and Spring Stuff in the directory.



It is the author's opinion that those two add-ons are not production-ready by contrast, brilliant ideas for successful Spring integration can be found at <https://vaadin.com/web/petter/home/-/blogs/experimenting-with-vaadin-spring-and-serialization>.



Let's rework on some of the Twaattin code with CDI to make our components less coupled and more testable in regard to the earlier stated three improvements.

Core concepts

Before diving into how to integrate CDI into our Vaadin applications, a little introduction is in order. Feel free to skip if you already are familiar with CDI.

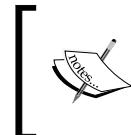
Inversion of Control and Dependency Injection

Inversion of Control and Dependency Injection are notions that are often used interchangeably, but there is a slight difference between them.

Inversion of Control (IoC), is the principle by which a component can get a reference on another component without first instantiating it so there is low coupling between the two.

In Java and Java EE, there are several ways to achieve IoC. Some of them are listed as follows:

- The first way is through the **Abstract Factory** [GOF:87] pattern which is possible in pure Java.
- Another common implementation in Java EE is through the **Service Locator** pattern. In this pattern, the application server instantiates services and makes them available, whereas applications look up for them in order to use them.
- The last option is Dependency Injection.



For more information on Service Locator, visit the following URL:
[http://java.sun.com/blueprints/corej2eepatterns/
Patterns/ServiceLocator.html](http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html)

Dependency Injection (DI), is a specific form of IoC. In DI, both object creation and injection of its dependencies are delegated to a specific part of the system.

DI use cases

DI is used to decouple the building blocks of our application from one another, be they classes, packages, or JARs. DI lets us abstract our dependencies, so we can inject a class in an environment, such as an integration test, and another class in the standard running environment.

This also allows us to stub our dependencies with mock objects to test our class in isolation, for unit testing.

As an example, the Twitter service used in Twaattin is a good use case of using DI. The current code has flaws: the application is tightly coupled to the service, thus preventing unit testing. It is not possible to test the UI without using Twitter, which makes testing difficult at best (and impossible when offline).

Prerequisites

Before diving right into CDI Utils, we need to do a few things.

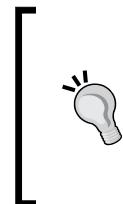
First, we need a Java EE compliant server. The good news is that we already have one fully integrated ever since we worked on JPA Container. If that is not the case, please refer to the *Application server* section.

Then, we just need the dependency as usual. Update the `ivy.xml` settings file like so:

```
<dependency org="org.vaadin.virkki" name="cdi-utils" rev="2.1.1" />
```

How-to

As in any other CDI application, the first step is to create a `beans.xml` file under `WEB-INF` to enable the application for CDI.

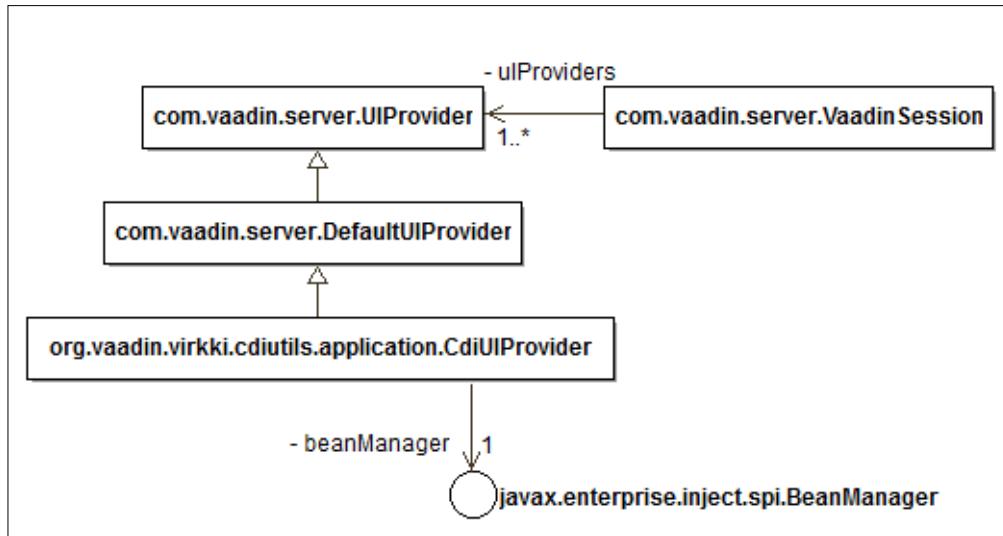


NullPointerException

If you encounter `NullPointerException` when running a CDI application and it seems that beans are not injected, the first thing to check is whether the `beans.xml` file is there (and at the right location, of course).

The second step is to configure a **UI provider**. UI providers are dedicated components responsible for creating UI. The add-on provides the `CdiUIProvider` implementation, which is able to connect to the CDI context itself (or more precisely to the underlying `BeanManager`) and register beans in it.

The following diagram is the UI provider class diagram:



Configuration to use a specific UI provider is done in the web deployment descriptor, as a Vaadin Servlet initialization parameter.

```

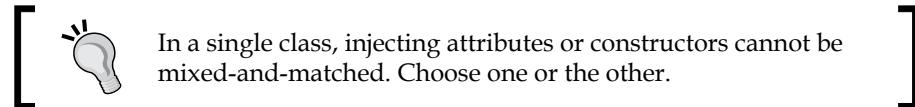
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/
  xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.
  com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>CDI Application</servlet-name>
    <servlet-class>com.vaadin.server.VaadinServlet</servlet-class>
    <init-param>
      <param-name>UIProvider</param-name>
      <param-value>
        org.vaadin.virkki.cdiutils.application.CdiUIProvider
      </param-value>
    </init-param>
    <init-param>
      <param-name>UI</param-name>
      <param-value>com.packtpub.learnvaadin.cdi.ui.CdiUI</param-value>
    </init-param>
  </servlet>
  ...
</web-app>
  
```

The final step is to annotate the UI itself with `org.vaadin.virkki.cdiutils.application.UIContext.UIScoped`. This binds the UI lifecycle management with Vaadin.

```
@UIScoped  
public class CdiUI extends UI {  
    ...  
}
```

CDI injection

In CDI, injection can be achieved either at the attribute or at the constructor level through annotations use.



Instantiation of the needed class, as well its injection at the right injection point, is taken care of by the CDI container.

Taking Twaattin as an example, reworking on the coupling between screen and behavior can be achieved either way by simply using `@javax.inject.Inject` annotation in the desired place.

```
// Attribute injection  
public class LoginScreen extends VerticalLayout {  
  
    @Inject  
    private LoginBehavior behavior;  
  
    private Button submitButton = new Button("Submit");  
  
    public LoginScreen() {  
  
        ...  
        submitButton.addClickListener(behavior);  
    }  
}  
  
// Constructor injection  
public class LoginScreen extends VerticalLayout {  
  
    private Button submitButton = new Button("Submit");
```

```
@Inject  
public LoginScreen(LoginBehavior behavior) {  
  
    ...  
    submitButton.addClickListener(behavior);  
}  
}
```

Both snippets are equivalent: however, note constructor injection has the advantage of not storing the behavior instance as an attribute.

In Twaattin original code, the PIN field has to be injected in both screen – to display it, and behavior – to get its value. Unfortunately, this is not something that should be done in CDI and we have to use another Java EE feature, though decoupling strategy/service and behavior/strategy can be achieved using this approach.

Event observers

Along with CDI comes a neat feature of Java EE 6 containers, event-listener support as follows:

- `javax.enterprise.event.Event<?>` is a dedicated component to fire events of type `T`, with no requirement on type `T`. Instances of this component have to be injected by CDI.
- Methods interested in those events – observers, have to take `T` as a single parameter and be annotated as `javax.enterprise.event.Observes`.

Login processing is a perfect fit for event subscription: when the user logs in, an event is fired, wrapping all necessary input values. Login behavior listens to those events and acts in regard to values sent.

In order to implement this in Twaattin, we need to create an event type to wrap the PIN value:

```
package com.packtpub.learnvaadin.cdi.presenter;  
  
public class LoginEvent {  
  
    private final String pinValue;  
  
    public LoginEvent(String pinValue) {  
        this.pinValue = pinValue;  
    }  
  
    public String getPinValue() {  
        return pinValue;  
    }  
}
```

The next step is to wire Vaadin user-generated events to CDI events. Login screen has to be updated to listen to the former as well as the latter:

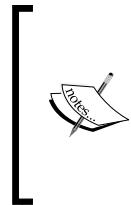
```
public class LoginScreen extends VerticalLayout implements  
ClickListener {  
  
    @Inject  
    private javax.enterprise.event.Event<LoginEvent> eventManager;  
  
    private Button submitButton = new Button("Submit");  
    private TextField pinField = new TextField();  
  
    @Inject  
    public LoginScreen() {  
  
        ...  
  
        submitButton.addClickListener(this);  
    }  
  
    @Override  
    public void buttonClick(ClickEvent event) {  
  
        LoginEvent loginEvent =  
            new LoginEvent(pinField.getValue());  
  
        eventManager.fire(loginEvent);  
    }  
}
```

Finally, the behavior class has to listen to those CDI events:

```
public class LoginBehavior {  
  
    public void handleLogin(@Observes LoginEvent event) {  
  
        ...  
    }  
}
```

Components declaration

CDI Utils also provides a way to inject components declaratively instead of instantiating them programmatically.



Just because it is possible to achieve something, this does not mean that it has to be done. In particular, injection is used to break dependencies and thus, make code more changeable and testable. It is the author opinion that injecting UI components into other UI components does not bring any of those qualities. You may have another point of view, so here is how you can do that.

This feature is only available for Vaadin out-of-the-box components that are injected. Usage is as follow: annotate the same injection point as `@Inject` with `@Preconfigured`. Valid attribute are equivalent of `AbstractComponent` properties.

For example, the following two snippets do the same:

```
// Programmatic way
private TextField textField = new TextField();

// Somewhere after (e.g. in the constructor)
textField.setCaption("First name");
textField.setImmediate(true);

// Declarative way
@Inject
@Preconfigured(caption = "First name", immediate = true)
private TextField textField;
```

Conclusion

CDI Utils is currently "the" way to use Vaadin and CDI together. Note that it will be replaced by CDI Integration, which is now in alpha and does not allow for component injection. Complete sources for previous add-on examples can be found at <https://github.com/nfrankel/cdiutils>.

Summary

In this chapter, we detailed how Vaadin can be enriched with additional features coming from the Vaadin directory add-ons and how we could search the latter.

In particular, we had a look at these particular add-ons:

- Button group is a simple example of UI add-on. We had a look at what was mandatory to use such graphical add-ons.
- Although experimental, Clara is interesting because it let us declare UI with XML instead of programming them.
- Most applications need at least a data backend and Vaadin-based ones are not different. The JPA Container add-on is an easily configurable adapter allowing us to wrap a JPA in a Vaadin table.
- Finally, CDI Utils brings the power of Context and Dependency Injection into the Vaadin realm.

These are only a small sample of all available add-ons. It is simply not feasible to list them all! It is advised to have a look at the directory regularly to know about the latest published add-ons. Alternatively, one can also subscribe to the @Vaadin Twitter channel at <https://twitter.com/vaadin> where new releases are automatically published.

Now that we have the know-how regarding integration of third-party add-ons, now would be a good time to learn how to develop our own, or more precisely to be able to be independent in case no add-on really applies to some use-cases. This is exactly what the next chapter proposes to describe.

9

Creating and Extending Components and Widgets

Although Vaadin provides many great components out-of-the-box, we sometimes need to go further. We saw in the previous chapter how we could extend Vaadin with provided add-ons. Now is the time to learn how to extend it on our own.

Extending Vaadin is a feature designed into the framework itself. In this chapter, we'll have a look at different ways to achieve this:

- Creating custom component in Vaadin by composing them from other components
- Extending existing widgets on the client side
- Wrapping existing GWT widgets into Vaadin components
- Wrapping existing JavaScript into Vaadin components

Learning these techniques will get you a long way towards getting the best out of Vaadin.

Component composition

Composing components can either be very simple or a daunting task, depending on the number of components involved.

Components composition is by far the easiest way (compared to other techniques) to provide custom-tailored components. However, this fits only limited use-cases, for example, to reuse some application GUI part. The keyword here is being reusable. Examples of such component include:

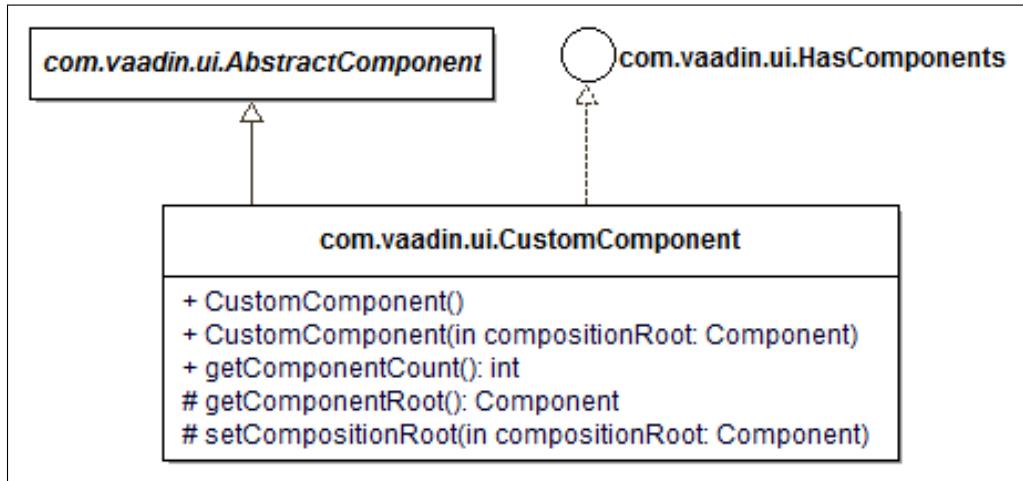
- An address component, with address lines, state, zip code, and city
- A yes/no/cancel options dialog, similar to the ones found in Swing
- A screen template, including a top menu and a left bar
- A reusable menu among all the screens of an application

The following screenshot is an example of a reusable dialog box:



Manual composition

The root of custom component composition in Vaadin is the `CustomComponent` class, which inherits `AbstractComponent` and implements `HasComponents`.



Adding components to the custom component is just a matter of:

- Setting the composition root to a container, probably a layout so as to add more than one subcomponent
- Adding components to the root



An important limitation of this approach is that we have to basically create at least two classes: one being the custom component, playing the role of the wrapper, and the other being the topmost component, the content container. Such containers can be panels or whatever, so long as we can reuse them.

This is it! There's no fancy API: we just add the child widgets to our custom component's root and we are good to go.

As an example, let's create a reusable confirm dialog, just like the one seen previously. It would look something like the following code:

```
import com.vaadin.ui.Button;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.Button.ClickListener;
import com.vaadin.ui.CustomComponent;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;
import com.vaadin.ui.VerticalLayout;
import com.vaadin.ui.Window;

@SuppressWarnings("serial")
public class CustomDialog extends Window {

    private String clickValue;

    public CustomDialog() {
        setContent(this.new CustomDialogContent());
    }

    public ClickValue getClickValue() {
        return clickValue;
    }

    private class CustomDialogContent extends CustomComponent {
```

```
private Label message = new Label("This is a confirm  
dialog.");  
private Button yesButton = new Button("Yes");  
private Button cancelButton = new Button("Cancel");  
private Button noButton = new Button("No");  
  
public CustomDialogContent() {  
  
    VerticalLayout mainLayout = new VerticalLayout();  
  
    mainLayout.setMargin(true);  
    mainLayout.setSpacing(true);  
  
    setCompositionRoot(mainLayout);  
  
    mainLayout.addComponent(message);  
  
    HorizontalLayout buttonBar = new HorizontalLayout();  
  
    buttonBar.setSpacing(true);  
  
    mainLayout.addComponent(buttonBar);  
  
    Button[] buttons =  
        new Button[] { yesButton, noButton, cancelButton };  
  
    for (final Button button : buttons) {  
  
        buttonBar.addComponent(button);  
        button.addClickListener(new ClickListener() {  
  
            @Override  
            public void buttonClick(ClickEvent event) {  
  
                clickValue = button.getCaption();  
  
                UI.getCurrent().removeWindow(CustomDialog.this);  
            }  
        });  
    }  
}
```

As stated before, we have two component classes: `CustomDialogContent` for the content and `CustomDialog` for the dialog itself.

Designing custom components

The reason behind using custom components is reuse. As such, a well thought-out design is a must-have in those cases.

In the preceding code, this is very straightforward: we provided the `getClickedValue()` method to return the value clicked. And yet, there are some ways for improvement:

- Is `String` the right return type? Or should we provide an `enum` to carry the return type?
- Neither text for the buttons nor dialog message can be customized. For buttons, it prevents internationalization, but a hard-coded message prevents reuse altogether!
- Shouldn't the cancel button be optional?
- Providing only `CustomDialogContent` and let users wrap it in their own window will improve reusability.

In practice, reusable component design should be constrained by two major concerns: providing sensible defaults, but enabling parameterization at the same time so that the component can really be reused in a variety of contexts.

Apart from that, only experience will tell you how much effort you should invest in your personal case.

Graphic composition

Previous sections explained about manual coding of custom components to meet our needs.

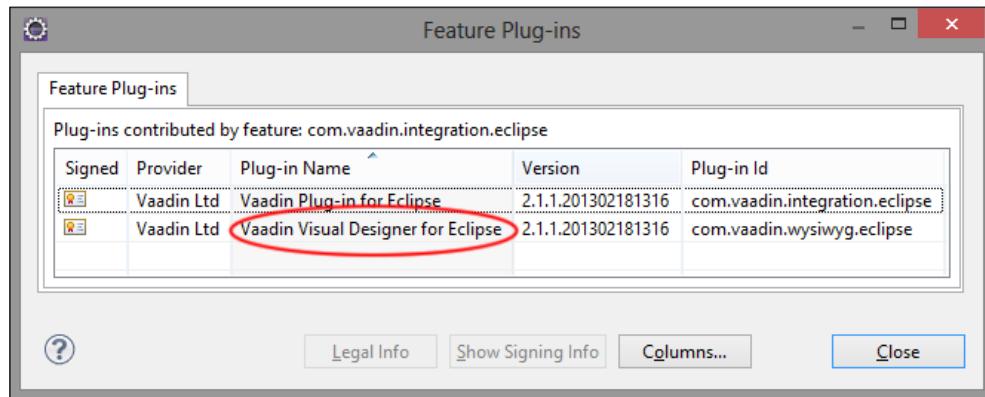
Nonetheless, the easier way to create a custom component by assembling other ones is to do it graphically. It's a nice thing then, that the Vaadin Eclipse plugin includes a graphical editor, named the **Visual Designer**.

Visual editor setup

Vaadin's graphical editor is bundled within the plugin. To check that it is installed, go to **Help | About Eclipse** and click on the Vaadin features icon.



In the opening pop-up, the Vaadin plugin should appear. Click on the **Plugin details** button.



If everything is in order, the Visual Designer plugin should be listed in the list.

Visual Designer use

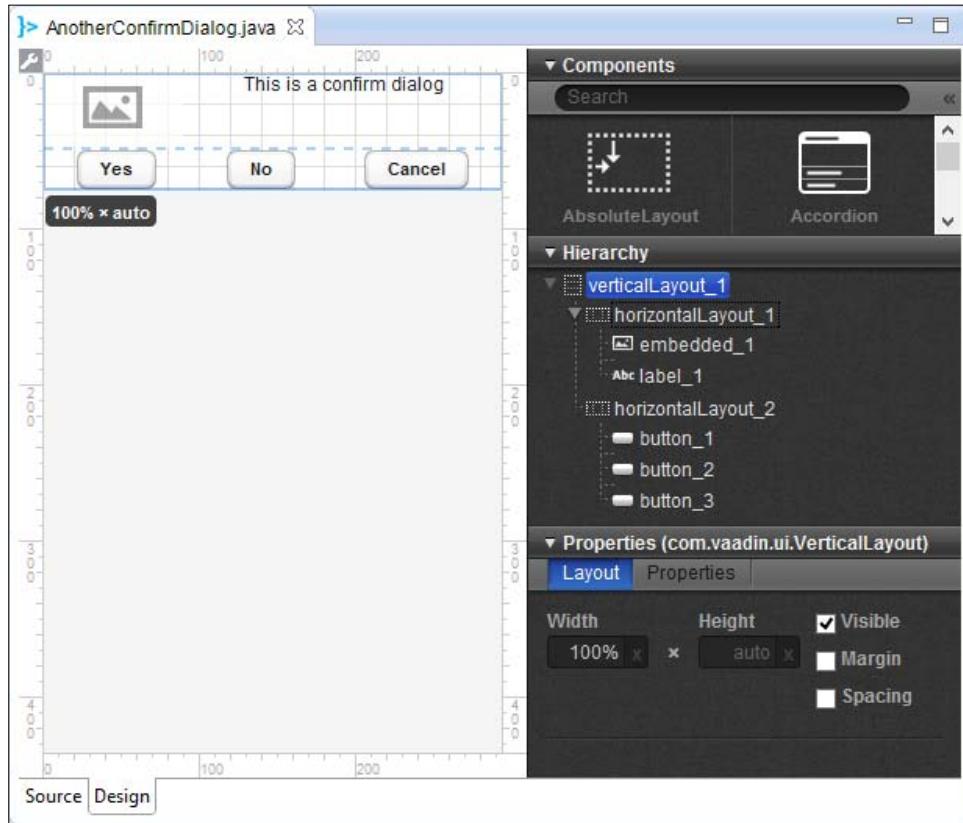
Now, just go the **File** menu and click on **New**.

Then choose **Vaadin | Vaadin Composite**. It immediately opens the standard Java code viewpoint where there are two tabs underneath, a code tab selected by default, and a design tab.

Selecting the latter will display the "true" graphical editor. It is separated in two main parts:

- A canvas with a gridline occupies the main space.
- A vertical bar on the left displays the following three sections:
 - A component palette showing all available components.
Remember that layouts are components too (albeit special ones).
 - The custom component hierarchy.
 - Available properties of the currently selected component.

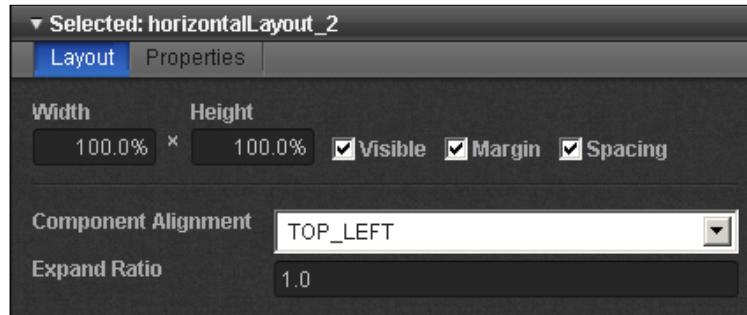
Components from the palette can be dragged-and-dropped either to the canvas or to the component hierarchy.



A full guide to the Visual Designer is beyond the scope of this book, but there are nonetheless some guidelines that can save us a considerable amount of time.

Position and size

The first tab for a selected component is the layout one. It represents properties that manage position and size.



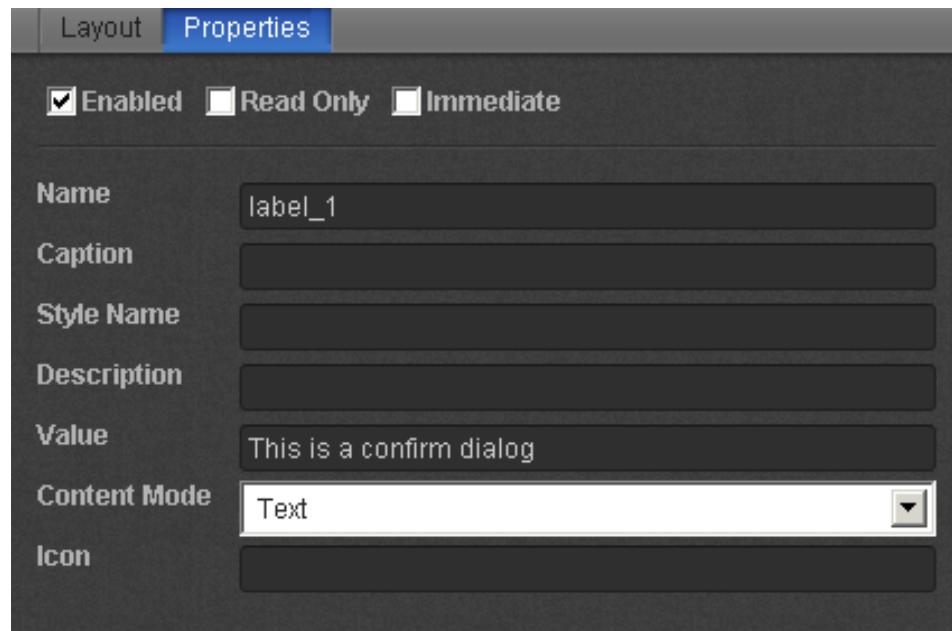
Available properties are:

- **Width** and **Height**. Those correspond respectively to the `setWidth()` and `setHeight()` methods of the `Sizeable` interface, and accept the same values. In order to use `setXXXUndefined()`, set value to "auto".
- **Visible**, **Margin**, and **Spacing** are standard properties seen in *Chapter 4, Components and Layouts*. Note that setting different margins on different borders is not possible with the VD.
- **Component Alignment** manages the position of the element in its available space.
- **Expand Ratio** refers to the space provided to the component when space has to be shared between multiple components.

On the design canvas, we can also see a small rectangle when selecting a component. It's a shortcut to changing size and position.



The second tab displays every property not shown on the first tab, and is dependent on the component's type.



For example, the preceding screenshot presents the available properties for a label.

Limitations

Note that the editor is an important tool for designing Vaadin applications and deemed as such by Vaadin's team. It has some limitations, though.

Restricted compatibility

The most important limitation of the Visual Designer in its current state is its inability to display existing components that were not initially designed with the VD.

Top-level element

Moreover, the top level element designed in the VD can only be a layout. By default, it's `AbsoluteLayout`. If it has to be changed, we cannot graphically update the type: we have to update the code itself manually.

This prevents us from creating reusable screen, complete with title. The biggest reusable unit is the screen's **content**.

Rigid structure

The generated code is set to some pattern that the editor expects. This has three important consequences:

- Every component is declared as an attribute and assigned a `@AutoGenerated` annotation
- It is instantiated and configured in its own private builder method that is called in the constructor
- Custom non-generated code, if it exists, has to take place after those calls in the constructor

Any change to these will likely prevent Vaadin from opening the Visual Designer or have unwanted side effects. Don't do that!

Client-side extensions

Vaadin's greatest advantage is that we code everything in Java, which means important stuff is executed on the server-side. However, some events are better handled client-side.

For example, picture a requirement to display a tool tip when the cursor hovers over a field. Going server-side would require each mouse move to send an event to the server to check whether the mouse is over the component: a sure way to saturate the network and kill scalability. Another common use-case is the need to tabulate between fields in a predefined order.

Those are clearly use-cases better handled client-side.

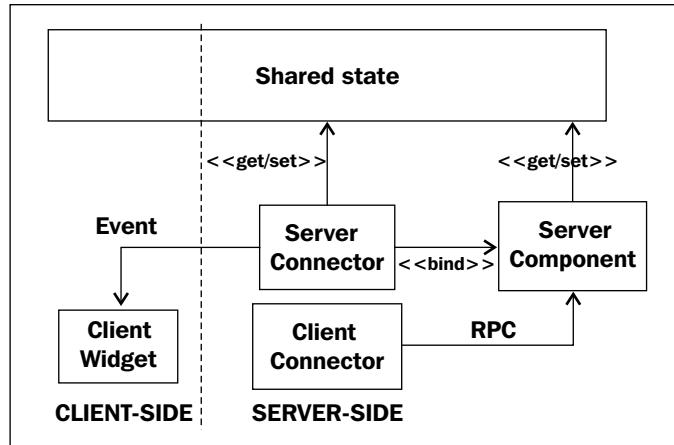
Connector architecture

Before going further, a more detailed understanding of client-server communication in Vaadin 7 is in order.

Vaadin 7 introduces the concept of connector. Connectors are code bits that bind between client-side widgets and server-side components.

- Client connectors provide a way to manage widgets and make RPC calls to components
- Server connectors offer event firing features

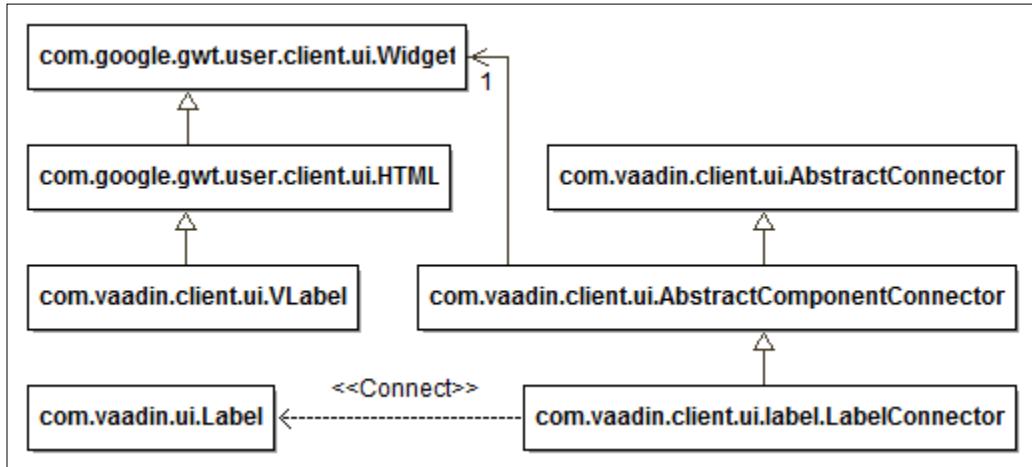
- State is managed in a so-called shared state class that exists on both sides. Synchronization between those is handled transparently by the Vaadin framework.



[ Note that connectors hold references to their respective client and server classes, not the other way around. This enables working with existing components and widgets, as seen in the following section!]

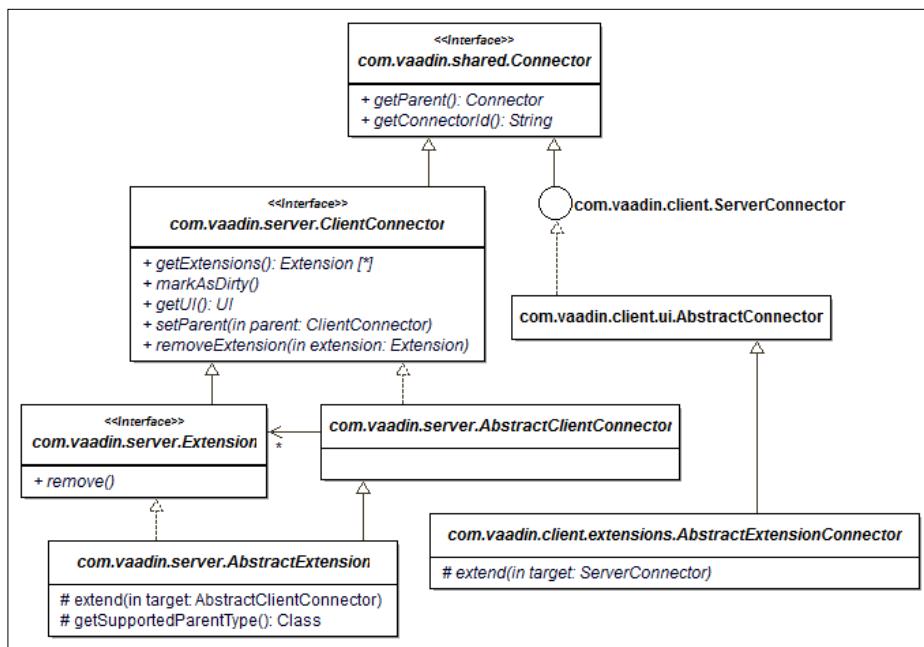
Let us study a simple existing Vaadin class with our newfound knowledge, `Label`. We already know about Component hierarchy. What about the connector one?

`LabelConnector` is the corresponding connector. It holds a reference on the `VLabel` widget. On the other side, it also binds the `Label` component through an annotation.



The only difference with the extension hierarchy is that component connector provides capabilities by itself, while extension connector is only meant to supplement those coming with the attached widget.

Here is the class diagram of the full-blown extension connector architecture:



How-to

Extension is exactly that: an attachment for an existing widget, with independent feature, and communication between client and server. Vaadin provides a root class for our custom extensions through the `AbstractExtension` class. It is only designed to restrict types the extension can be attached to.

Extensions, like any other Vaadin component, must be provided a client connector. Creating a custom extension requires both.

As an illustration, we will create a button that prevent multiple submits. It gets disabled and asks the user to wait when clicked.



The following is to be taken as an example. Vaadin provides such a feature in the form of the `setDisableOnClick(true)` method in the `Button` class, although caption does not change.

The first step is to create the extension proper, to restrict adding it on Button types only. Easy as pie:

```
import com.vaadin.server.AbstractExtension;
import com.vaadin.ui.Button;

@SuppressWarnings("serial")
public class DisableOnClickButtonExtension extends AbstractExtension {

    protected void extend(Button button) {

        super.extend(button);
    }
}
```

The second step is where the magic takes place:

```
import static com.vaadin.client.ApplicationConnection.DISABLED_
CLASSNAME;

import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.packtpub.learnvaadin.DisableOnClickButtonExtension;
import com.vaadin.client.ComponentConnector;
import com.vaadin.client.ServerConnector;
import com.vaadin.client.extensions.AbstractExtensionConnector;
import com.vaadin.client.ui.VButton;
import com.vaadin.shared.ui.Connect;

@SuppressWarnings("serial")
@Connect(DisableOnClickButtonExtension.class)
public class DisableOnClickButtonConnector extends
AbstractExtensionConnector {

    @Override
    protected void extend(ServerConnector target) {

        final VButton button =
            (VButton) ((ComponentConnector) target).getWidget();

        button.addClickHandler(new ClickHandler() {

            @Override
            public void onClick(ClickEvent event) {
```

```
        button.setEnabled(false);
        button.addStyleName(DISABLED_CLASSNAME);
        button.setText("Please wait...");
    }
}
}
}
```

Here, we have to use the GWT API itself:

1. We get a reference on the GWT widget to add a GWT event handler.
2. In the handler code, we disable the button and set the desired text. Also, since Vaadin buttons do not render as HTML buttons but as `div`, we also need to set a specific style name to render the button as disabled.

[ This book is not about GWT. If more detailed understanding than provided is needed, please see Google documentation at <http://code.google.com/webtoolkit/>.]

Finally, we have to compile client-side code. This is easily achieved by clicking on the **Compile Vaadin widgets** button, as seen as in the *Button group* section of *Chapter 8, Featured Add-ons*. This will create a `gwt.xml` file, update the web deployment descriptor, and compile the client-side properly.

[ **Package structure**
The extension connector class should be located in a `client` subpackage, relative to the extension class. Extension and the `gwt.xml` file should be in the same package. If those requirements are not met, nothing will happen, but there will be no error shown.]

This is it: now, when the button is clicked, it is disabled while its caption is updated.



The code source for this secure submit extension first version can be found online at <https://github.com/nfrankel/client-extension/tree/v1.0-simple-extension>.

At present, there is a slight drawback: it only shows a single label when pressed. This completely defeats reusability and internationalization. We need to add a feature to enable disabled text configuration.

Shared state

Shared state is a way for client widget and server component to share state that is, common attribute values, hence its name. At runtime, a shared state object for each connector will be present in both the JVM and the JavaScript container. Vaadin takes care of keeping them in synch, so that state changes on one side will be reflected on the other side.

This object is available on both sides through the `getState()` method, available on connector and extension (and component). This method has just to be overridden to return the right shared state class.

How-to

Shared state classes should extend `com.vaadin.shared.communication.SharedState`, although components should use the more specialized `com.vaadin.shared.AbstractComponentState` class instead.

 Since shared states are compiled into JavaScript, they have to be present in a `client` subpackage.

In our case, we want to be able to configure the disabled label, so the shared state should be very simple, with a single attribute.

```
import com.vaadin.shared.communication.SharedState;

@SuppressWarnings("serial")
public class DisableOnClickButtonSharedState extends SharedState {

    private String disabledLabel;

    public String getDisabledLabel() {
        return disabledLabel;
    }

    public void setDisabledLabel(String disabledLabel) {
        this.disabledLabel = disabledLabel;
    }
}
```

Then we need to override the `getState()` method, client-side and server-side to return our new class: a no-brainer.

```
@Connect(DisableOnButtonClickExtension.class)
public class DisableOnButtonClickConnector extends
AbstractExtensionConnector {

    @Override
    public DisableOnButtonClickSharedState getState() {

        return (DisableOnButtonClickSharedState) super.getState();
    }

    ...
}
```

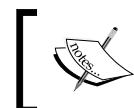
We also need to provide the label configuration feature in the extension.

```
public class DisableOnButtonClickExtension extends AbstractExtension {

    public DisableOnButtonClickExtension(String disabledLabel) {

        getState().setDisabledLabel(disabledLabel);
    }

    ...
}
```



Notice we designed the disabled label as an immutable property, so it cannot be changed afterwards. It is clearly a design choice-if needed, just create an adequate setter.



The last thing is to access the state when the button is pressed, and set the returned label as the button text.

```
@Override
public void onClick(ClickEvent event) {

    button.setEnabled(false);
    button.addStyleName(DISABLED_CLASSNAME);
    button.setText(getState().getDisabledLabel());
}
```



Note the `addStyleName()` call. This is needed because Vaadin buttons are not native HTML buttons. Disabling a button will not render it as visually disabled (although clicking on it will have no effect), hence the needed CSS class.

From this point on, the disabled label can be configured when the extension is instantiated and it will be shown when the button is clicked! The code source for this enhanced version is available at <https://github.com/nfrankel/client-extension/tree/v1.1-shared-state-implementation>.

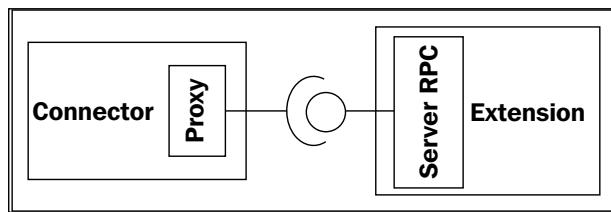
Curious readers may have tried to change the code above in order to reset the button state from the server side. Unfortunately, this does not work. The reason is that component state is managed server-side, so from a Vaadin point-of-view, the button is not disabled. We can with a little effort fix the behavior, by using **RPC** calls.

Server RPC

RPC stands for Remote Procedure Calls and is a very old computer science feature. This will let you immediately disable the button widget client-side as well as call the server to update its counterpart component.

Server RPC architecture

Extensions can expose callable points in the form of server RPC. Such RPC is just an interface that has to extend `ServerProxy`, which is a marker interface (an interface without methods).



On the client side, connectors just need to create a proxy over this RPC. Vaadin will transparently carry calls to the proxy from the connector to the server, hence the RPC name to it.

How-to

In our case, the RPC should disable the button and set its label. The RPC interface looks like:

```
import com.vaadin.shared.communication.ServerRpc;

public interface DisableOnClickButtonRpc extends ServerRpc {

    void disableButton(String disabledLabel);
}
```

Troubleshooting

Note that since the RPC class will be proxied, it has to be in a client subpackage to be compiled. As ever, if not the case, it will fail silently, but surely.

The extension has to wrap such an implementation and register it (that is, listen to RPC calls made by the client-side proxy).

```
import com.vaadin.shared.communication.ServerRpc;
...

@SuppressWarnings("serial")
public class DisableOnClickButtonExtension extends AbstractExtension {

    private Button button;

    private ServerRpc rpc = new DisableOnClickButtonRpc() {

        @Override
        public void disableButton(String disabledLabel) {

            button.setCaption(disabledLabel);
            button.setEnabled(false);
        }
    };

    public DisableOnClickButtonExtension(String disabledLabel) {

        registerRpc(rpc);
        getState().setDisabledLabel(disabledLabel);
    }

    protected void extend(Button button) {
```

```
    this.button = button;
    super.extend(button);
}
...
}
```

Note that we need to get a handle on the button to change its state. This is easily achieved in the `extend(Button)` method already available to us.

The client side also creates a server RPC but this time through a factory method returning a proxy. Then we call the only method of the interface and watch the magic happen on the server side.

```
import com.vaadin.client.communication.RpcProxy;
...
@SuppressWarnings("serial")
@Connect(DisableOnClickButtonExtension.class)
public class DisableOnClickButtonConnector extends
AbstractExtensionConnector {

    private DisableOnClickButtonRpc rpc =
        RpcProxy.create(DisableOnClickButtonRpc.class, this);

    ...
    button.addClickHandler(new ClickHandler() {
        @Override
        public void onClick(ClickEvent event) {
            String disabledLabel = getState().getDisabledLabel();
            button.setEnabled(false);
            button.addStyleName(DISABLED_CLASSNAME);
            button.setText(disabledLabel);

            rpc.disableButton(disabledLabel);
        }
    ...
})
}
```

This time, when we query for the button status, it returns disabled (as well as the right caption).

Full sources for this final version can be found online on GitHub at
<https://github.com/nfrankel/client-extension/tree/v1.2-server-rpc>.



Beware that for event queuing reasons, querying for status in a click listener attached to the button itself may not return the correct status. It has to be done in another component (to give the server some time to handle the RPC call).

GWT widget wrapping

Another way to create custom components in Vaadin is to wrap GWT widgets under a thin Vaadin layer. This is the way some out-of-the-box Vaadin components are themselves provided.

Vaadin GWT architecture

Vaadin GWT architecture is based on two foundations: client-side and server-side.

How-to server-side

On the server-side, things are like the former extension. We need to create a class implementing `Component`, though it is well-advised to extend `AbstractComponent` for Vaadin takes care of much boring code for us.

In the case of state sharing, we have to override the `getState()` method to narrow the return state type.

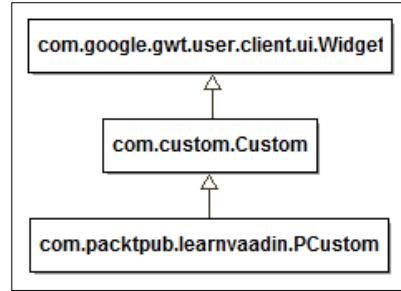
How-to client-side

On the client-side, there are no requirements for the wrapped widget, beyond inheriting from `com.google.gwt.user.client.ui.Widget`!

Here's an example of the widget hierarchy:

1. At the hierarchy top lies `Widget`, from the GWT framework itself.
2. Some third-party library provides a widget we want to use within our application. There can be as many inheritance levels from this widget to `Widget`.
3. Optional: subclass the former if we need to add custom code, server RPC, for instance.

In this case, as a convention, Vaadin widgets are prefixed with **V** for "Vaadin". It is advised to have your own prefix for widgets, generally taken from your company: we will use the **P** as in "Packt Publishing".

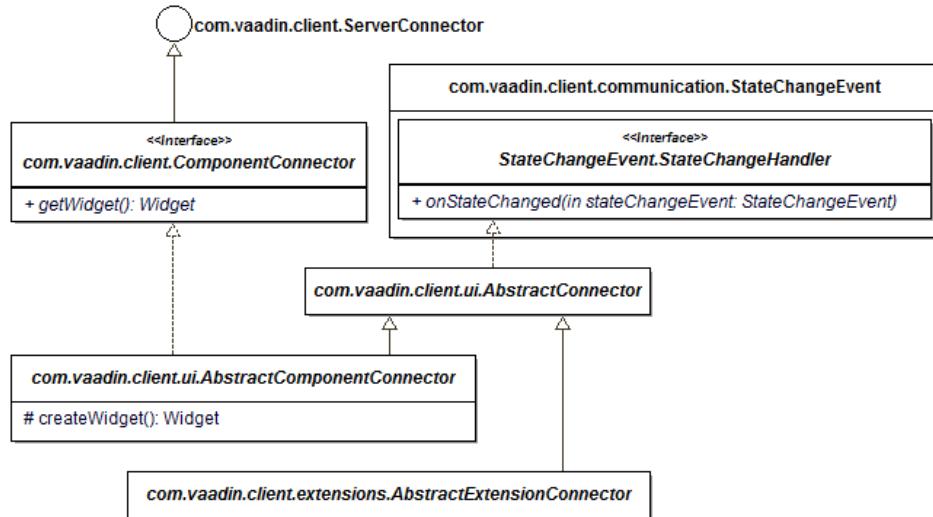


When relevant, connector, shared state, and server RPC also have to be developed client side, as for extensions. Refer to the appropriate sections if needed, since it behaves exactly in the same way.

There is a slight difference in the connector part, though. Whereas extensions inherited from `AbstractExtensionConnector`, full-fledged widgets have to extend `AbstractComponentExtension`.

In particular, concrete component connectors have to implement:

- `createWidget()` to create the new widget instance
- `getWidget()` to narrow the returned type

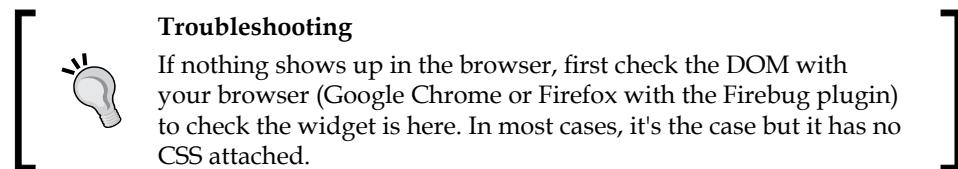


Widget styling

Style for unknown widgets cannot be inferred by Vaadin. This means we have to provide a CSS for them, regardless of the current theme. Theming can be used to customize the widget base look for a particular theme.

GWT offers a way to reference CCS: it can be named however we like, but has to be present in a `public` directory and referenced in the `widgetset.gwt.xml` file like so:

```
<stylesheet src="stylesheet.css" />
```



Example

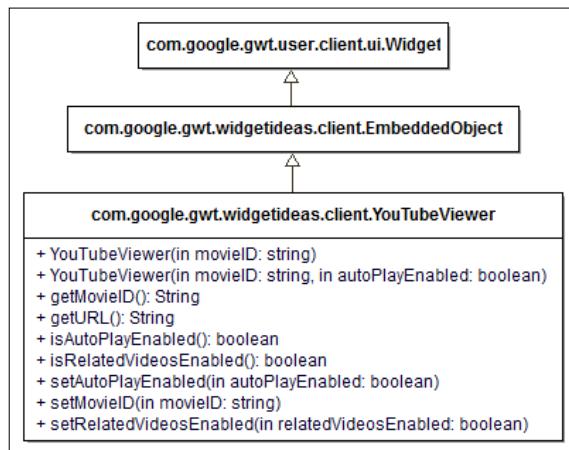
As an example, we will wrap an existing GWT widget. The **GWT Incubator** project offers one such widget, `YouTubeViewer`.

Like its name implies, it provides a way to display the chosen video on YouTube.



The Incubator project has not been released since 2010, and the viewer itself is deprecated. Nonetheless, it makes for a simple illustration.

In order to go further, we just have to detail the existing widget to be wrapped.



The widget provides some attribute for customization: to make things simpler, we will just use the movie ID.

Prerequisites

Prerequisites are like for any standard Java libraries in Java web applications:

1. Download the JAR library at <http://google-web-toolkit-incubator.googlecode.com/files/gwt-incubator-20101117-r1766.jar>.
2. Put it in the WEB-INF/lib folder of your project to make it accessible.

Server component

For the server component, we just have to extend `AbstractComponent`, as stated above, and wire the right shared state class. We also need a way to provide the video ID. From a design point-of-view, this can be achieved through a setter, a constructor, or both shown as follows:

```
import com.packt.learnvaadin.gwt.client.YouTubViewerState;
import com.vaadin.ui.AbstractComponent;

@SuppressWarnings("serial")
public class YouTubeViewer extends AbstractComponent {

    public YouTubeViewer() {}

    public YouTubeViewer(String movieId) {

        setMovieId(movieId);
    }

    @Override
    protected YouTubViewerState getState() {

        return (YouTubViewerState) super.getState();
    }

    public void setMovieId(String movieId) {

        getState().setMovieId(movieId);
    }
}
```

This is all that is mandatory in the server component class.

Client classes

On the client side, we need to create the shared state used previously. As stated, we will only use the movie ID. We already did that in the extension section:

```
import com.vaadin.shared.AbstractComponentState;

@SuppressWarnings("serial")
public class YouTubViewerState extends AbstractComponentState {

    private String movieId;

    public String getMovieId() {
        return movieId;
    }

    public void setMovieId(String movieId) {
        this.movieId = movieId;
    }
}
```

Next, let us create the connector:

```
import com.google.gwt.user.client.ui.Widget;
import com.google.gwt.widgetideas.client.YouTubeViewer;
import com.vaadin.client.communication.StateChangeEvent;
import com.vaadin.client.ui.AbstractComponentConnector;
import com.vaadin.shared.ui.Connect;

@SuppressWarnings({ "serial", "deprecation" })
@Connect(com.packt.learnvaadin.gwt.YouTubeViewer.class)
public class YouTubeViewerConnector extends AbstractComponentConnector
{

    @Override
    public YouTubViewerState getState() {

        return (YouTubViewerState) super.getState();
    }
}
```

```
    @Override
    public YouTubeViewer getWidget() {

        return (YouTubeViewer) super.getWidget();
    }

    @Override
    protected Widget createWidget() {

        return new YouTubeViewer("");
    }

    @Override
    public void onStateChanged(StateChangeEvent stateChangeEvent) {

        super.onStateChanged(stateChangeEvent);

        String movieId = getState().getMovieId();

        getWidget().setMovieID(movieId);
    }
}
```

As opposed to the shared state, there are several comments to make on the connector code:

- Binding the connector to the server component is achieved through the `@Connect` annotation, as for extensions
- On the contrary, binding the client widget is made through an implementation of the `createWidget()` method
- Finally, notice we overrode the `onStateChanged()` method so that when state changes, that is. when the movie id is set on the server-side, the widget is set the same id and displays the desired movie

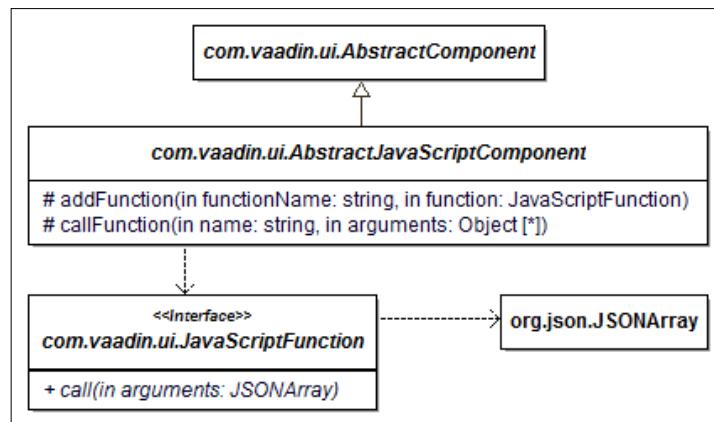
The source code for this example is available at <https://github.com/nfrankel/gwt-integration>.

The final result looks somewhat like the following screenshot: a Youtube player with a field for the movie ID. When the field value changes, the player displays the new movie (if the ID is valid).



JavaScript wrapping

The previous section taught us about GWT widget wrapping, but what to do when there is no GWT widget available, but plain JavaScript? Do we need to first wrap the JavaScript in a custom made widget? That was the case for version 6 of Vaadin, but version 7 takes care of that.



How-to

On the server-side, there are a couple of requirements:

1. The component class must inherit from `AbstractJavaScriptComponent`.
2. It must be annotated with `@Javascript`. This annotation takes an array of all JavaScript files needed for value. JavaScripts can be either referenced by an absolute URL or relative to the component class. In the latter case, it is much simpler to put them in the same package as the component to avoid unnecessary hassle and directly reference them by their name (including `.js` extension).
3. Finally, we also need a `gwt.xml` file referencing at least Vaadin default widget set. It has to be set as an `init-param` in the web deployment descriptor, as before.

On the client-side, a single JavaScript is to be Vaadin entry-point; it must have the following structure:

```
window.server_package_with_underscore_for_separator = function() {  
  
    ...  
  
    this.onStateChange = function() {  
  
        ...  
    }  
}
```

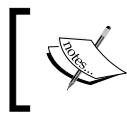
The `onStateChange` function will be called at initialization, so this is where we need to put code that we want to be executed at that time, the rest can be put in other scripts.

Beware that since `AbstractJavaScriptComponent` also may have shared state (refer to the section *Shared state*), this method will also be called when state changes. State class to inherit is then `com.vaadin.shared.ui.JavaScriptComponentState`.

Example

We are going to display a simple 3D scene with Vaadin using **WebGL**. First of all, we need to check our browser is compatible (see <http://caniuse.com/webgl>).

WebGL itself being kind of low-level, we are going to use `Three.js`, a JavaScript library.



This book is neither about WebGL nor Three.js and both require dedicated books. For more information, go respectively to <https://www.khronos.org/webgl/> and <http://threejs.org/>.



Prerequisites

Download the library from <http://github.com/mrdoob/three.js/zipball/master> and extract the build/three.min.js.

Core

We need a server component:

```
package com.packtpub.learnvaadin.js;

import com.vaadin.annotations.JavaScript;
import com.vaadin.ui.AbstractJavaScriptComponent;

@SuppressWarnings("serial")
@JavaScript({
    "https://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquery.min.js",
    "three.min.js",
    "scene.js"
})
public class ThreeJs extends AbstractJavaScriptComponent {

}
```

It is quite simple. Notice we did not need to download JQuery, we used the Google provided online instance. The scene.js will be our entry point, and is the next step.

```
window.com_packtpub_learnvaadin_js_ThreeJs = function() {

    // set the scene size
    var WIDTH = 400, HEIGHT = 300;

    // set some camera attributes
    var VIEW_ANGLE = 45, ASPECT = WIDTH / HEIGHT, NEAR = 0.1;
    var FAR = 10000;

    // create a WebGL renderer, camera and a scene
    var renderer = new THREE.WebGLRenderer();
    var camera = new THREE.PerspectiveCamera(
        VIEW_ANGLE, ASPECT, NEAR, FAR);
    var scene = new THREE.Scene();
```

```
// create the cube's material
var material = new THREE.MeshLambertMaterial({
    color : 0xCC0000
});

// create a new mesh with cube geometry, the material next!
var cube = new THREE.Mesh(new THREE.CubeGeometry(
    50, 50, 50, 16, 16, 16), material);

// create a point light
var light = new THREE.PointLight(0xFFFFFF);

// set its position
light.position.x = 10;
light.position.y = 50;
light.position.z = 130;

// the camera starts at 0,0,0 so pull it back
camera.position.z = 150;

// start the renderer
renderer.setSize(WIDTH, HEIGHT);

cube.rotation.x += 0.5;
cube.rotation.y += 0.4;

// add the cube to the scene
scene.add(cube);

// and the camera
scene.add(camera);

// add to the scene
scene.add(light);

// get the DOM element to attach to assume we've got jQuery to hand
var $container = $("<div id='container' />").appendTo('.v-ui');

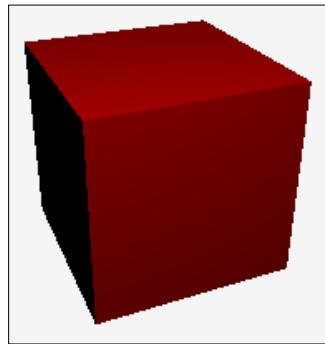
// attach the render-supplied DOM element
$container.append(renderer.domElement);
this.onStateChange = function() {

    // draw!
    renderer.render(scene, camera);
}
}
```

Important parts of this code are highlighted:

- We put the JavaScript server-side in the `com.packtpub.learnvaadin.js` package (same as the component), so the namespace mimics it, replacing dots with underscores
- We use jQuery to create a `div` on the HTML page and it will be used to display the scene
- Finally, the only code we need to call in the `onStateChange` function is the rendering itself

This example's complete code for can be browsed and forked at <https://github.com/nfrankel/js-integration>. In a browser, this is what should be displayed:



It is nothing fancy, a simple cube, and yet we used Vaadin to display raw JavaScript.

 **Because we can**

Just because we can do something that it should be done. In particular, one advantage of Vaadin is to wrap messy HTML, JavaScript, CSS, and AJAX under a nice and clean Java layer. Using JavaScript directly in one part or two of your application is definitely OK, but if you find yourself writing more JavaScript than Java code, you would better use the framework capabilities or remove it completely.

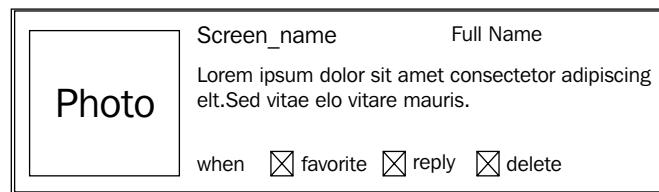
Componentized Twaattin

The table we used to display our tweets in *Chapter 7, Core Advanced Features* displayed the data, but it was lacking in design.

Moreover, Twitter itself has another way of showing the tweets: looking at the site, we can easily see there's potential for a reusable component in the form of a tweet widget!

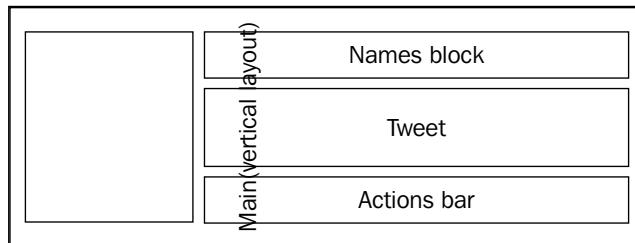
Designing the component

On Twitter's site, a tweet looks similar to the following diagram:



There is not much chance a GWT widget is available. So, we are going to go for the composition approached, we learned about in the first section.

Laying out the component would look similar to the following diagram:

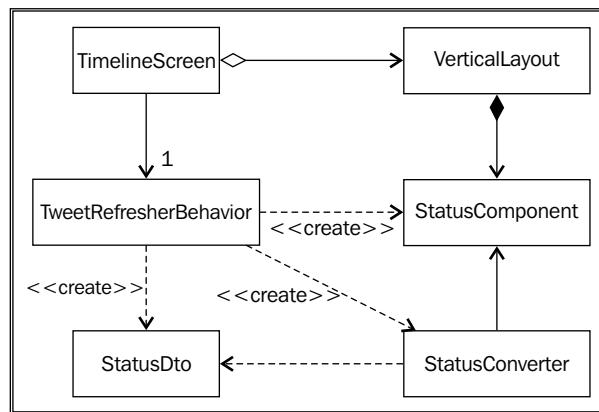


Updating Twaattin's code

The most important Twaattin's change is the new component. However, we need a way to pass Twitter data to the component. In the previous section, it was handled through bean items: it is not possible anymore.

A first-approach implementation could be to pass the raw `Status` interface to the component in the constructor. Unfortunately, this would couple the component to **Twitter4J API**. It would also put logic for computing fields in the component, which is too much responsibility in a good object-oriented design: the component's role should only be to display content, nothing else.

A more refined design would be to create a dumb data placeholder, also known as a **Data Transfer Object**, and pass it instead. A third-party class would hold logic to compute fields from the `Status` to the DTO.



Data Transfer Object

```
public class StatusDto {  
  
    private String name;  
    private String screenName;  
    private String profileImage;  
    private String tweet;  
    private String retweetedBy;  
  
    // Getters and setters go here  
  
}
```

As stated earlier, this class is very simple, with no logic.

Status component

In the component, we only use very basic widgets seen before: layouts, labels and links. Images are new, but are only thin wrappers around external resources.

```
import static com.vaadin.server.Sizeable.Unit.PIXELS;
import static com.vaadin.shared.ui.label.ContentMode.HTML;

import com.packtpub.learnvaadin.twaattin.ui.convert.StatusConverter;
import com.packtpub.learnvaadin.twaattin.ui.convert.StatusDto;
import com.vaadin.server.ExternalResource;
import com.vaadin.server.Resource;
import com.vaadin.ui.Button;
import com.vaadin.ui.CustomComponent;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Image;
import com.vaadin.ui.Label;
import com.vaadin.ui.Link;
import com.vaadin.ui.VerticalLayout;

@SuppressWarnings("serial")
public class StatusComponent extends CustomComponent {

    public StatusComponent(StatusDto dto) {

        ExternalResource userPage =
            new ExternalResource(StatusConverter.TWITTER_USER_URL +
                dto.getScreenName());

        Link name = new Link(dto.getName(), userPage);

        Label screenName = new Label('@' + dto.getScreenName());

        HorizontalLayout names = new HorizontalLayout(name,
            screenName);

        names.setSpacing(true);

        Label tweet = new Label(dto.getTweet(), HTML);

        HorizontalLayout actionsBar =
            new HorizontalLayout(new Button("Reply"),
                new Button("Retweet"), new Button("Favorite"));

        setCompositionRoot(new VerticalLayout(names, tweet, actionsBar));
    }
}
```

```
    actionBar.setSpacing(true);

    String retweetedBy = dto.getRetweetedBy();

    VerticalLayout rightSide;

    if (retweetedBy == null) {

        rightSide = new VerticalLayout(names, tweet, actionBar);

    } else {

        Label label = new Label("Retweeted by " + retweetedBy,
                               HTML);

        rightSide = new VerticalLayout(
            names, tweet, label, actionBar);
    }

    rightSide.setSpacing(true);

    Resource pictureRes = new
        ExternalResource(dto.getProfileImage());

    Image picture = new Image(null, pictureRes);

    picture.setHeight(50, PIXELS);
    picture.setWidth(50, PIXELS);

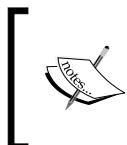
    HorizontalLayout mainLayout =
        new HorizontalLayout(picture, rightSide);

    mainLayout.setMargin(true);
    mainLayout.setSpacing(true);

    setCompositionRoot(mainLayout);
}
```

Apart from the whole layout code, two important things are worthy of notice:

- The first is the component constructor. It has data for all data needed for the component creation in a nice single DTO parameter. We don't provide a setter so that this data is used for initialization only; the component is immutable.
- Also, we remembered to use the `setCompositionRoot()` method and not to add components directly to the component but to the root instead.



Code online displays the `setStyleName()` calls: they are here so we can customize the design with theming but are not mandatory. They should be used in any composite component worth reusing though.



Status converter

The previous component is used only for display. The transformation of status data into HTML and hyperlink is done in the converter.

Timeline screen

The timeline screen has to be changed to remove the table, and add a vertical layout to stack the tweet components.

```
public class TimelineScreen extends VerticalLayout {

    private static final long serialVersionUID = 1L;

    public TimelineScreen() {
        setMargin(true);

        Label label = new Label(VaadinSession.getCurrent()
            .getAttribute(Principal.class).getName());

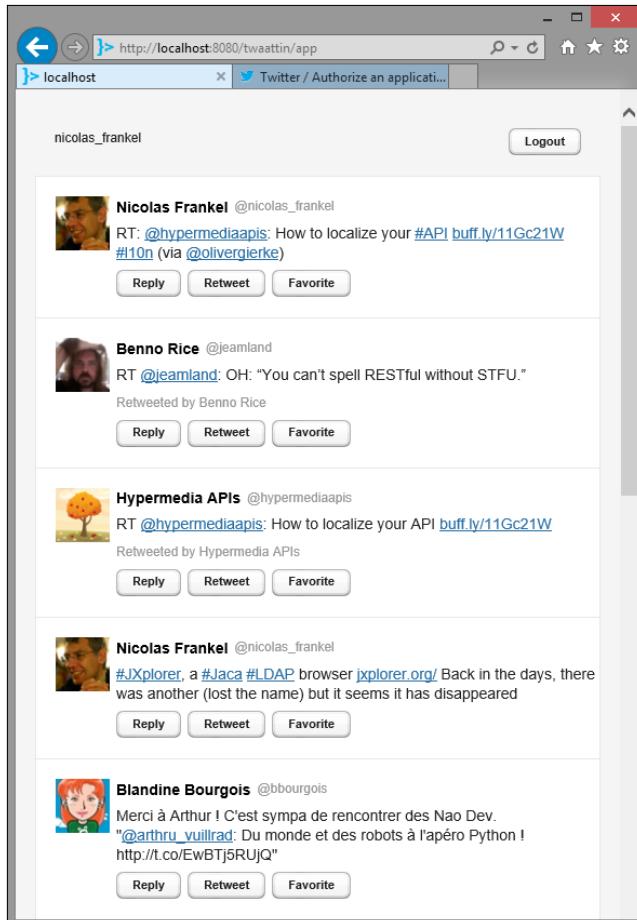
        Button button = new Button("Logout");
        button.addClickListener(new LogoutBehavior());

        HorizontalLayout menuBar = new HorizontalLayout(label,
            button);
        menuBar.setWidth(100, PERCENTAGE);
    }
}
```

Creating and Extending Components and Widgets

```
menuBar.setComponentAlignment(button, MIDDLE_RIGHT);  
menuBar.setMargin(true);  
  
addComponent(menuBar);  
  
addComponentAttachListener(new TweetRefresherBehavior());  
  
VerticalLayout timeline = new VerticalLayout();  
  
addComponent(timeline);  
}  
}  
}
```

The source code is available online at <https://github.com/nfrankel/twaattin/tree/chapter9>.



Summary

In this chapter, we learned about several ways to add custom components to extend our palette of existing components.

The first section told us about component composition. Composition is all about inheriting from `CustomComponent`. This can be done manually or graphically through the Visual Designer available in the Eclipse Vaadin plugin.

We also described how to create add-ons, to add client-sided behavior in existing widgets. Then we moved further to wrap existing client-side GWT widgets in new Vaadin components. Finally, we learned how to do without GWT and directly interact with JavaScript. All of those are based on these core classes:

- Connectors allow for Vaadin communication between client and server code
- Shared state represent common data structure synchronized between both sides
- RPC directly execute server method calls from client code

This chapter should have brought you all the skills necessary to go beyond Vaadin out-of-the-box components. The next chapter will be about using Vaadin in the enterprise.

10

Enterprise Integration

Until now, we used Vaadin in a "standard" fashion, packaged with an IDE-based build and deployed on an application server (or JSP/servlet container). This chapter will show us details on how to integrate Vaadin in some components that are part of the enterprise ecosystem:

- Maven-based build
- Deploying Vaadin portlets on portals
- Deploying Vaadin-based services on OSGi platforms
- Finally, deploying Vaadin applications "in the cloud"

We will take a brief look at what each platform really is, and then see how Vaadin can run on them with a little tweaking.

First we will see how to manage multiplatform development and how build tools minimize the required effort.

Build tools

Until now, in order to create our WAR, we either ran the application inside Eclipse (or other IntelliJ) or used its export feature, which is nice but not really automated enough. During the normal course of enterprise projects, packages are created through standard tools. There are the following few arguments in favor of such tools:

- Nowadays, we need automated and reproducible builds so that we may have continuous builds. Every time a commit is detected or at fixed times, the build is launched and a bad commit – one that breaks the build – is spotted as early as possible.

- Moreover, if we want our project to be able to run on multiple platforms, and as some configuration may be in conflict, we need to remove those manual and error-prone configuration changes for each build.
- If our project has more than one developer, it lets each one develop with their favorite IDE, confident in the fact that a third-party tool will handle the build itself.
- Finally, build tools let us plug in other features, such as automated tests.

Available tools

Nowadays, plenty of enterprise-quality build tools are available for free. The problem lies in choosing the right one.

Apache Ant

In Java, the first portable build tool was Apache Ant (<http://ant.apache.org/>). At the time, it was a revolution that soon engulfed the entire ecosystem: every project worth its salt provided an Ant build file that allowed users build it regardless of their respective operating system.

However, Ant's following limitations soon became apparent:

- The Ant build file is very liberal in its approach. To be simplistic, it just defines targets, whatever those may be: compile, copy, create the archive, and so on. As such, no two build files appear the same even if they perform the same thing. In short, build files are not paragons of readability.
- In addition, Ant's compile target needs the classpath definition. As a build has to be consistent, external libraries have to be provided and thus committed to the project's source versioning software that it then bloats, and adds no particular value to.

Apache Maven

In order to correct Ant's flaws, Apache launched a new build tool named Maven (<http://maven.apache.org/>). It corrected the previous shortcomings with two brand-new ideas at the time:

- Maven brought standardization to builds:
 - A standardized build cycle: compile, copy, archive, and so on. As such, Maven's build file, the famed **Project Object Model (POM)** does not tell what it does, like in Ant, but only configures how it does it.
 - Standardized subprojects in the form of modules. Now, two projects using WAR inclusion in an EAR look alike!
- Maven also brought the concept of a repository where every project should put its artifacts. Then, instead of manually downloading third-party libraries, projects would only need to reference them in the registry.

Fragmentation

Maven's approach was not to every developer's taste. Some argued against XML format's verbosity, others against the POM's lack of extensibility, some just wanted to hold onto something they practiced for years, such as Ant.

The explosion of languages on the JVM added to the confusion, with every language coming with its own build tool: Gradle for Groovy, Rake for Ruby, Gant for Grails, SBT for Scala, and the list goes on.

Final choice

Choosing a build tool in such a context is hard, yet necessary. In the context of this book, we will use Maven to manage our build.

Two major arguments in favor of Maven are as follows:

- Despite many criticisms against Maven, it actually is a major build tool, if not the tool of choice
- Vaadin provides an archetype (see below) to help us create Vaadin projects

Tooling

In Eclipse (or Eclipse-based STS), Maven tooling is provided by the Eclipse's m2e plugin available at <http://download.eclipse.org/technology/m2e/releases/> update sites. Please refer to *Chapter 3, Hello Vaadin!*, for a reminder regarding the use of update sites.

After having installed m2e, navigate to **Window | Preferences | Maven | Catalog**. Click on **Open Catalog**. Search on wtp and install the found **m2e-wtp** connector as a regular plugin.

Troubleshooting

If during the web application testing Vaadin throws the following exception, check the deployment folder on the server:

`javax.servlet.ServletException: Failed to load application class`

Chances are that one or more Vaadin JAR files are missing. In this case, it is perhaps because the m2e-wtp connector is not installed.

As for IntelliJ, it supplies Maven support out of the box. Enjoy!

Maven in Vaadin projects

In Eclipse, there are basically two ways to build Vaadin projects with Maven, and this is to either create a Vaadin project and add Maven features to it or the other way around.

Both are valid depending on our use cases.

Mavenize Vaadin projects

First, create a Vaadin project just as we did in *Chapter 3, Hello Vaadin!*, with the following three important differences:

- For the Java sources folder:
 - Click on **Remove** on `src`
 - Then **Add Folder** `src/main/java`
- For default output folder, enter `target/classes`
- Finally, for the web content directory folder, enter `src/main/webapp`

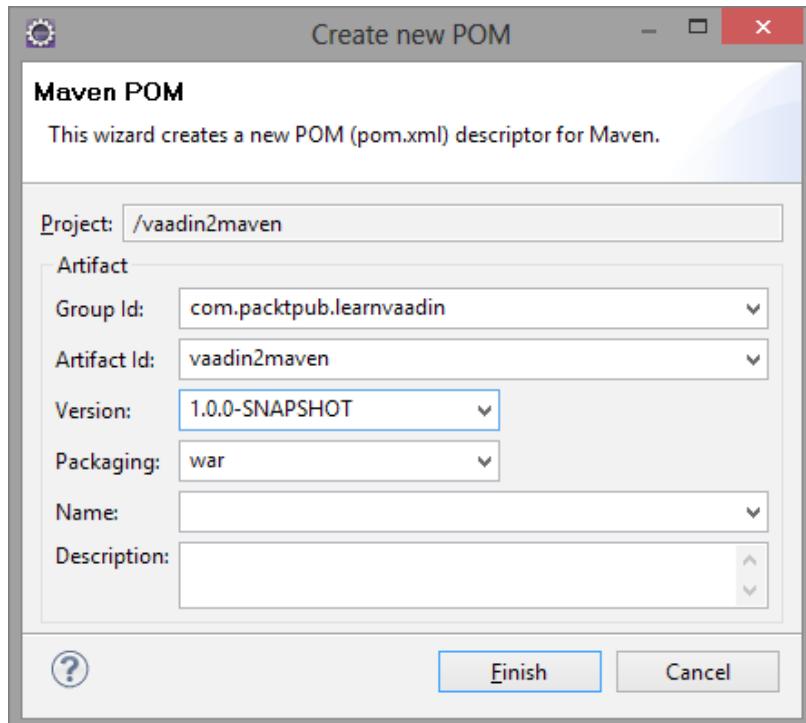


If these values remind you of the standard Maven folder structure, you are absolutely right! Entering these values will prevent us from manually moving folders around after adding Maven to the mix.

Then, we need to use Maven dependencies management instead of Ivy:

1. Right-click on the project directory and navigate to **Configure | Convert to Maven Project**. This opens a window: **Group Id**, **Artifact Id**, **Version**, **Packaging**, **Name**, and **Description** fields are used by the plugin to create the POM. Fill the values as you would a brand new POM. However,

Packaging should be WAR in all cases! Click on **Next**, as shown in the following screenshot:



2. Right-click again and navigate to **Ivy | Remove Ivy dependency management....** Click on **Yes** on the opening popup. Remove `ivy.xml` and `ivysettings.xml` files, as they are now useless.

Now comes the manual part; we have to manually clean up our project. Click on the project and navigate to **Java Resources | Libraries**. Now select Apache Tomcat 6.0 (the server library), EAR libraries, and Web App Libraries, right-click and navigate to **Build Path | Remove from Build Path**.

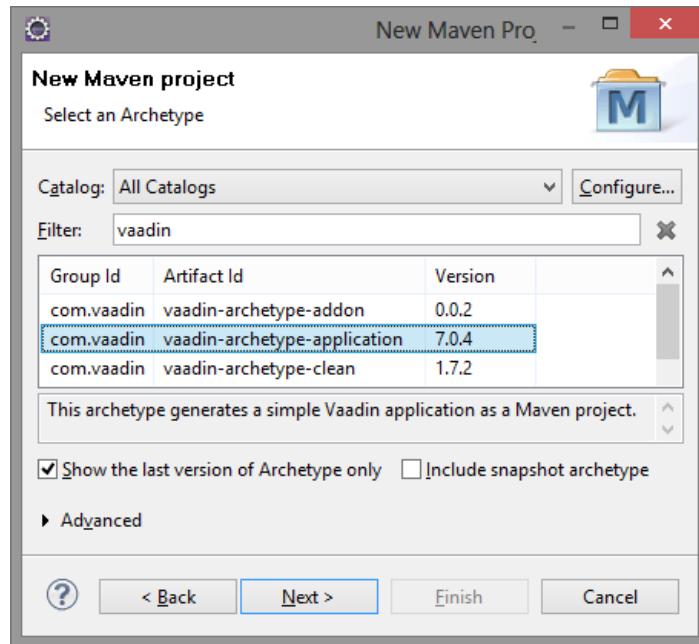
Finally add the mandatory Vaadin dependencies. For a list of these, we should use the Vaadin artifact (see next section).

Vaadin support for Maven projects

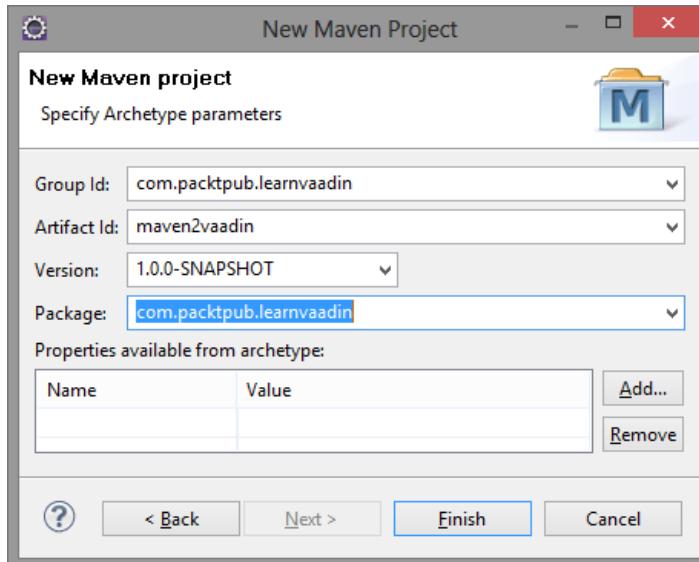
The other possible way is to create a Maven project and add Vaadin support in Eclipse.

1. Navigate to **File | New | Project**. Locate a Maven project and click on **Next** two times (default values are ok for the first window).

2. For archetype selection, filter with Vaadin and select **com.vaadin: vaadin-archetype-application** and click on **Next**, as shown in the following screenshot:



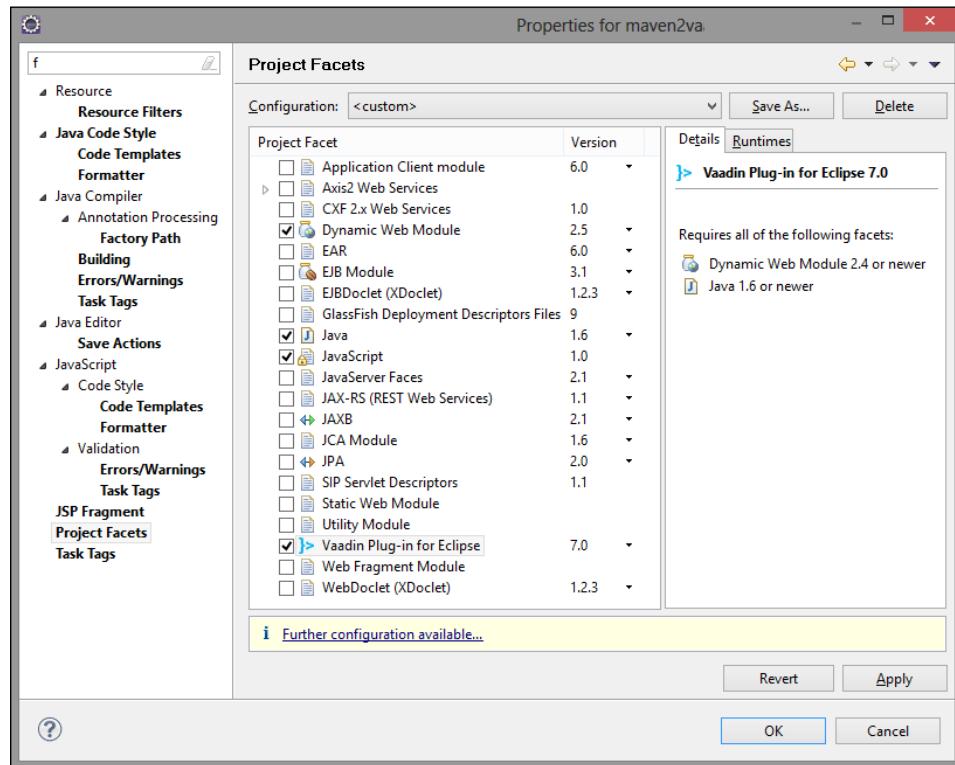
3. Enter the values (take a look at the *Mavenize Vaadin projects* section if you lack imagination) and click on **Finish**:



A quick glance at the POM tells us it is much more furnished as compared to adding dependency management to a Vaadin project:

- The project .build .sourceEncoding property avoids making our build platform-dependent
- The other properties manage versions: Vaadin JARs and the plugin
- The Maven compiler plugin is configured to use Java 6
- The Jetty plugin will let us easily test our Vaadin application with the Jetty servlet container
- Finally, two additional repositories are added besides the main Maven repository: one for Vaadin snapshots, the other for Vaadin add-ons

In order to use the Vaadin Eclipse plugin, right-click on the project and click on **Properties**. Select **Project Facets** on the left-hand side and check **Vaadin Plug-in for Eclipse**, as shown in the following screenshot:



Alternatively and independently of any IDE, we could also use the Maven command-line to create the project. It can be achieved with the following command:

```
mvn -B archetype:generate -DarchetypeGroupId=com.  
vaadin -DarchetypeArtifactId=vaadin-archetype-application  
-DarchetypeVersion=7.0.4 -DgroupId=com.packtpub.learnvaadin  
-DartifactId=maven2vaadin -Dversion=1.0.0-SNAPSHOT -Dpackaging=war
```

Once the project is created, it can be imported in Eclipse and converted to a Vaadin project as done previously.

Widget compilation

It may be necessary to add client compilation during the build process.

In that case, Vaadin provides the following snippet in the generated POM to handle that:

```
<plugin>  
    <groupId>com.vaadin</groupId>  
    <artifactId>vaadin-maven-plugin</artifactId>  
    <version>${vaadin.plugin.version}</version>  
    <configuration>  
        <extraJvmArgs>-Xmx512M -Xss1024k</extraJvmArgs>  
        <webappDirectory>  
            ${basedir}/src/main/webapp/VAADIN/widgetsets  
        </webappDirectory>  
        <hostedWebapp>  
            ${basedir}/src/main/webapp/VAADIN/widgetsets  
        </hostedWebapp>  
        <noServer>true</noServer>  
        <!-- Remove draftCompile when project is ready -->  
        <draftCompile>false</draftCompile>  
        <compileReport>true</compileReport>  
        <style>OBF</style>  
        <strict>true</strict>  
        <runTarget>http://localhost:8080/</runTarget>  
    </configuration>  
    <executions>  
        <execution>  
            <goals>  
                <goal>resources</goal>  
                <goal>update-widgetset</goal>  
                <goal>compile</goal>  
            </goals>  
        </execution>  
    </executions>  
</plugin>
```

Seasoned Maven users will notice that the widgetset generation output, either through Eclipse or through Maven, is localized in `src/main/webapp` and not in `target`. This means that we have no need to compile it with each build and it can be safely isolated in an activatable profile. However, this means we also have to take care not to commit the `VAADIN/widgetset` folder.

As this plugin takes a good part of the build time, it is worthwhile.

Running Twaattin outside IDE

With our new Maven build, we can effortlessly run Vaadin outside any IDE by just typing the following command:

```
mvn jetty:run
```

Twaattin is available at `http://localhost:8080/app`.



Further Jetty configuration is also available; the whole documentation is available at <http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin>.



Mavenizing Twaattin

There is no advantage to migrate Twaattin under Maven at this time, but we will prepare the project now to be ready for the rest of the chapter.

Preparing the migration

As we are to update an existing project instead of creating the right project structure from the beginning, we will need to move the directories around:

1. First, create the `src/main/java` and `src/main/resources` directories. Move Java files from `src` to the former and properties files to the latter. Then, right-click on the project and select **Properties**. Click on **Build Path** from the **Source** tab. Remove `src` and add both.
2. While we are at it, set the output directory to `target/classes` instead of `build/classes` on the same tab.
3. Create a folder named `webapp` at `src/main/` and move files from `WebContent` to it. Eclipse does not take on this sort of operation kindly; we will need to smooth it somewhat. To do so, locate the `org.eclipse.wst.common.component` file under the `.settings` directory. Change the value of the `source-path` attribute of the following tag by the new location:
`<wb-resource deploy-path="/" source-path="/src/main/webapp"/>`
4. Finally, remove the `lib` directory, as it is an Ivy-generated folder

Enabling dependency management

Now we just need to follow the preceding section procedure. Dependencies go well beyond Vaadin. We also need the Servlet API and Twitter4J.

Finishing touches

The last step just lets us use Maven dependency management. A few steps are still needed in order to have a nice polished build file.

Cleaning up warning messages

There may be some disturbing warning messages that pollute our nice build, such as the following:

```
Using platform encoding (Cp1252actually) to copy filtered resources, i.e.  
build is platform dependent
```

The preceding message appears when developing on Windows. We need to add a property to our POM to specify the file encoding:

```
<project ...>  
...  
  <properties>  
    <project.build.sourceEncoding>  
      UTF-8  
    </project.build.sourceEncoding>  
  </properties>  
</project>
```

Optimizations

Remove the following unnecessary plugins from the POM:

- `vaadin-maven-plugin`: There is no need to compile widgetset as there are no custom widgets
- `maven-clean-plugin`: As there is no widget generation, there is no need to clean it
- `lifecycle-mapping`: This plugin is only made to integrate Maven build into Eclipse

Final POM

The final POM being quite large, it is available online at <https://github.com/nfrankel/twaattin/blob/chapter10-maven/pom.xml>.

This may seem quite a struggle for little added value, but it is the prerequisite to effortlessly add additional platforms to run Twaattin on.

Portals

Although not as widespread as some wished them to be 10 years ago, portals are still common enough to be a target of choice for Vaadin web applications.

Portal, container, and portlet

Before going further, one has to understand some essential notions about what a portal is and how it is constituted.

Three different concepts are each attached to a different granularity level:

- **Portlet:** A portlet is a pluggable software component meant to be displayed inside a portal. As opposed to a servlet, a portlet only generates a part of the rendered page. If we make a parallel to a wall, a portlet would be a brick.
- **Portal:** A portal is a full-fledged application that aggregates portlets. Most portals will also allow administrators (or individual users) customize the portlets layout, as well as the global portal's look-and-feel.

To continue our wall analogy, the portal would be the wall itself.

- **Portlet container:** A portlet container is the technical layer that manages portlets. It plays the same role for them, as would a servlet container for servlets, including request forwarding, response handling, lifecycle management, and so on.

In a wall, the container would be the cement that sits between the wall and each separate brick.

Choosing a platform

Currently, there are the following two different JSR specifications for portlet API:

- **JSR-168**: This is also known as the Java Portlet API, is the first generation specification introducing portlets in Java. As many first specifications go, it has limits which the next JSR tries to address.
- **JSR-286** : This is the Java Portlet API 2.0 and is meant to replace the former specification. It tries to align itself with the OASIS portlet specifications (WSRPS 2.0) and introduces new features such as the following:
 - Inter-portlet events
 - Shared rendering parameters
 - Non-HTML resource serving
 - Portlet filters

In addition, different products implementing these specifications are available. At the time of writing this book, enterprise-grade portals that can be considered for use comprise both commercial products and free/open source ones.

- Commercial products include the following:
 - IBM WebSphere Portal is part of the many offerings of IBM WebSphere
 - Oracle WebLogic Portal
- Free/open source products consist of the following:
 - Apache Jetspeed 2 is a portal relying on Apache Pluto, a raw portlet container. It does not seem very widespread.
 - JBoss GateIn (<http://www.jboss.org/gatein>) is the result of the merging of former projects JBoss Portal and eXo Portal.
 - Last but not least, Liferay is a portal commonly found in the enterprise. It is developed by Liferay Inc. which also provides commercial fee-based support for it.

Liferay

Vaadin Ltd. and Liferay Inc. already work together in a partnership to integrate their products with one another, so there is plenty of good documentation available online, mostly on www.vaadin.com and on www.liferay.com that describe how to do that.

Starting with Version 6, Liferay comes bundled with Vaadin library and widgetsets, so there is nothing to install on the platform to run Vaadin applications on it.

If one encounters difficulties running Vaadin applications as portlets, one should turn to Vaadin forums, which provide answers to many questions.

GateIn

The platform of choice taken as an example for the rest of this section is GateIn. What is explained in the following sections can be adapted to your portal of choice.

 For enterprise users, GateIn is also available as an enterprise edition, JBoss Portal Platform. It is a particular version of GateIn for which JBoss provides support—at a price. For more information, visit <https://www.redhat.com/products/jbossenterprisemiddleware/portal>.

Downloading and installation

The latest version of GateIn (3.5.0 at the time of this writing) comes bundled with either JBoss AS 7 or with Tomcat 7. Using one or the other depends on one's requirements and personal tastes. In the context of this book, we will use Tomcat 7 as we do not need specific Java EE features.

Download the version that is suitable for you from <http://www.jboss.org/gatein/download>.

Installing GateIn is just a matter of unzipping the downloaded archive. For Windows users, take care to extract it under a path that contains no space characters.

 By default, GateIn uses **HyperSonic SQL (also known as HSQLDB)**, a file-based database. Production-grade installations require a more robust database backend. What is enough for the scope of this book is not enough for real-world cases. Please refer to the online documentation to configure the underlying database at https://docs.jboss.com/gatein/portal/latest/reference-guide/en-US/html_single/#sect-Reference_Guide-Database_Configuration.

Launch

Navigate to <GATEIN_HOME>/bin and type:

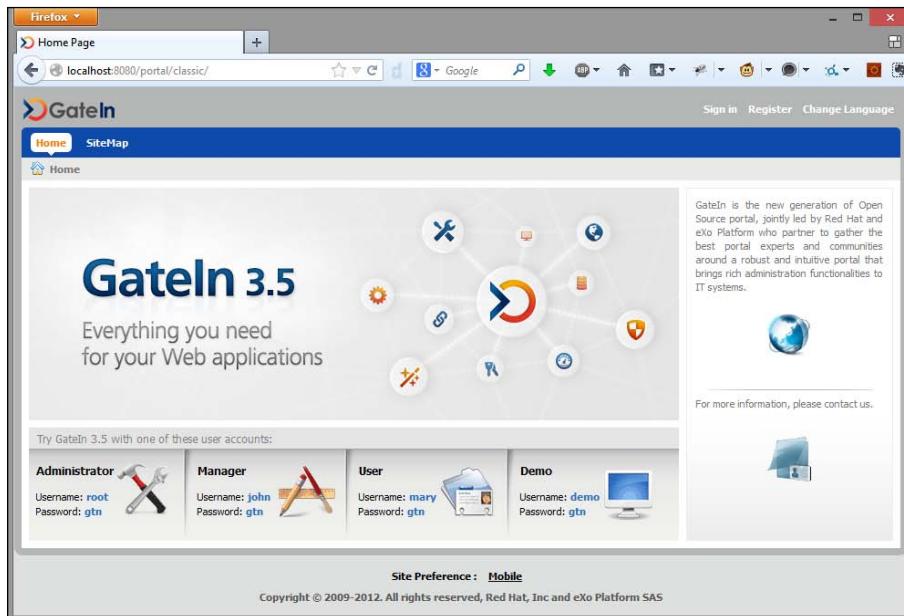
```
gatein start
```



Alternatively, calling `gatein jpda start shell` (or `bat`) instead lets us launch GateIn in debug mode, in order to connect to the JVM within the IDE.



In order to check whether GateIn launched normally, navigate to `http://localhost:8080/portal`. The default portal home page should be displayed, as shown in the following screenshot:



Now it is done, we are good to go further!



Troubleshooting

If Tomcat runs fine but you get a "404 not found" error in your browser, be sure to check that the `CATALINA_HOME` environment variable is not set (or at least set to GateIn extract directory) and carefully read the console.



Tooling

The good news here is that we already have all the needed tooling at our disposal, as the Vaadin Eclipse plugin has the right parameters to create portlets instead of standard web applications.

A simple portlet

As an example, we will develop a simple portlet that displays a message when a button is clicked.

The development of such an application holds no secret for us, so let's focus our attention on the differences when developing portlets.

Creating a project

Portlet project creation starts like any other Vaadin project; by navigating to **File | New | Other** and choosing **Vaadin7 Project**.

Fill the wizard as we did in *Chapter 3, Hello Vaadin!*. Now in deployment configuration, replace **Servlet (default)** with **Generic Portlet (Portlet 2.0)**.

In order to configure the context root, enter `hello` during the WebModule configuration step; for the portlet title, type `vaadinportlet`.

 In the final step, the Portlet version is asked for again; do not change it as it could have adverse effects on the project's integrity

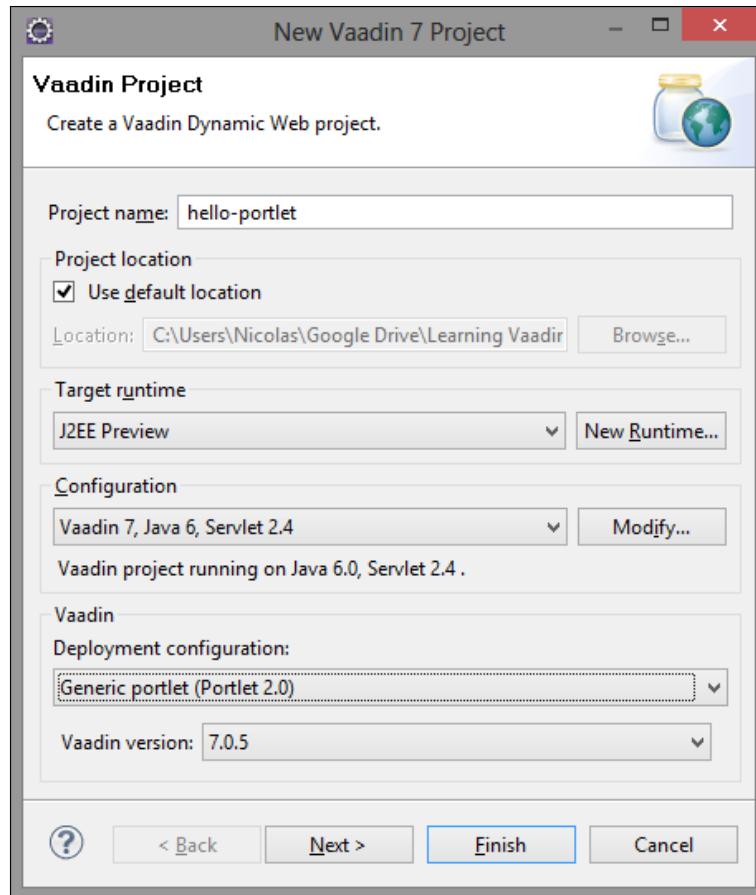
Finishing the wizard, Vaadin creates the project, just like we saw in *Chapter 3*.

Portlet project differences

There are some subtle (and not so subtle) Portlet project differences that we will look into in detail.

Portlet deployment descriptor

The Vaadin Eclipse plugin created a `portlet.xml` file under the `WEB-INF` folder in the new project. Developers familiar with portals know this file as the portlet deployment descriptor. For those unfamiliar, it is very akin to a web deployment descriptor (`web.xml`), but aimed at portals instead of application servers.



The generated descriptor is as follows (comments excluded):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<portlet-app version="2.0"
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">
<portlet>
    <portlet-name>Hello Portlet</portlet-name>
    <display-name>hello-portlet</display-name>
    <portlet-class>com.vaadin.server.VaadinPortlet</portlet-class>
    <init-param>
        <name>UI</name>
        <value>com.packtpub.learnvaadin.HelloPortletUI</value>
    </init-param>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
        <title>hello-portlet</title>
        <short-title>hello-portlet</short-title>
    </portlet-info>
</portlet>
</portlet-app>
```

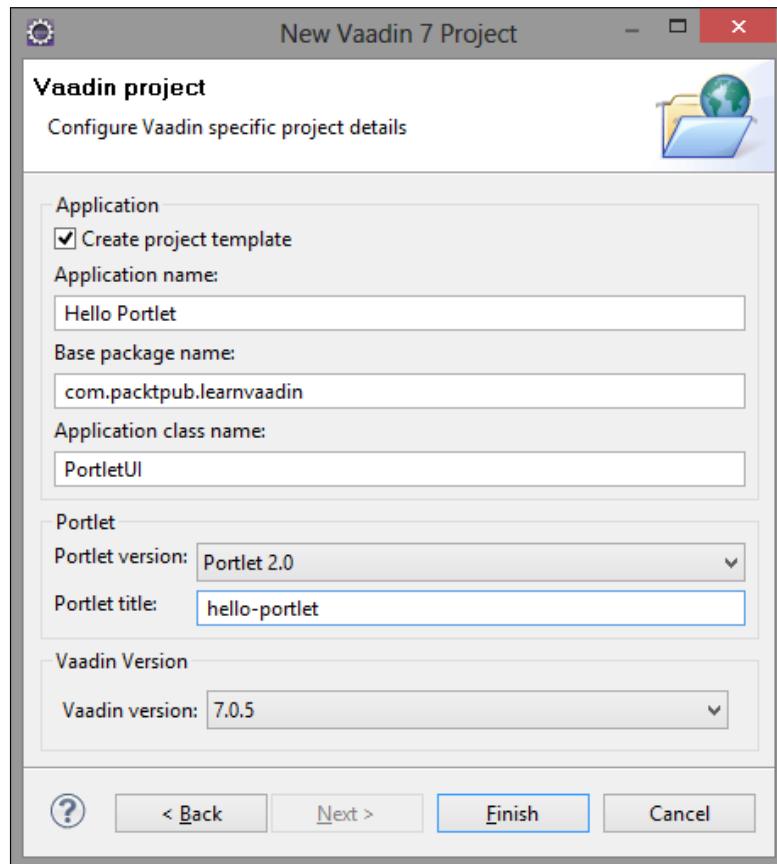
The following describes the differences as compared to a generated web deployment descriptor:

- The XML schema points to a portlet schema, not a web app schema.
- The support section displays both the output mime-type and the portlet-mode. As a general rule, the former should not be changed as it fits Vaadin's output. As for the mode, the specification defines the following three modes:
 - View displays the portlet; it is the standard mode
 - Edit lets the user change/configure/manage the portlet
 - Help displays help for the portlet

In addition to the three standard modes, it is also possible to define custom modes.

Although each of these modes has a semantically unique meaning, nothing prevents us from using the view mode to configure the portlet. By default, the mode is "view" and it matches most of our use cases.

- Finally, title and short-title have to be filled. Both Liferay and GateIn will look for these pieces of data and won't display the portlet if they are missing:



Portal proprietary files

The Vaadin Eclipse plugin also creates the Liferay proprietary files: `liferay-display.xml`, `liferay-portlet.xml`, and `liferay-plugin-package.properties`.

When using GateIn as the portal platform, those can safely be ignored or deleted.

Similarities

Despite the previous differences, keep in mind that there are also a few similarities; the application instance is the same regardless of the deployed platform. This means that we can keep our code, just update the deployment descriptor, and we are good to go on an entirely different platform!



There is slight reservation to this, however, as listeners (if in scope) are context-dependent; see the section named *Handling portlet specifics* in this chapter.



Using the portlet in GateIn

Now all the necessary development tasks are done, we still have to make the portlet available to GateIn and add it in a page.

Deploying in GateIn

Deploying the portlet in GateIn will make it available for further use. In order to do this, perform the following steps:

1. Export the portlet as a WAR file. Right-click on the project and choose **Export | WAR file**.
2. Copy-paste the exported WAR file under <GATEIN_HOME>/webapps.

If the initial configuration is kept, Tomcat will automatically and recursively unpack the exported WAR file in the webapps directory and will start the application. This can be verified by the following Vaadin trace in Tomcat's command window:

```
May 05, 2013 7:09:30 PM org.apache.catalina.startup.HostConfig deployWAR
INFO: Deploying archive hello-portlet.war of webapp
May 05, 2013 7:09:32 PM com.vaadin.server.
DefaultDeploymentConfigurationcheckProductionMode
Warning:
=====
Vaadin is running in DEBUG MODE.
Add productionMode=true to web.xml to disable debug features.
To show debug window, add ?debug to your application URL.
=====
```

Adding the portlet to a page

In order to add the portlet to a page, we need to log in to the portal with the credentials to do it.

Sign in

Click on **Sign in** on the left-hand side of the menu bar. It opens a login pop-up window. By default, root/gtn will enable us to have enough credentials to make changes to pages (or add new ones, for that matter).

Refresh portlets

Before adding the portlet to a page, a user action is needed. Once logged in, choose the **Group** menu and navigate to **Administration | Application Registry**. It opens the complete list of available portlets. At this point, we cannot see our newly deployed servlet.

Click on the **Import Applications** option at the top-left corner of the window (after navigating to **Portlet | Gadget**). A confirm dialog opens asking whether portlets should be imported (this will be done in their respective category) and click on **OK**.

A new category appears, matching the deployed WAR file's name. Under it, there should be a single portlet that takes its name from the portlet deployment descriptor.

Add portlet

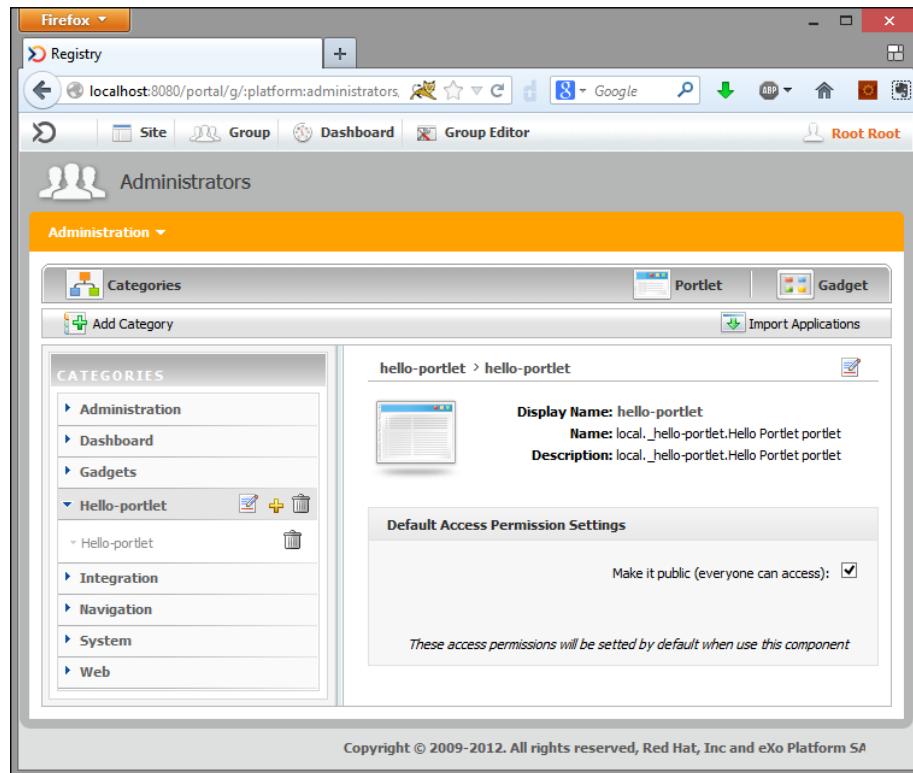
From this point on, portlets exported to the GateIn webapps directory will be available for use by end users.

Therefore, navigate to the **Site Editor** menu and choose **Edit Page**. This will change the page mode to edit and will change the display. On the **Page Editor** tool pane, search for the freshly added category, and drag-and-drop the child portlet where you want on the main layout.

Saving is achieved by clicking on the disk icon in on the **Page Editor** tool pane. The new portlet will be displayed but unfortunately, nothing shows apart from the following plain message:

```
Failed to load the bootstrap javascript: /html/VAADIN/vaadinBootstrap.js
```

The next section will get us this answer to this error.



Configuring GateIn for Vaadin

In fact, we missed a crucial step in using Vaadin in GateIn and that is the portal configuration. We will correct this in the following sections.

Themes and widgetsets

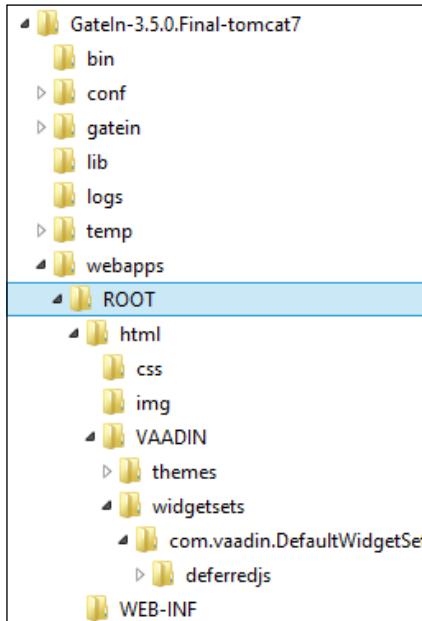
Vaadin client's code needs access to the /VAADIN path, where both themes and GWT-compiled widget sets lie. When served by a servlet, this path is relative to the root of the web application.

Unfortunately, in the context of a portal, Vaadin has no reference to the servlet context, thus it cannot get the webapp's root. Therefore, the framework will try to access /html/VAADIN relative to the server's root to get these files.

There are some options to make this work.

Serve files from the portal

The first solution is to serve files directly from the portal. We need to extract files from the `vaadin-client-compiled.jar` and `vaadin-themes.jar` files in the `ROOT` webapp under `html`, as well as the `vaadinBootstrap.js` file:



The latter file is available in the `VAADIN` folder in `vaadin-server.jar`.

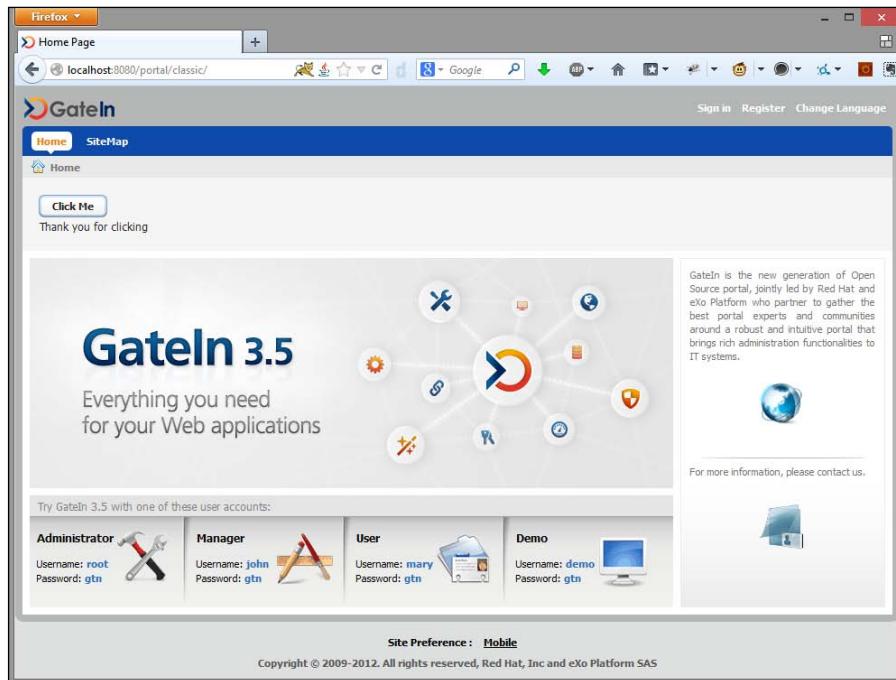
This approach has the advantage of putting the default widgetset in a common location, so that all Vaadin's portlets use it. On the other side, this makes the build process more complicated as we have to separate static Vaadin files in one archive and bytecode inside the WAR file.

Serve files from an HTTP server

As an alternative, if we have an HTTP server (Apache HTTP server, Microsoft IIS, or another) in front of our application server(s), the former could serve these files instead of the latter.

It has the same pros and cons as the previous solution, but with an additional tier.

Pick a solution and use it here; refreshing the page will display the Vaadin portlet, as shown in the following screenshot:



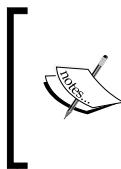
Advanced integration

Beyond a simple Hello World, we need advanced capabilities brought by Vaadin. We will check how they work in a portal context.

Restart and debug

We can use Vaadin restart and debug features in GateIn (or any other portal) like we used in standard web applications.

Just append `restartApplication` and/or `debug` query parameters and watch the magic happen.



Be wary that in this case, it will restart all Vaadin portlets displayed on the page at refresh time. Moreover, it will only show the debug window of a single Vaadin portlet, in a non-deterministic way. Hence, it is easier to use these parameters during development when there is only a single Vaadin portlet; that is when they are used anyway.

Handling portlet specifics

Portlets have some features that are unique as regards to standard web applications.

First, they have a unique lifecycle. Beyond the portlet standard `init()` and `destroy()` methods, two steps are also important: the process action phase and the render phase.

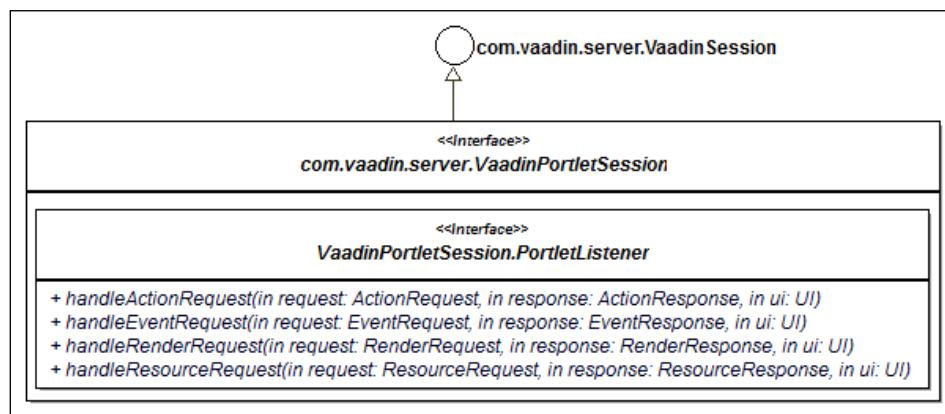
Second, a portlet container also manages the following for each portlet:

- Its mode: Edit, view, and help
- Its window state: Normal, minimized, and maximized

Finally, JSR 286 also adds event handling between portlets.

All of these features translate into the portlet API. However, much like the servlet API, the Vaadin framework hides the latter, so developers do not have to worry about it.

In order to let us interact with these elements, Vaadin makes the `PortletListener` interface available, defined in the `VaadinPortletSession` class.



Each of the listener's method maps one of the following portlet request-processing phases:

Phase	Method	Description
Render	<code>handleRenderRequest()</code>	Generate HTML
Action	<code>handleActionRequest()</code>	Process user actions
Resource	<code>handleResourceRequest()</code>	Serve resources
Event	<code>handleEventRequest()</code>	Manage events

Adding a listener to a GUI component is just a matter of checking the session to check its type (`VaadinServletSession` or `VaadinPortletSession`). The following snippet illustrates this:

```
import static javax.portlet.PortletMode.EDIT;
import static javax.portlet.PortletMode.VIEW;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.EventRequest;
import javax.portlet.EventResponse;
import javax.portlet.PortletMode;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.ResourceRequest;
import javax.portlet.ResourceResponse;

import com.vaadin.server.VaadinPortletSession;
import com.vaadin.server.VaadinPortletSession.PortletListener;
import com.vaadin.server.VaadinSession;
import com.vaadin.ui.Button;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.CustomButton;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;
import com.vaadin.ui.VerticalLayout;

@SuppressWarnings({ "serial", "deprecation" })
public class HelloScreen extends CustomComponent implements
PortletListener {

    private Button button = new Button("Click Me");

    public HelloScreen() {

        final VerticalLayout layout = new VerticalLayout();
        layout.setMargin(true);
        setCompositionRoot(layout);

        button.addListener(new Button.ClickListener() {
```

```
    public void buttonClick(ClickEvent event) {  
  
        layout.addComponent(  
            new Label("Thank you for clicking"));  
    }  
});  
  
layout.addComponent(button);  
  
if (VaadinSession.getCurrent() instanceof VaadinPortletSession) {  
  
    VaadinPortletSession portletSession =  
        (VaadinPortletSession) VaadinSession.getCurrent();  
  
    portletSession.addPortletListener(this);  
}  
}  
  
@Override  
public void handleRenderRequest(RenderRequest request,  
    RenderResponse response, UI ui) {  
  
    PortletMode mode = request.getPortletMode();  
  
    if (mode == VIEW) {  
  
        button.setVisible(false);  
    } else if (mode == EDIT) {  
  
        button.setVisible(true);  
    }  
}  
  
// Other methods left empty  
...  
}
```

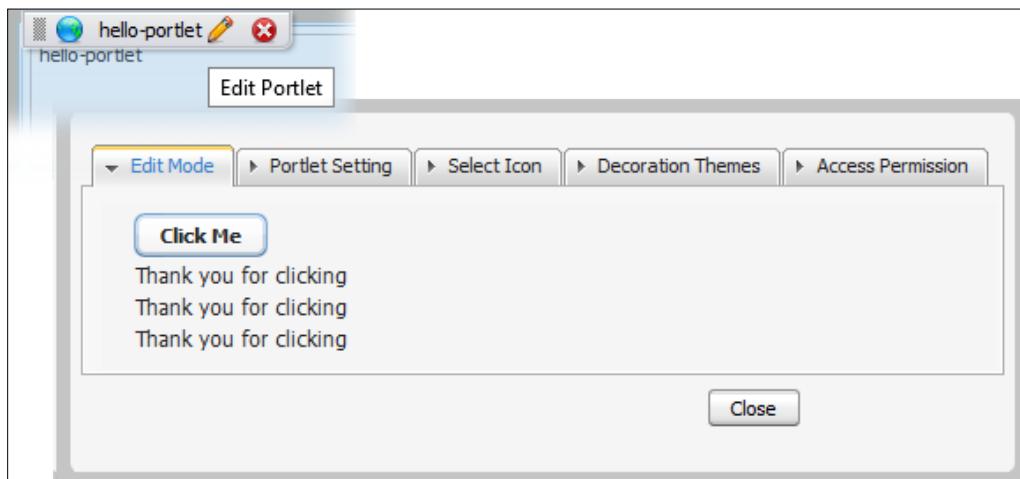
This code is pretty self-explanatory: in the edit mode, the button is visible while in the view mode, it is not.

We also need to add support for edit in the portlet deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<portlet-app ...>

<portlet>
    ...
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
    </supports>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>edit</portlet-mode>
    </supports>
    ...
</portlet>
</portlet-app>
```

In order to check the behavior, click on the **Edit Portlet** icon in the portlet bar during page edit. This opens a pop up, showing the portlet in edit mode, *button included*. Click on it a couple of times then click on **Close** at the bottom of the pop up. After quitting page edition, the button will not be shown anymore, but labels will be, and as many times as the button was clicked during editing:



Check complete source online at <https://github.com/nfrankel/hello-portlet>.

Portlet development strategies

During development, chances are we won't get our portlet right the first time. In order to ease our work, there are some techniques we can use.

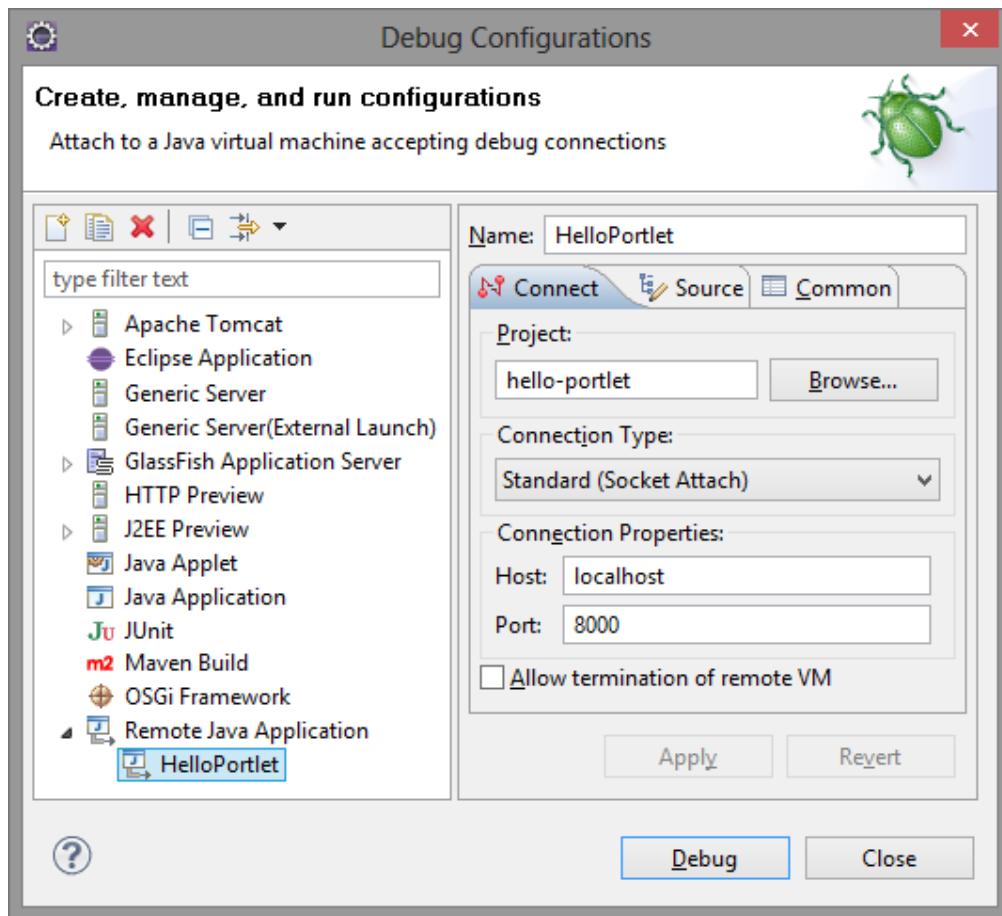
Keep our portlet servlet-compatible

As a rule of thumb, portlet development is generally much slower than servlet development because of all the packaging and deployment involved. Tomcat or NetBeans integration in Eclipse lets us update classes and see changes on the fly most of the time. Hence, it is better if we can develop a servlet; it is advised to keep the servlet-compatibility mode as long as possible to test our portlet in a simple servlet container.

Portal debug mode

When this parallel cannot be maintained, for example, in order to add inter-portlet communication features, we will have to deploy our newly-developed portlet on the target portal to develop further. In order to be able to debug the portlet, the following actions are in order:

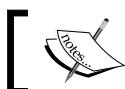
1. Launch GateIn in debug mode with the help of the `gatein jpda start` command (instead of the standard `gateinstart`).
2. In Eclipse, connect to the JVM launched in the debug mode. In order to accomplish this, click on the scrolling menu of the **Debug** button on the toolbar (the one that looks like a bug) and choose **Debug Configurations**.
3. This will open a window. Select **Remote Java Application** in the list and click on the **New Launch Configuration** in the upper-left corner. The default port (**8000**) is suitable if GateIn also uses the default configuration:



From this point on, we can set breakpoints in our portlet and manage the flow from inside the IDE!

Updating a deployed portlet

Finally, successive portlet deployments are likely to be in order. In order to do that for the first deployment, all we have to do is export the WAR to GateIn and then import applications in the portal (see the *Deploying in GateIn* section in this chapter for a quick reminder).



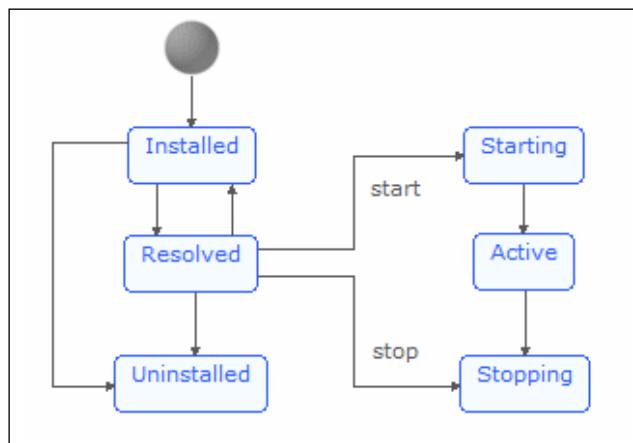
The current version of Vaadin does not support Push mode in portlets. Therefore, we cannot migrate Twaattin to a portlet.

OSGi

OSGi is a very promising technology that aims to resolve modularization granularity and classpath issues inherent to Java. Despite its inherent structure, OSGi concepts are straightforward and aim to work at the following three different levels:

- First, OSGi comes with the concept of a bundle. **Bundles** are JAR files, but provide additional information in their manifest; most notably, a bundle expresses which packages it exposes to the outside world (read which ones have public visibility) and which one it needs in order to fulfill its dependencies. Optionally, it may also define an activator class, which is a listener used on the next level. This defines the **modularity layer**.
- OSGi also provides the way to manage bundle's lifecycle. Basically, a bundle may be in six different states, and transitions between states are well defined. The six states are as follows:

State	Description
INSTALLED	Platform is aware of the bundle.
RESOLVED	Checked for OSGi-correctness.
STARTING	Being started. If an activator is defined for the bundle, calls its <code>start()</code> method.
ACTIVE	Running and available in the OSGi platform to be used by other bundles.
STOPPING	Being stopped. If an activator is defined for the bundle, calls its <code>stop()</code> method.
UNINSTALLED	Final state.



This layer is known as the **lifecycle** layer. A major benefit of this layer is the hot deployment of bundles.

- Finally, some commons capabilities are described in the **service** layer, each in the form of a Java interface. These interfaces are grouped in the following three different sets:
 - System, such as logging, deployment, and admin
 - Protocol, such as HTTP and Universal Plug-and-Play
 - Miscellaneous, such as XML parsing

Bundles may implement those services and register in the services registry for other bundles to discover and use them.

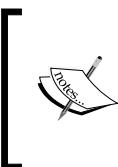
Also, note that OSGi is a standard promoted by the OSGi Alliance, which includes notable members such as IBM, Oracle Red Hat, and VMware. For a deeper insight into OSGi, visit the OSGi Alliance site at <http://www.osgi.org/>.

Choosing a platform

Some OSGi platforms are available to add our own bundles to the following:

- Knopflerfish (<http://www.knopflerfish.org/>), an independent OSGi implementation.
- Apache Felix (<http://felix.apache.org/>), a small yet compliant OSGi platform.
- Eclipse IDE's manages its plugin architecture with the help of OSGi. The Equinox project (<http://www.eclipse.org/equinox/>), a whole OSGi implementation was started to achieve that.

Many applications servers also run on top of OSGi in order to manage complexity through modularization: Oracle Glassfish 3, Oracle WebLogic, IBM WebSphere, Red Hat Flyweight (formerly JBoss Application Server), and OW Jonas 5 are such application servers.



Another way to use OSGi is to embed a compatible container in the application. As one should probably tweak the application server the application will run into, chances are that it won't be considered as a viable option in the enterprise. Thus, we will focus on the "running in an OSGi container" way.

In order to illustrate OSGi, we will use Oracle Glassfish in the rest of this document. Glassfish uses Felix under the cover. In most cases, however, it should play no part as OSGi is a specification and Felix is only an implementation among others.

Glassfish

Like many other providers, Oracle supplies two distributions of its Glassfish application server: Glassfish Server Open Source Edition (available under CDDL or GPLv2 license) and Glassfish Server which is available under a commercial license.

In order to stay true to the open source approach, we will use the Open Source distribution.

Deploying bundles

There are basically three ways to deploy an OSGi bundle to Glassfish. However, for the first two, we will need to update Glassfish configuration in order to enable them. These are worth the effort.

Prerequisites

In order to enable OSGi access, go to the Glassfish root and type the following command:

```
asadmin create-jvm-options -Dglassfish.osgi.start.level.final=3
```

Start (or restart) Glassfish.



This works with Glassfish v3.1.2 or later. Former versions should follow the procedure shown at <http://stackoverflow.com/questions/8349209/launching-felix-shell-on-glassfish>.



Telnet deployment

Once the configuration is updated, we can type the following in the command prompt:

```
telnet localhost 6666
```

The command prompt will display the following text:

```
Welcome to Apache Felix Gogo
g! help
```



For security-minded readers, Glassfish should only allow local telnet connections by default. We should not worry too much about letting remote users access the console.



Installing an OSGi bundle is just a matter of typing the following command:

```
install file:///path/to/bundle
```

Glassfish will neatly return the newly installed bundle's ID:

```
Bundle ID: xxx
```

Just remember to then start the bundle (pass the `start` command the returned ID).

```
start xxx
```

To be sure everything went ok, type `lb` on the command prompt. It displays the whole list of all installed bundles, as well as their current status; it should show the new bundle (probably as the last item) as ACTIVE.

File system deployment

The second way to deploy a bundle to the Glassfish server is the simplest; just put the bundle in the `bundles` folder under `<GLASSFISH_HOME>/glassfish/domains/<MY_DOMAIN>/autodeploy/`.

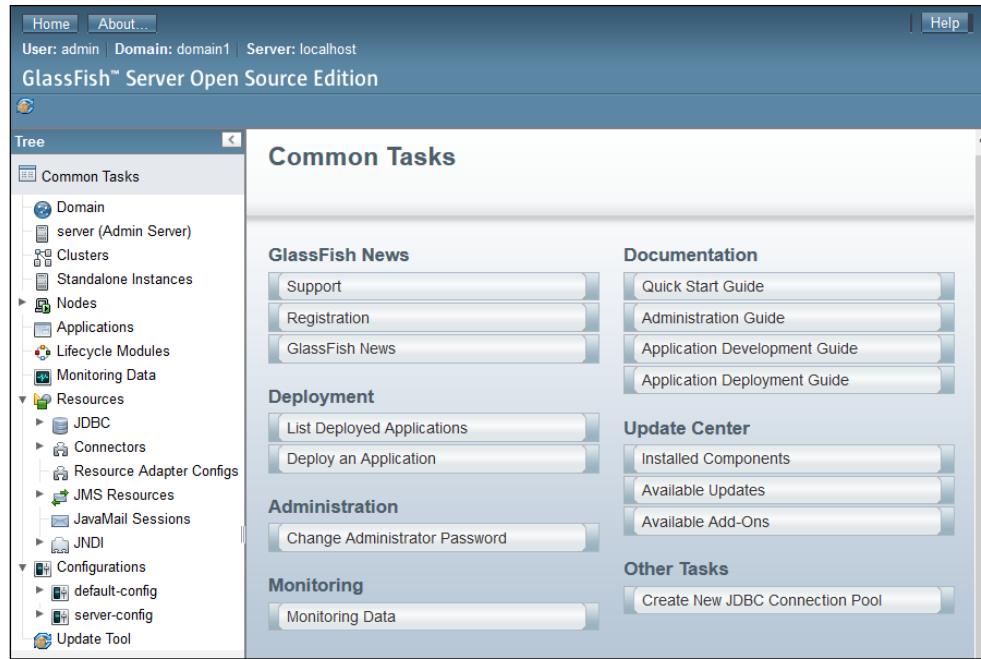
A look at the log can confirm it works:

```
INFO: Installed C:\Development\glassfishv3\glassfish\domains\domain1\autodeploy\bundles\jsoup-1.6.3.jar
INFO: Started bundle: file:/C:/Development/glassfishv3/glassfish/domains/domain1/autodeploy/bundles/jsoup-1.6.3.
```

In addition, we can check the Felix console just like with the previous method.

Web console deployment

Navigate to the Glassfish administration panel at <http://localhost:4848>.

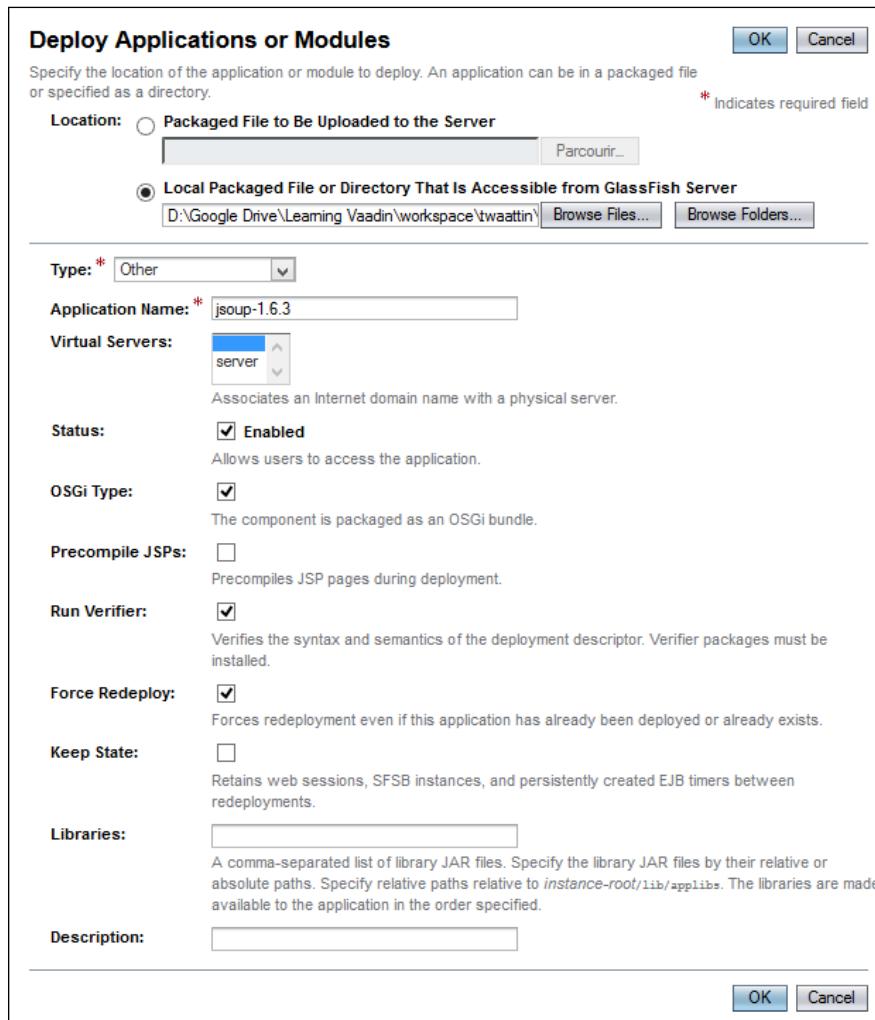


Click either on **Deploy an Application** on the homepage or on the **Applications** menu on the left-hand side.

Fill the opening window as follows:

- Select the bundle to deploy
- **Type: Other**
- **Status:** check **Enabled**
- **OSGi Type:** check
- **Run Verifier:** check
- **Force Redeploy:** check if the bundle is already deployed

Click on **OK**; this will deploy the bundle. The preceding options are depicted in the following screenshot:



Logs should display the operation's status as follows:

```
INFO: Installed org.jsoup [257] from reference:file:/C:/Development/glassfishv3/glassfish/domains/domain1/applications/jsoup-1.6.3/
INFO: Started org.jsoup [257]
INFO: jsoup-1.6.3 was successfully deployed in 5 867 milliseconds.
```

Whatever the method used, the result would be the same, separating OSGi dependencies from the WAR.

Tooling

The good news here is that there is no need for further tooling beside what we already installed; we will keep our IDE and Glassfish will serve as an OSGi container.

The bad news is that the many OSGi advantages come at a price; OSGi makes development more structured. Although we previously could use a build tool or not, now we have to use one in order to reproduce builds through automation. In order to be consistent with former sections of this chapter, it is advised to use Maven; how to do it is the goal of the following sections. Alternatively, one could choose Ant, SBT, or even Make if one is really desperate.

Vaadin OSGi use cases

Benefits from OSGi are varied and depend on what layers are used.

Vaadin bundling

Strategies regarding libraries in an application server context come in the following three different flavors, each having pros and cons:

- Often, every web application comes with its libraries bundled in its WEB-INF/lib folder. The good part is that each is then independent; the bad is that even with similar libraries in the same versions, it adds to the WAR size, but also unnecessarily increases the memory load as each WAR's class loader has to load an instance of the library.
- Another approach is to put libraries in the application server's shared libraries folder. In this case, WAR can be very lightweight. Moreover, only the application server class loader has to load the library; it is done once. On the downside, applications have no choice on the version of these libraries, and have to use the one provided by the application server (much like the servlet API library).
- Finally, most application servers allow administrators to add libraries to the classpath of single applications; some even allow the defining of groups of libraries to ease that. This strategy gets the best of both worlds— independency and memory optimization—at the cost of a much higher administration cost.

With OSGi, the administrator could deploy different versions of Vaadin on the OSGi platform and each deployed applications would specify which version it needs. The platform would then resolve dependencies for us.

Modularization

A non-Vaadin-specific use case for Vaadin we could benefit from is modularization. We could decouple an application in modules, and then manage each one's lifecycle independently from one another.

A good example of module granularity in the case of multiple Vaadin application per WAR would be application objects. A slightly more convoluted, yet still a possible case would be Vaadin windows; one could conceive an application so that screens could be upgraded separately.

Hello OSGi

As an example, we will deploy a simple application as an OSGi bundle.

First, we will need to create a Vaadin project that we will make OSGi compatible. Proceed as usual or refer to the description in *Chapter 3, Hello Vaadin!*.

Making a bundle

As can be expected, making the JAR OSGi-compliant is just a matter of putting the right information in the manifest. At the very least, manifest headers should include the following:



For a complete list of headers, visit <http://www.osgi.org/Specifications/ReferenceHeaders>.



Header	Value	Description
Bundle-ManifestVersion	2	OSGi version compatibility. 2 means OSGiR4.2
Bundle-Name	hello osgi	Human-readable name
Bundle-SymbolicName	com.packtpub.learnvaadin.osgi	System name
Bundle-Version	1.0.0	Bundle version
Web-ContextPath	/osgi	Context root
Import-Package	javax.servlet, javax.servlet.http, com.vaadin.server, com.vaadin.ui	Packages dependency

Header	Value	Description
Bundle-ClassPath	WEB-INF/classes, WEB-INF/lib/vaadin-client-compiled-7.1.0.jar, WEB-INF/lib/vaadin-themes-7.1.0.jar	This is the classpath. As it's not a standard webapp but a bundle, we have to redefine the classpath the OSGi way.

Inaccessible OSGi libraries



Note that bundle classpath not only uses `WEB-INF/classes`, but also the necessary client JARs (those that contain widgetsets and themes). Those can be deployed as OSGi bundles but do not seem to be accessible from other web applications. We need to keep them inside the library folder and reference them explicitly.

The final manifest looks like the following code snippet:

```
Bundle-ClassPath: WEB-INF/classes,  
WEB-INF/lib/vaadin-client-compiled-7.1.0.jar,  
WEB-INF/lib/vaadin-themes-7.1.0.jar  
Bundle-ManifestVersion: 2  
Bundle-Name: hello-osgi  
Bundle-SymbolicName: com.packtpub.learnvaadin.osgi  
Bundle-Version: 1.0.0  
Bundle-Vendor: Nicolas Frankel  
Import-Package: com.vaadin.server;version=[7.1.0,7.1.0],  
com.vaadin.ui;version=[7.1.0,7.1.0],  
javax.servlet,  
javax.servlet.http  
Manifest-Version: 1.0  
Web-ContextPath: /osgi
```

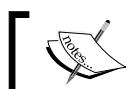
 The formatting should be taken care of precisely, as there is a limit of 72 characters for a line, according to the JAR specifications (visit <http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html#Name-Value%20pairs%20and%20Sections> for more information).

Export, deploy, and run

The previous manifest is enough. Now is the time to deploy the application.

First, we need to deploy dependent bundles, as was explained in the *Deploying bundles* section:

- jsoup-1.6.3
- vaadin-server-7.1.0
- vaadin-shared-7.1.0
- vaadin-shared-deps-1.0.2



They can (and should) be safely taken from the exported web application.



We then need to export the webapp; right-click on the project and choose export as WAR. As Eclipse has no hint that we want an OSGi bundle, we need to use our favorite ZIP tool, open the exported archive and remove all JARs under the WEB-INF/lib directory those referenced in the manifest (vaadin-client-compiled and vaadin-themes).

Finally, deploy the web application itself and it is done. At this point, navigate to <http://localhost:8080/osgi> and watch the magic happen.

Correcting errors

Actually, the magic consists of a bug that makes the application unusable as it is. It is referenced in Vaadin's bug tracking system at <http://dev.vaadin.com/ticket/9942>.

While this bug is fixed, we can certainly find workarounds.

Vaadin servlet

OSGi is very strict on-class loading and classloader isolation. This means that just enhancing a standard JAR with the previous manifest to make it a bundle is not enough.

The vaadin-server bundle will throw a `ClassNotFoundException` when trying to instantiate the UI in our own bundle.

The reason is that the former cannot know about our classloader used to load the latter. In order to fix this, we need to provide a custom `VaadinServletService` able to return its own classloader:

```
public class FixOsgiClassLoaderVaadinServlet extends VaadinServlet {
    @Override
```

```
protected VaadinServletService createServletService(
    DeploymentConfiguration deploymentConfiguration)
throws ServiceException {

    VaadinServletService servletService =
        new VaadinServletService(this, deploymentConfiguration) {

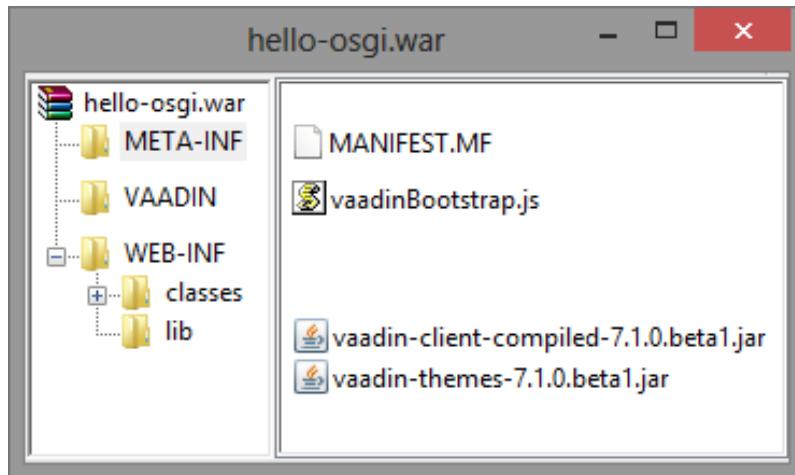
            @Override
            public ClassLoader getClassLoader() {
                return getClass().getClassLoader();
            }
        };
    servletService.init();
    return servletService;
}
```

This new servlet class has to be referenced in the web deployment descriptor in place of the standard `VaadinServlet`.

Vaadin bootstrap

The `vaadinBootstrap.js` script will not be found either, because it is part of the `vaadin-server.jar`. Here, the fix is either to embed the JAR in `WEB-INF/lib` and reference it in the manifest or copy it in the web application (as for portlets).

The former option would defeat the whole point of OSGi and bundles, so we will opt for the latter. The final webapp structure would look something like the following:



Integrating Twaattin

Now it is time to go beyond a simple example and update Twaattin to be an OSGi bundle in its own right.



At the time of this writing, Vaadin Push is not OSGi-compatible, so be aware the following will not work. You're advised to follow newer versions of Vaadin Push to check if they correct the problem. I'll try to update Twaattin to be OSGi-compatible when this happens.

Bundle plugin

In the previous Hello OSGi example, we manually crafted the OSGi manifest. As Twaattin already uses Maven, much information is already available in the POM; it is a good idea to also let Maven handle the OSGi manifest.

To this end, the Apache Felix project provides the `maven-bundle-plugin`, which is based on the Bnd utility. The complete documentation of the plugin is available online at <http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>.



Bnd is a free tool (provided by aQute) that can generate a project's OSGi manifest from its classes and libraries. More information can be found at <http://www.aqute.biz/Bnd/Bnd>.

The plugin takes Bnd one step further and uses all Maven-provided data, including dependencies that suits our purpose just fine. Goals are provided not only to generate the OSGi bundle, but also to just generate the manifest.

Like with the rest of the Maven ecosystem, the plugin infers reasonable defaults for most pieces of information, meaning we only need to configure specific parts.

It translates like the following for Twaattin:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.4.0</version>
  <extensions>true</extensions>
  <executions>
    <execution>
      <id>bundle-manifest</id>
      <phase>process-classes</phase>
      <goals>
```

```
<goal>manifest</goal>
</goals>
</execution>
</executions>
<configuration>
    <supportedProjectTypes>
        <supportedProjectType>war</supportedProjectType>
    </supportedProjectTypes>
    <manifestLocation>
        ${project.build.directory}/${project.artifactId}-${project.version}/
        META-INF
    </manifestLocation>
    <instructions>
        <Bundle-ClassPath>
            .,WEB-INF/classes,
            WEB-INF/lib/twitter4j-core-${twitter4j.version}.jar,
            WEB-INF/lib/twitter4j-stream-${twitter4j.version}.jar
        </Bundle-ClassPath>
        <Import-Package>
            !twitter4j,!twitter4j.auth,*
        </Import-Package>
    </instructions>
    </configuration>
</plugin>
```

This configuration does many things; we will have a look at each part.

The first thing to know is that by default, the plugin only operates on `jar` and `bundle` packaging types. It does not work for other artifact types (it fails silently). Nevertheless, it can be configured to allow other packaging; the first thing to do is to instruct the plugin we understand it is a WAR file, but we still want the OSGi manifest. This is done with the `supportedProjectType` tag in the preceding XML snippet.

Then the goal is only to create the manifest so that we may include it in our bundle. In order to achieve this, we isolate it in a specific folder, defined as a Maven property so we don't have to hard-configure it in the assembly.

Finally, the `instructions` tag references individual headers we will find in the OSGi manifest. The following two things are of particular interest:

- Some libraries are OGSi bundles, some are not. The latter, such as Twitter4J, are to be kept inside `WEB-INF/lib`, as was the case with client JAR in Hello-OSGi formerly.

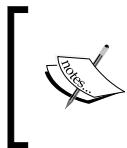
- OSGi-specific classloading system disregards the standard WEB-INF/classes and WEB-INF/lib folders, meaning we have to configure them in the manifest. Libraries kept inside the webapp also have to be configured in the POM.

The previous fragment produces the following OSGi manifest (for readability purposes, the presented manifest does not respect the 72 characters per line limit):

```
Manifest-Version: 1.0
Bnd-LastModified: 1372611930283
Build-Jdk: 1.7.0_02
Built-By: Nicolas
Bundle-ClassPath: .,WEB-INF/classes,WEB-INF/lib/twitter4j-core-3.0.3.jar
,WEB-INF/lib/twitter4j-stream-3.0.3.jar
Bundle-ManifestVersion: 2
Bundle-Name: twaattin
Bundle-SymbolicName: com.packtpub.learnvaadin.twaattin
Bundle-Version: 1.0.0.SNAPSHOT
Created-By: Apache Maven Bundle Plugin
Export-Package: com.packtpub.learnvaadin.authentication;version="1.0.0.
SNAPSHOT",com.packtpub.learnvaadin.osgi;uses:="com.vaadin.server";versio
n="1.0.0.SNAPSHOT",com.packtpub.learnvaadin.service;version="1.0.0.SNAP
SHOT",com.packtpub.learnvaadin.twaattin.presenter;uses:="com.vaadin.ui";
version="1.0.0.SNAPSHOT",com.packtpub.learnvaadin.twaattin.ui.convert;ve
rsion="1.0.0.SNAPSHOT",com.packtpub.learnvaadin.twaattin.ui;uses:="com.
packtpub.learnvaadin.twaattin.presenter,com.packtpub.learnvaadin.twaa
ttin.ui.convert,com.vaadin.annotations,com.vaadin.server,com.vaadin.
ui";version="1.0.0.SNAPSHOT"
Import-Package: com.vaadin.annotations;version="[7.1,8)"
,com.vaadin.server;version="[7.1,8)",com.vaadin.shared.
ui.label;version="[7.1,8)",com.vaadin.ui;version="[7.1,8)"
Tool: Bnd-2.1.0.20130426-122213
```

Note that metadata such as `Bundle-Name`, `Bundle-SymbolicName`, and the like are automatically computed from POM's `artifactId`, `groupId`, and `version` values. Although it is possible to override these values, defaults are good enough for Twaattin.

Moreover, the `Export-Package` header is taken care of by the plugin (and Bnd underneath) and read from our source code.



If crafted by hand, take care that OSGi expects the version to be in MAJOR.MINOR.MICRO.QUALIFIER format. These components have to be separated by dots, so the Maven -SNAPSHOT qualifier should be replaced.



Multiplatform build

In order for our build to truly be multiplatform, we now just have to provide an assembly descriptor for the bundle (as well as configure it in the POM).

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
    <id>osgi</id>
    <formats>
        <format>war</format>
    </formats>
    <includeBaseDirectory>false</includeBaseDirectory>
    <fileSets>
        <fileSet>
            <directory>src/main/webapp</directory>
            <outputDirectory>/</outputDirectory>
        </fileSet>
        <fileSet>
            <directory>${project.build.outputDirectory}</directory>
            <outputDirectory>/WEB-INF/classes</outputDirectory>
        </fileSet>
    </fileSets>
    <dependencySets>
        <dependencySet>
            <outputDirectory>/WEB-INF/lib</outputDirectory>
            <excludes>
                <exclude>*:war</exclude>
                <exclude>com.vaadin.external.atmosphere:*</exclude>
                <exclude>com.vaadin.external.slf4j:*</exclude>
                <exclude>com.vaadin:vaadin-server</exclude>
                <exclude>com.vaadin:vaadin-push</exclude>
                <exclude>com.vaadin:vaadin-shared</exclude>
                <exclude>com.vaadin:vaadin-shared-deps</exclude>
                <exclude>com.vaadin:vaadin-theme-compiler</exclude>
```

```

<exclude>org.jsoup:</exclude>
<exclude>*:commons-cli</exclude>
<exclude>*:commons-jexl</exclude>
<exclude>*:commons-logging</exclude>
<exclude>net.sourceforge.cssparser:</exclude>
</excludes>
</dependencySet>
</dependencySets>
</assembly>
```

We see that OSGi packaging is not more complex than for our previous portlet; the real piece of work is done in the bundle plugin.

We just have to take care to remove the Vaadin JAR from the WEB-INF folder, for it is provided by the OSGi container.



As mentioned before, we sadly need to stop there as Vaadin Push is not OSGi-friendly yet.



Cloud

What is dangerous with the world "cloud" nowadays is that it is a relatively young approach, so not everyone necessarily has the same definition of it.

In order for certain terms to have the same meaning in the scope of this book, we first need to define them.

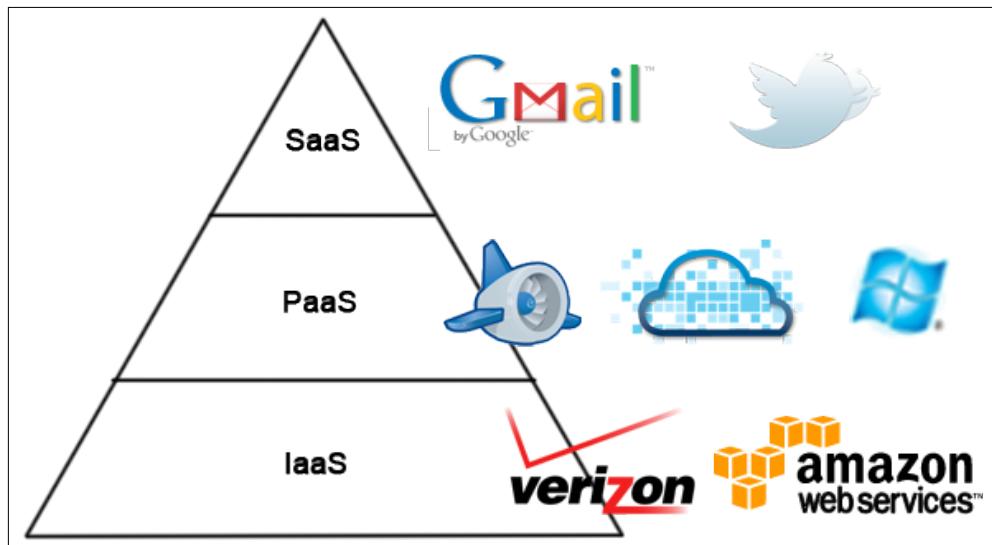
Cloud offering levels

Currently, there is some agreement that there are three different service levels offered by clouds. This should be the shared understanding for our following work on clouds:

- **Infrastructure as a Service (IaaS):** In IaaS, only the hardware is dematerialized. In effect, this only affects system administrators in that they interact with a server whose location they don't know, as opposed to one they know of. The net gain here is a decrease on hardware costs as it is mutualized. We still have to install JVM, application servers, and the rest.
- **Platform as a Service (PaaS):** PaaS goes one step further and also provides the platform, for example, the JVM and the application servers, on top of the distant hardware. In this case, the cloud vendor completely isolates users from the underlying infrastructure behind a façade, which also offers a user interface to configure the different platforms.

- **Software as a Service (SaaS):** The last level of cloud offering is the SaaS, which is a distantly hosted application.

What is common in all three levels is the virtualization of hardware. One never knows on which physical server one runs the OS, the platform, or the software.



For our purposes, the needed offering is situated on the PaaS level; we will just deploy our software "in the cloud".

State of the market

This field being relatively new, things are changing fast. In the Java ecosystem, however, there are stable players providing a cloud platform for our software to run on:

- Google was the first major company to provide a "free" cloud platform for Java web applications in the form of Google App Engine. This approach has some disadvantages, including a reduced scope of the Java API that prevent some tasks such as thread launching, file creation, and so on. Moreover, persistence can only be achieved through JPA or JDO, and only so with the help of the DataNucleus product, thus forcing us to use it locally as well.

Finally, although free at the entry level, there are some fees when one goes above some quotas (but these are quite high; for an up-to-date reference on these limits, refer to <http://code.google.com/intl/fr/appengine/docs/quotas.html>).

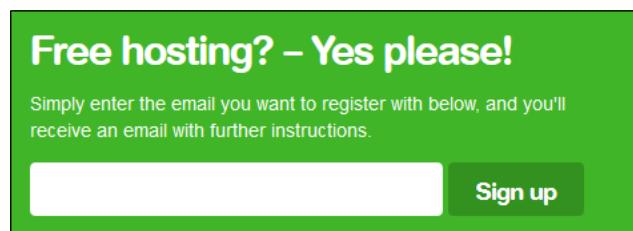
- The second Java offering comes from CloudBees, nearly a pure-player in this area. CloudBees also provides build services along Paas, enabling continuous deployment.
- VMware is also a player on this field by supplying the Cloud Foundry software. This software is provided as an open source project (for private clouds), as well as a platform to deploy on.
- Finally, Jelastic is a relative newcomer but has partnered with Vaadin to provide free (but time limited) hosting to Vaadin applications. In the scope of this book, it is our choice.

Hello cloud

In this section, we will deploy our first web application to Jelastic. Please first create a Vaadin project as we have done before or reuse an existing project.

Registration

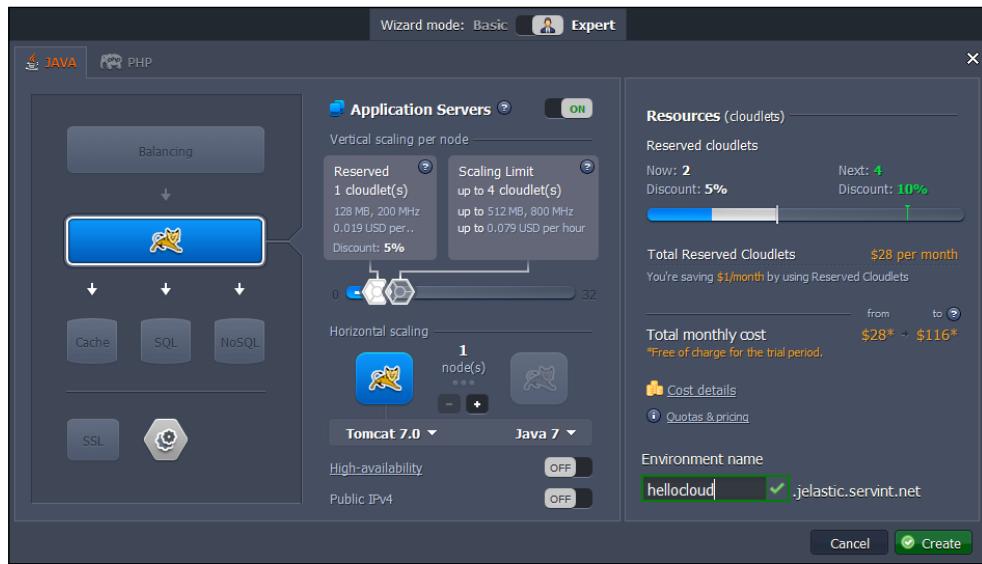
If you intend to use the Jelastic platform, you need to register to get an account first. Go to <https://vaadin.com/cloud>, follow the instructions, and come back when it is done.



Cloud setup

In a standard environment, we would need to create a new virtual server. In the cloud, we just need to configure it, but the steps are the same.

Once logged into the provider, this is exactly what the first screen aims for. Configuration is available for every component of a standard infrastructure:



Resources

The load balancer, as well as most resources described in the following can be *scaled vertically*, meaning each may be allocated more resources. To simplify management, those resources are measured in terms of **cloudlet**, each cloudlet being equivalent to 128MB RAM and 200MHz CPU.

Cloudlet resources are defined using a range:

- The lowest level sets the minimum number of used cloudlets. Even when deployed applications are not in use, this requested number of cloudlets will be dedicated to your infrastructure.
- The higher number sets the maximum number of possible cloudlets. When the load increases, in order to keep acceptable performance, Jelastic will allocate up to this number of cloudlets.

Pricing is per cloudlet per hour.

Balancing

The topmost layer is the **load balancer**. It is the component tasked of dispatching requests between application containers if there is more than one of them. The only choice here is nginx 1.2.

Application server and JDK

Just below balancing configuration lies the application server itself and its associated JDK.

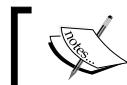
Application servers may be *scaled horizontally*, meaning we can add more application server nodes to serve requests (thus making load balancing mandatory). In this case, once a user request is served by a specific node, the load balancer always dispatches consecutive user requests to the same node, in order to keep user session on the same application server. This strategy is called **sticky session**. The drawback is that if the used node fails, user requests are successfully redirected to the other running node, but all session-relevant data (such as authentication) is lost. In order to remedy to this, the high-availability configuration parameter has to be turned on.

Some of the available application servers are already known to us, Glassfish 3.1 and Tomcat 6.0 and 7.0; others are brand new, Jetty 6.1 and TomEE 1.5. As for JDK, we can choose between 6.0 and 7.0.

Database and cache

Some applications (if not all) need to store data persistently between runs. The traditional way to do this is to use a SQL database. Available options are MySQL 5.5, Maria DB (a spin-off of MySQL) 5.5 and 10.0, and PostgreSQL 8.4 and 9.2.

Alternative to SQL containers is the new NoSQL movement. Jelastic also provides NoSQL backends in the form of CouchDB 1.2 and MongoDB 2.2.



NoSQL stands for "No SQL" or "Not Only SQL" depending on who you ask.



Finally, accessing SQL backends has a huge impact on scalability. Most of the time, it can be improved by keeping data in memory, where access time is much lower than on hard drives. The cache solution offered by Jelastic is Memcached 1.4.

Miscellaneous

Other parameters include the following.

- SSL (Secure Socket Layer), which protects data sent to and from the server from undesired readings and writings.
- Build, as we can use Jelastic not only to run our applications but also to build them.
- Last but not least, we have to choose our domain name, ending with `jelastic.servint.com`.



Beware that once the domain name is chosen, it cannot be changed. When using a trial account, this is a source of concern as only a single environment is allowed; choose wisely!



The **Create** button at the bottom-right corner of the screen creates the environment according to our specifications. Of course, it is always possible to update it later.



A full-blown infrastructure creation can take some time. Enjoy a well-deserved pause during it.



The environment should look like the following screenshot:

The screenshot shows the ServInt interface. At the top, there's a navigation bar with links for 'Upgrade vaadin account', 'Vote for Features', 'Help', and a user account. Below the navigation is a 'Create environment' section with a table showing two environments: 'hellocloud' (Running, Not deployed) and 'Tomcat 7.0.37' (Running, Not deployed). Both environments have 1 instance and 1/4 capacity. Below this is a 'Deployment manager' section with a table for uploaded WAR files. One file, 'HelloWorld...', is listed with a size of 82 KB. At the bottom is a 'Tasks' section with a table showing two recent tasks: 'Adding Tomcat 7.0 node to hellocloud' and 'Creating environment hellocloud', both marked as 'Success'.

A **HelloWorld** application is already available; feel free to delete it (or deploy it). Hover over its line and select the **Delete** button.

Application deployment

Deploying our application "in the cloud" is a 2-step process:

1. Once built as a Web Application Archive, we can easily upload our application.

In the **Deployment Manager** section, just click on the **Upload** link; select the WAR and click on the **Upload** button.

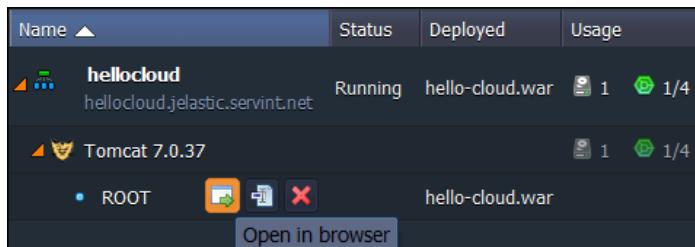
2. Once at least one application is uploaded, it can be deployed on an application server. Hover over the application and click on **Deploy to**; at

At this point, we have to choose the context the application will be deployed. If nothing is typed, it will be at the root of the previously chosen domain.



Deploying an application to a context that is already used by an application will undeploy the oldest application and replace it with the newer.

When the application is finished deploying, it appears under the application server with its associated context. Hovering over it and choosing **Open in browser** will display it in all its glory "in the cloud":



Before going further, deploy some already available applications; it is fun to see them online!

Summary

In this chapter, we have deployed Vaadin applications on exotic platforms that go well beyond simple application servers: portals, OSGi containers, and the cloud. In each case, we used a product, respectively GateIn, GlassFish, and Cloud Foundry to demonstrate the feasibility of it. Should the need arise; however, we now have all the necessary keys to deploy on other products, for example, Liferay, Felix, or Google App Engine.

The lesson here is that Vaadin applications can be run on a variety of platforms without much adaptation.

Index

Symbols

`@SuppressWarnings annotation` 79

A

Absolute layout, UI components 114

AbstractBeanContainer 173

AbstractClientConnector 80

AbstractComponent

about 80

immediate mode 81

properties 80

Abstract Factory 299

AbstractField class 91

AbstractProperty 151

accept criterion 202

access() method 266

ActiveX 16

addClickListener(ClickListener) method

136

addListener() method 134

add-ons categories

data components 272

miscellaneous 272

official 272

themes 272

tools 272

UI components 272

add-ons presentation

detailed view 273

summarized view 273

add-ons search 272

add-ons stability levels

beta 272

certified 272

experimental 272

stable 272

addWindow(Window) method 84

advanced integration, portal

about 367

portlet specifics, handling 368-371

restart and debug features 367

advanced layouts, UI components

about 114

Absolute layout 114

CSS layout 115

CustomLayout 115

AJAX 16, 230

anonymous inner classes

about 138

cons 138

pros 138

Apache Ant

about 346

limitations 346

URL 346

Apache Felix

URL 375

Apache Jetspeed 2 356

Apache Maven 346

Apache Pivot project

URL 17

Apache Portable Runtime 65

Apache Tomcat 79

application deployment 394

application tiers

about 8

data tier 9

logic tier 8

presentation tier 8

appliesToProperty(Object) method 169
architectural considerations, Vaadin event model
about 137
anonymous inner classes, as listeners 138
components as listeners 138, 139
influencing factors 140, 141
presenters as listeners 139
services as listeners 140
architecture, SQL container 245
Asynchronous JavaScript with XML. See AJAX
AutoGenerated annotation 316

B

basic configuration, Windows 85
basic embedding, Vaadin 231
binding
limitation 162
Bnd utility 385
bootstrap script 233
Buffered interface 96
buffering 148
build tools
about 345
Apache Ant 346
Apache Maven 346
bundles 374
Button group
about 276
composition structure, creating 279-281
conclusion 281
core concepts 276
GWT modules 277
prerequisites 276
widget sets 277-279

C

caption property, AbstractComponent 81
CDI Utils
about 22, 298
CDI injection 302, 303
components declaration 305
conclusion 305
core concepts 299

DI use-cases 299
event observers 303, 304
Inversion of Control and Dependency Injection 299
prerequisites 300
reference link 305
using 300-302
center() method 85
Clara
about 281
behavior, adding 283
complex components, creating 284
conclusion 285
data sources 284
inflating 283
limitations 284
prerequisites 282
using 282
XML files 282
ClientConnector interface 79
client part, Vaadin architecture 57, 59
client server 9, 10
client-server communication, Vaadin architecture 56
client-server synchronization, Vaadin architecture 60
client-side extensions
about 316
connector architecture 316
Cloud 389
CloudBees 391
cloudlet 392
Cloud offering levels
IaaS 389
PaaS 389
SaaS 390
Cloud setup 391
application server and JDK 393
balancing 392
database and cache 393
miscellaneous 394
resources 392
collapsing column 188
columns, tables
about 187
collapsing 188

footer 189, 190
generating 191-195
header 189, 190
ordering 191
reordering 191
row header column 190
table properties, formatting 191-195
width 188

commit/discard features 166, 167

commit() method 153

compareToIgnoreCase() method 172

compareTo() method 172

component 78

component class design
about 78
AbstractClientConnector 80
AbstractComponent 80
component interface 79
Component interface 79
diagram 78
MethodEventSource interface 80

component composition
about 307
custom components, designing 311
graphic composition 311
manual composition 308

component error handling 237, 239

Component interface
about 79
ClientConnector 79
Serializable 79
Sizeable 79

CompositeValidator 94

concrete indexed containers 173-175

connection, SQL container 246, 247

connector architecture, client-side extensions 316

connectors
client connectors 316
server connectors 316

container components 181-185

container datasource
about 177
items, displaying 178, 179
new items, handling 179
null items 180

containers
about 168
concrete indexed containers 173-175
filterable containers 169
filtering capability 169
hierarchical 176
item sorter 172, 173
ordered container 170, 171
sorting capability 169

Context and Dependency Injection 298

converters
about 90
StringToBooleanConverter 90
StringToDateConverter 90
StringtoDoubleConverter 90
StringtoFloatConverter 90
StringtoIntegerConverter 90
StringtonumberConverter 90
working 90

Cross-Site Scripting 231

CSS 12

CSS layout, UI components 115

custom component
about 308
designing 311

CustomComponent class 308

custom component composition
components, adding 309

custom layout, UI components 115

D

database compatibility, SQL container
248-253

data binding
about 147, 149
properties 148

data binding, properties
buffering 148
editor component 148
renderer component 148

data tier 9

Data Transfer Object 339

data, Vaadin
containers 176
entity abstraction 149

DefaultErrorHandler method 240
Dependency Injection (DI) 299
deployed portlet
 updating 373
description property, AbstractComponent
 81
detailed view, add-ons presentation 273
differences, portlet project
 portal proprietary files 362
 portlet deployment descriptor 360-362
discard() method 153
div placeholder 232
Document Object Model (DOM) 12
drag-and-drop capabilities, tables
 accept criterion 202
 DragSource 201
 drop target model 201
 Transferable 200
DragSource 201
drop target model design 201
dynamic view providers, navigator 227, 229

E

EclipseLink 260
Eclipse, setting up
 Eclipse bundled with WTP, downloading
 28
 server runtime, creating 32
 Vaadin plugin, installing 30, 31
Eclipse Vaadin project
 creating 32-34
 servlet mapping 34, 35
 testing 35
 WTP, adding to Eclipse 37, 39
 WTP, checking 36
ECMAScript 12
editor component 148
EmailValidator 94
EndlessRefresherRunnable
 about 265
 features 265
entity abstraction
 about 149
 container 168
 Item 156
 Property interface 149, 150

error correction, Hello OSGi
 Vaadin bootstrap 384
 Vaadin servlet 383, 384
error messages 236
error type notifications 107
error view, navigator 227
event-driven model
 about 125
 events, in Java EE 127
 observer pattern 125, 126
event firing 138
event model, Navigation API 230
event model, Vaadin

 about 129
 architectural considerations 137
 events, outside UI 136
 standard event implementation 129
 view, expanding 135

events outside UI, Vaadin event model
 about 136
 user change event 136

extension connector
 architecture, diagrammatic
 representation 318
 working 318-321

Ext-GWT 22

F

fat client 11
 updating, through update sites 19
features, SQL container
 initialization 246
 paging 246
 programmatic filtering 246
 programmatic ordering 246
 transaction management 246

field group
 about 161, 162
 captions, modifying 165
 commit/discard features 166, 167
 field types, configuring 163, 165
 functionalities 161

field types
 configuring 163, 165

files
serving, from HTTP server 366
serving, from portal 366
filterable containers 169
filters 169
Flash 16
Flex 17
formats, labels
about 88
HTML 88
preformatted 88
text 88
form layout, UI components 114
fragmentation 347
free form queries 256-259

G

Gant for Grails 347
GateIn
about 357
configuring, for Vaadin 365
downloading 357
installing 357
launching 357, 358
portlet, deploying in 363
portlet, using 363
general error handling 240-244
generated code 74
getButton() method 135
getClickedValue() method 311
getContent() method 82
getConvertedValue() method 91
getState() method 322
getType() method 87
getUriFragment() 103
getValue() method 199
Glassfish
about 376
bundles, deploying 376
Google App Engine 79
Google Chrome Frame
URL 232
Google Web Toolkit (GWT) 19, 57
Gradle for Groovy 347

graphic composition
about 311
limitations 315
Visual Designer, using 312, 313
visual editor, setting up 311, 312
grid layout, UI components 113
GWT dev mode 58
GWT Incubator project
about 329, 330
client classes 331, 332
prerequisites 330
server component 330
GWT widget wrapping
about 327
example 329
GWT Incubator project 327

H

HasComponents interface 82
HbnContainer 259
Hello cloud
registration 391
Hello OSGi
about 381
bundle, creating 381, 382
deploying 383
errors, correcting 383
horizontal and vertical layouts, UI components 113
HTML, labels 88
HTTP protocol, Vaadin architecture 56
HTTP server
files, serving from 366
HttpServletRequest object 218
HttpServletResponse object 218
HttpSession object 218
humanized notifications 107
HyperSonic SQL (HSQLDB) 357
Hyper Text Markup Language (HTML) 10

I

IDE-managed server
creating 61, 62
installation, verifying 63
tab, selecting 61

iframe tag 231, 233
immediate property, AbstractComponent 81
improvements, Twaattin
 Ivy dependencies 265
 Twaattin UI 266, 267
 Tweet refresher behavior 268
 Twitter service 269
inaccessible OSGi libraries 382
influencing factors, architectural considerations
 application size 140
 expected lifespan 140
 QA 140
 team experience 140
initial view, navigator 227
init() method 71
inputs, text fields
 about 97
 field 98, 99
 focusable 98
Integrated Development Environment (IDE) 19
integrated frameworks
 about 21
 JPA 22
 levels 21
integration level 1, Vaadin 21
integration level 2, Vaadin 22
integration level 3, Vaadin 22
integration platforms
 Liferay 22
IntelliJ IDEA
 about 39
 installing 27
 setting up 40-42
 using 39
 Vaadin 7 plugin, adding 43
IntelliJ IDEA Vaadin project
 context-root 48
 creating 44, 45
 deploying 46
 framework support, adding 46
 result, adjusting 45
 servlet mapping 49
 testing 47
 Vaadin version, changing 47
Inversion of Control (IoC) 22, 299
isEmpty() method 99
isModal() method 86
isRoot() method 205
Item, entity abstraction
 about 156
 field group 161, 162
 method property 156
 right level of abstraction 156-160
item sorter 172, 173
Ivy dependencies 265

J

JavaEE API
 accessing 217
 servlet request 218, 219
 servlet response 220, 221
 wrapped session 222
Java EE events, event-driven model
 about 127
 UI events 128
Java Network Launching Protocol (JNLP) 18
Java Persistence API. See JPA
JavaScript Native Interface (JSNI) 58
JavaScript wrapping
 about 333
 example 334
 requisites 334
JavaScript wrapping example
 core 335
 prerequisites 335
Java Server Faces 13
JavaServer Pages (JSP) 13
JBoss GateIn 356
JDBC connection pool 246
JDiskReport
 URL 19
JMS 265
joins, SQL container
 about 253
 free form queries 256-259
 references 254, 256
JPA 22
JPA Container
 about 260, 275, 285
 concepts 286

conclusion 298
features 296
prerequisites 286
similar add-ons 285
URL 285
using 296, 297

JSON 56

JSON message format, Vaadin architecture 56, 57

JSR-168 356

JSR-286 356

JSR specifications, for portlet API
about 356
JSR-168 356
JSR-286 356

JWS

about 18, 19
URL 18

K

Keep It Simple and Stupid (KISS) principle 139

Knopflerfish

URL 375

L

LabelConnector 317

labels

about 86
class hierarchy diagram 87
formats 88
Property interface 87

layouts, UI components

about 112
abstract layout 113
advanced layouts 114
component container 112
selecting 116
simple layouts 113
types 113

lifecycle layer 375

Liferay 22, 357

limitations, graphic composition

about 315
restricted compatibility 315
rigid structure 316

top level element 315
load balancer 392
location, Windows 85
logic tier 8
long-lived HTTP connections 260
long polling 260

M

mainframes 9

Maven

in Vaadin projects 348
Twaattin, migrating under 353

Maven dependencies management
using 348, 349

Mavenize Vaadin projects 348, 349

Maven projects

Vaadin support 349-352

Maven tooling 347

MethodEventSource interface 80

MethodProperty class 156

modality, Windows 86

Model View Controller (MVC) design pattern 139

Model-View-Controller paradigm 13

modularity layer 374

N

navigateTo(String) method 229

Navigation API

about 222
event model 230
overview 230
URL fragment 223
views 223

navigator

about 224, 226
dynamic view providers 227-229
error view 227
initial view 227

nominal embedding, Vaadin

about 232
bootstrap script 233
div proper 232
page headers 232
UI initialization call 233-235

notification class 106

notifications
about 107
additional properties 108
displaying 109
error type notifications 107
humanized notifications 107
tray notifications 107
warning notifications 107

notify() method 127

O

OAuth
about 206
URL, for info 206

ObjectProperty 152

observer pattern enhancements,
event-driven model
event 127
event details 127
event types 127

observer pattern, event-driven model
enhancements 126

onMessage() method 127

ordered container 170, 171

OSGi 374, 375

OSGi bundles, deployment to Glassfish
file system deployment 377
prerequisites 376
Telnet deployment 376
web console deployment 377, 379

OSGi platforms
Apache Felix 375
Knopflerfish 375

OWASP
URL 231

P

page
about 102
navigation 103
title 103
URL fragment 103

page headers 232

passesFilter(Object, Item) method 169

Plain Old Java Objects (POJO) 139

Play Framework 55

portal
about 355
files, serving from 366

portal configuration
themes 365
widgetsets 365

portal proprietary files 362

portlet
about 355
adding, to page 363
deploying, in GateIn 363
using, in GateIn 363

portlet container 355

portlet deployment descriptor 360-362

portlet development strategies
about 372
deployed portlet, updating 373
portal debug mode 372
portlet servlet-compatible 372

portlet project
creating 359
differences 359
similarities 362

portlet specifics
handling 368-371

portlet.xml file 360

preformatted, labels 88

prerequisites, JPA Container
application server 288
data source, creating 292-295
dependency 286
Glassfish, integrating with Eclipse
IDE 289-292
model 286, 288

presentation tier 8

programmatic sorting 195

Project Object Model (POM) 347

project sources, Twaattin killer application
about 119
login screen 121
timeline screen 121
UI 120

property formatter 152, 153

Property interface, entity abstraction
AbstractProperty 151

modifications, handling 153, 155
ObjectProperty 152
property formatter 152, 153
Property interface, labels 87
pseudo-shareable pool 246
push innards, server push 261

Q

Quartz 265
queries, SQL container 246, 247
query delegates 247

R

Rake for Ruby 347
real-world error handling
about 236
component error handling 237, 239
error messages 236
general error handling 240-244
real-world text field example 102
references 254, 256
refresh() method 246
RegexpValidator 93
related add-ons, SQL container 259, 260
removeListener() method 134
removeWindow(Window) method 84
renderer component 148
rich applications
about 8
application tiers 8
limitations, of thin client 11
rich client approaches
Ajax 16
fat client, deploying 18
fat client, updating 18
GWT 19
plugin 16, 17
rich clients 15
Rich Internet Application (RIA) 8
rootItemIds() method 205
row header column 190

S

SBT for Scala 347
Search Engine Optimization (SEO) 230
server part, Vaadin architecture 59, 60
server push
about 260
example 263-265
installation 262
push innards 261
Server RPC
about 324
architecture 324, 326
service layer 375
Service Locator 299
service() method 70
servlet request, JavaEE API 218, 219
servlet response, JavaEE API 220, 221
setChildrenAllowed() method 205
setConverter() method 91
setDraggable(boolean) method 85
setHeight() method 314
setLocation(String) method 103
setModal(boolean) method 86
setMultipleSelect(true) method 199
setParent() method 205
setPosition(Position) method 108
setPositionX(int) method 86
setPositionY(int) method 86
setSelectable(true) method 199
setSplitPosition() method 117
setTitle(String) method 103
setValue() method 150
setWidth() method 314
setXXXUndefined() method 314
shared state 322
similarities, portlet project 362
simple layouts, UI components
about 113
form layout 114
grid layout 113
horizontal and vertical layouts 113

SingleComponentContainer interface 82
Single Page Interface (SPI) 103, 230
Single Responsibility Principle 133
Sizeable interface 79
size, UI components
 about 110
 dimension, setting 111
sorting, tables
 programmatic sorting 195
 user sorting 195
source code 74
split panels
 about 117
 properties 117
Spring 22
SQL container
 about 244
 architecture 245
 connection 246, 247
 database compatibility 248-253
 features 246
 joins 253
 queries 246, 247
 related add-ons 259, 260
SQLContainer class 245
standard event implementation, Vaadin event model
 about 129
 abstract component 134
 event 130
 event class hierarchy 130
 EventRouter class 134
 listener interfaces 131, 132
 listeners, managing 133
 method event source details 133, 134
 typed events 130, 131
 Window component 132
StringToBooleanConverter 90
StringToDateConverter 90
StringtoDoubleConverter 90
StringtoFloatConverter 90
StringtointegerConverter 90
StringtonumberConverter 90
Struts 13
summarized view, add-ons presentation 273

T

table drag-and-drop 202, 203
tables
 about 186
 columns 187
 drag-and-drop capabilities 199
 editing 197, 199
 selecting 199
 sorting 195
 structure 186
 viewpoint 196
text field class hierarchy
 about 99
 cursor 101
 input prompt 101
 null 100
 password field 100
 properties 100
 real-world example 102
 selection 101
 simple text field 100
 text area 100
text fields
 about 89
 buffer, changing 96
 BufferedValidatable 97
 conversion 90, 91
 inputs 97
 validation 92
text format, labels 88
thin client applications approach
 limitations 11
thin client applications drawbacks
 about 11
 browser compatibility 14
 page flow paradigm 14
 unrelated technologies 12, 13
thin clients 10
third-party content
 about 104
 browser window opener 105
 resources 104
tier migration
 about 9
 client server 9

mainframes 9
thin clients 10
traditional polling 260
Transferable 200
TransferableImpl 200
tray notifications 107
trees
 about 204
 child node 205
 collapse feature 204
 expand feature 204
 item labels 205
 parent node 205
Twaattin
 about 118, 276
 adaptations 206
 code, updating 339
 component, designing 338
 design 118
 event-driven model 125
 improvements 265
 integrating 385
 login screen 118
 main screen 118
 migrating, under Maven 353
 prerequisites 206
 refining 205
 running, outside IDE 353
 source code 343
 status component 340, 342
 status converter 342
 timeline screen 342
 updated sources 206
Twaattin integration
 bundle plugin 385
 multiplatform build 388
Twaattin killer application
 project setup 119
 project sources 119
Twaattin migration, under Maven
 about 353
 dependency management, enabling 354
 Maven dependency management, using
 for optimization 354
 preparing 353
Twaattin UI 266, 267

Twaattin updated sources
 about 206
 column generators 212
 login behavior 208
 login screen 208
 timeline screen 208, 210
 tweets refresh behavior 210
Twaattin, Vaadin event model
 about 141
 login behavior 143
 login screen 142
 logout behavior 145
 project sources 141
 timeline window 144
 UI 141
Tweet refresher behavior 210, 268
Twitter4J API 339
Twitter service 269

U

UI.close() method 72
UI components
 laying out 110
 layouts 112
 size 110, 111
 split panels 117
UI events
 about 128
 client events 128
 client-server events 128
 limitations 129
UI initialization call 233-235
UI provider
 about 300
 class diagram 301
UIs
 about 81
 component hierarchy diagram 82
 HasComponents 82
 panel class 83
 SingleComponentContainer 82
 theming 83
 UI class 83
UI, Vaadin. *See* **Vaadin UI**

UML (Unified Modeling Language) 84
URL fragment, Navigation API 223

user messages

about 106
notification class 106
notifications additional properties 108
notifications, displaying 109

user sorting 195

V

Vaadin

about 7, 20, 53
architecture 55
basic embedding 231
benefits 23, 24
client-side extensions 316
component composition 307
components class design 78
concerns 23
data 149
embedding 230, 231
features 20-24
GateIn, configuring for 365
labels 86
market status 20
nominal embedding 232
page 102
philosophy 54, 55
terminology 78
text fields 89
third-party content 104
UIs 81
user messages 106
using, in real world 23
Windows 84

Vaadin 7

connectors 316

Vaadin add-ons

about 271
add-on presentation 273
add-ons search 272
Button group 276
CDI Utils 298
Clara 281
JPA Container 285

stability levels 272
typology 272

Vaadin application

browsing 66
creating 49
debug mode 67, 68
deploying 60
deploying, inside IDE 61
deploying, outside IDE 65
entry point, declaring 51
generated code 74
out-of-the-box helpers 66
restarting 69
servlet class, declaring 51
servlet mapping, adding 50
servlet mapping, declaring 51, 52
source code 74
stream redirection, to servlet 69
surface 69
surface, scratching 73
using 66

Vaadin application deployment, in IDE

about 61
application, adding 63
IDE-managed server, creating 61
server, launching 63, 64

Vaadin application deployment, outside IDE

about 65
server, launching 65
WAR, creating 65

Vaadin architecture

about 55
client part 57-59
client-server communication 56
client-server synchronization 60
HTTP protocol 56
JSON message format 56
server part 59, 60

Vaadin bootstrap 384

Vaadin bundling 380

Vaadin GWT architecture

about 327
client-side 327, 328
server-side 327
widget styling 329

Vaadin, in Eclipse
about 27
Eclipse, setting up 28

Vaadin integration
about 21
integrated frameworks 21
integrated platforms 22

Vaadin JPAContainer summary 273

Vaadin libraries
adding 49

Vaadin modularization 381

Vaadin OSGi use cases
about 380
modularization 381
Vaadin bundling 380

Vaadin projects
creating 348

Vaadin request handling 70

Vaadin servlet 383, 384

VaadinServlet.service() method 70, 71

Vaadin session 222

Vaadin support, for Maven project
Twaattin, running outside IDE 353
widget compilation 352, 353

Vaadin support, for Maven projects 349-352

Vaadin UI
about 71
configuration 72
features 71
session 72

validatable
about 95
isValid() method 95
validate() method 95

validation, text fields 92

validator hierarchy
diagrammatic representation 93

validators
about 92
CompositeValidator 94
EmailValidator 94
error message 94
RegexpValidator 93

viewpoint, tables
about 196
responsiveness, improving 197
viewpoint change event 196

views, Navigation API 223

view, Vaadin event model
buttons 135, 136
expanding 135

Visual Designer
about 311
position and size, managing 314
using 312, 313

W

W3C CSS1 specifications 110

warning notifications 107

Web Archive 65

WebGL 334

widget 78

Windows
about 84
basic configuration 85
customizing 85
location 85
modality 86
structure 84

wrapped session, JavaEE API 222

WTP 27

X

XmlHttpRequest object 14



**Thank you for buying
Learning Vaadin 7
*Second Edition***

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.

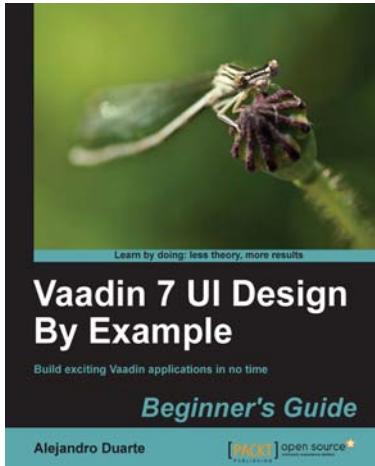
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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.

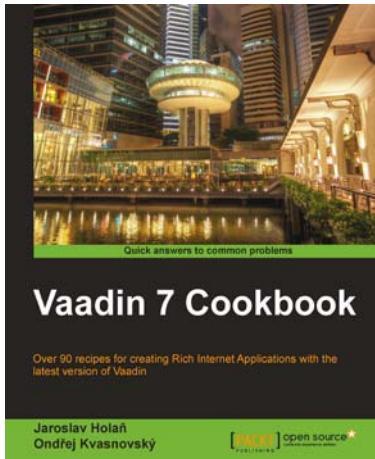


Vaadin 7 UI Design By Example Beginner's Guide

ISBN: 978-1-78216-226-1 Paperback: 253 pages

Build exciting Vaadin applications in no time

1. Learn how to develop Vaadin web applications while having fun and getting your hands dirty
2. Develop relevant and unique applications following step-by-step guides with the help of plenty of screenshots from the start
3. The best available introduction to Vaadin with a practical hands-on approach and easy to read tutorials and examples



Vaadin 7 Cookbook

ISBN: 978-1-84951-880-2 Paperback: 404 pages

Over 90 recipes for creating Rich Internet Applications with the latest version of Vaadin

1. Covers exciting features such as using drag-and-drop, creating charts, custom components, lazy loading, server-push functionality, and more. Tips for facilitating the development and testing of Vaadin applications. Enhance your applications with Spring, Grails, or Roo integration.

Please check www.PacktPub.com for information on our titles

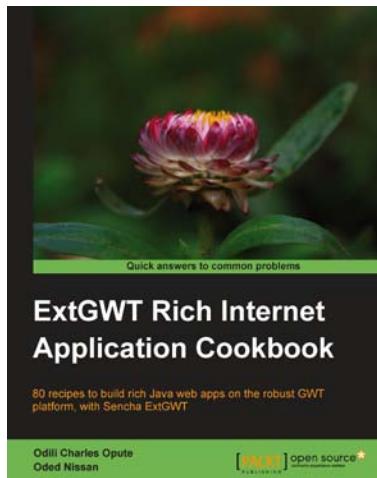


Google App Engine Java and GWT Application Development

ISBN: 978-1-84969-044-7 Paperback: 480 pages

Build powerful, scalable, and interactive web applications in the cloud

1. Comprehensive coverage of building scalable, modular, and maintainable applications with GWT and GAE using Java
2. Leverage the Google App Engine services and enhance your app functionality and performance
3. Integrate your application with Google Accounts, Facebook, and Twitter



ExtGWT Rich Internet Application Cookbook

ISBN: 978-1-84951-518-4 Paperback: 366 pages

80 recipes to build rich Java web apps on the robust GWT platform, with Sencha ExtGWT

1. Take your ExtGWT web development skills to the next level
2. Create stunning UIs with several layouts and templates in a fast and simple manner
3. Enriched with code and screenshots for easy and quick grasp

Please check www.PacktPub.com for information on our titles