The Definitive Guide to Spring Web Flow

Erwin Vervaet

Apress®

The Definitive Guide to Spring Web Flow

ISBN-13 (pbk): 978-1-4302-1624-7

ISBN-13 (electronic): 978-1-4302-1625-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

SpringSource is the company behind Spring, the de facto standard in enterprise Java. SpringSource is a leading provider of enterprise Java infrastructure software, and delivers enterprise class software, support, and services to help organizations utilize Spring. The open source-based Spring Portfolio is a comprehensive enterprise application framework designed on long-standing themes of simplicity and power. With more than five million downloads to date, Spring has become an integral part of the enterprise application infrastructure at organizations worldwide. For more information visit: `www.springsource.com`.

Lead Editor: Steve Anglin
Technical Reviewers: Keith Donald, Karl David Moore
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Kylie Johnston
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Pat Christenson
Proofreader: Lisa Hamilton
Indexer: Becky Hornyak
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

About the Author xi
About the Technical Reviewers xiii
Acknowledgments xv
Introduction xvii

CHAPTER 1 Introducing Spring Web Flow 1
CHAPTER 2 Getting Started 25
CHAPTER 3 Spring Web Flow's Architecture 51
CHAPTER 4 Spring Web Flow Basics 63
CHAPTER 5 Advanced Web Flow Concepts 131
CHAPTER 6 Flow Execution Management 197
CHAPTER 7 Driving Flow Executions 239
CHAPTER 8 Testing with Spring Web Flow 271
CHAPTER 9 The Sample Application 289
CHAPTER 10 Real-World Use Cases 309
CHAPTER 11 Extending Spring Web Flow 325
EPILOGUE 351
REFERENCES 357
INDEX 359

Contents

About the Author xi
About the Technical Reviewers xiii
Acknowledgments xv
Introduction xvii

CHAPTER 1 Introducing Spring Web Flow 1

Free Browsing 4
Controlled Navigation 6
Navigational Control 7
State Management 12
Modularity Concerns 14
Traditional Solutions 15
Spring Web Flow 20
Summary 22

CHAPTER 2 Getting Started 25

Downloading Spring Web Flow 25
Runtime Requirements 26
Build System Integration 26
Manual Integration 27
Integration with Ivy 29
Integration with Maven 30
Spring Jumpstart 30
Hello World 34
Spring Web Flow in a Development Environment 39
Installing an IDE 39
"Hello World" in Eclipse 42
Using Spring IDE 44
Spicing Up Hello World 46
Summary 49

CHAPTER 3 Spring Web Flow's Architecture 51

Language 52
Domain Vision Statement for Spring Web Flow 53
Architectural Layers 54
Execution Core 55
Execution Engine 58
Executor 59
Test Support 60
System Configuration 60
Summary 60

CHAPTER 4 Spring Web Flow Basics 63

Designing Flows 64
UML State Diagrams 66
Your First Flow Definition 67
Flow Builders 72
The XML Flow Builder 74
The Java Flow Builder 75
Choosing an Appropriate Flow Builder 79
Defining Flows 80
Flows 80
States 82
Transitions 85
Flow Definition Structure 89
Flow Executions 90
Flow Sessions 92
The Flow Execution Context 94
The Request Context 95
Implementing Actions 102
AbstractAction 104
MultiAction 106
Deploying Actions 107
Basic State Types 112
View States 113
Action States 118
End States 120

Flow Definition Registries 121
XML Flow Definition Registries 123
Java Flow Definition Registries 127
Combining Flow Definition Registries 128
Summary 129

CHAPTER 5 Advanced Web Flow Concepts 131

OGNL 131
OGNL by Example 133
OGNL in Action 136
The Conversion Service 153
Annotating Flow Definition Artifacts 156
Handling Exceptions 158
View Selections 162
Empty String 164
viewName 164
redirect:viewName 164
externalRedirect:url 165
flowRedirect:flowId?input1=value1&. . .&inputN=valueN 166
bean:id 166
Custom View Selectors 167
Data Binding and Validation 168
The FormAction 172
Subflows 178
Inline Flows 179
Flow Sessions Revisited 180
Declaring an Input-Output Contract 181
Mapping Input and Output Arguments 185
Enhancing the “Enter Payment” Flow 189
Flow Start and End Actions 190
The Complete “Enter Payment” Flow Definition 192
Summary 195

CHAPTER 6 **Flow Execution Management** 197

Introducing Flow Execution Repositories 197
Flow Executors 200
Launching a Flow Execution 201
Resuming a Flow Execution 203
Refreshing a Flow Execution 204
Request Handling 206
Configuring a Flow Executor 207
Flow Execution Repositories 211
Conversation Management 213
The Simple Repository 216
The Single Key Repository 217
The Continuation Repository 220
The Client Continuation Repository 224
Selecting a Repository 226
Flow Execution Listeners 227
Listener Invocation Examples 231
Listener Configuration 235
Summary 237

CHAPTER 7 **Driving Flow Executions** 239

Flow Executor Integration 240
Spring Web Flow View Development 242
Model Data 243
Building Requests 245
Host Framework Integrations 250
Spring Web MVC 251
Spring Portlet MVC 253
Struts 255
JavaServer Faces 258
Summary 270

CHAPTER 8 Testing with Spring Web Flow 271

Unit Testing 272
MockRequestContext 272
MockRequestControlContext 272
MockExternalContext 272
MockParameterMap 273
MockFlowExecutionContext 273
MockFlowSession 273
MockFlowServiceLocator 273
Testing with Mock Objects 273
Flow Execution Testing 275
Testing Java Flow Definitions 276
Testing XML Flow Definitions 277
Testing the "Enter Payment" Flow 278
Using Flow Execution Listeners 282
Integration Testing 284
Summary 287

CHAPTER 9 The Sample Application 289

Functional Requirements 290
Downloading and Building 290
The Domain Model 293
Application Setup 295
The Presentation Tier 297
Spring Web MVC Setup 297
Implementing the "Enter Payment" Use Case 302
Internationalization 305
Exception Handling 306
Page Layout 307
Summary 307

CHAPTER 10 Real-World Use Cases ... 309

Accessing the Host Environment ... 309
Flow Definition Parameterization ... 311
Leveraging Listeners ... 314
Securing a Flow ... 314
A Global Back Transition ... 315
Breadcrumbs ... 317
Load and Stress Testing ... 319
Spring Web Flow and AJAX ... 322
Summary ... 323

CHAPTER 11 Extending Spring Web Flow ... 325

Common Extension Points ... 325
Using Bean References ... 326
Extending the Flow Definition Constructs ... 327
Custom Flow Builders ... 333
A Database-Backed Conversation Manager ... 334
A Flow Servlet ... 342
Building Spring Web Flow ... 346
Summary ... 348

EPILOGUE ... 351

Spring Web Flow 2 ... 352
Choosing Between Spring Web Flow 1 and 2 ... 355
Concluding Thoughts ... 355

REFERENCES ... 357

INDEX ... 359

About the Author

ERWIN VERVAET is an independent consultant based in Leuven, Belgium. Erwin has been using Java since its inception and has extensive experience applying it in a wide range of application domains. He also runs his own software and consultancy company, Ervacon (`http://www.ervacon.com`).

Erwin enjoys writing, teaching, and speaking about Java- and Spring-related subjects. As the originator of the Spring Web Flow project, he currently co-leads its development together with Keith Donald.

About the Technical Reviewers

KEITH DONALD is a principal and founding partner of SpringSource (formerly Interface21), the company behind Spring. He is best known in the Spring community for creating Spring Web Flow with Erwin Vervaet. At SpringSource, Keith is the lead of the web application development products team. His team, based in Melbourne, Florida, helps sustain the development of Spring Web MVC and Web Flow and their associated integrations and is also responsible for future innovations in the domain of web application development frameworks.

Keith, together with Jay Zimmerman of NoFluffJustStuff Software Symposiums, is also the director of the Spring Experience conference series. He is responsible for the technical content of the conference, which takes place in Florida each December.

In addition, Keith is the principal architect behind SpringSource's state-of-the-art Spring training curriculum. This curriculum has provided practical training on Spring to over 3,000 students worldwide.

Keith, an experienced enterprise software developer and mentor, has built business applications for customers spanning a diverse set of industries including banking, network management, information assurance, education, and retail. He is particularly adept at translating business requirements into technical solutions.

Keith's blog can be found at `http://blog.springsource.com/main/author/keithd`.

KARL DAVID MOORE is a software engineer with over six years' commercial development experience. He holds a first class honors degree in software engineering from Sheffield Hallam University.

Karl has been a Spring Framework user since early 2004 and is an active contributor to the Spring forums, with over 8,000 posts. During his career, Karl has been extensively involved in all aspects of software development and has a particular passion for refactoring legacy systems. This has helped him foster a very strong interest in test-driven development, code quality, and attention to detail.

Karl enjoys researching and developing approaches to aid and simplify development, and evangelizing about techniques and tools to improve the skills of other developers.

You can find Karl's LinkedIn profile at `http://www.linkedin.com/in/karldmoore`.

Acknowledgments

This book, and Spring Web Flow itself, only exist thanks to the input and help of many different people.

Spring Web Flow was lucky and gained an active community right from the start. A special thanks goes out to the early adopters and forum members, whose invaluable feedback has helped mold Spring Web Flow into what it is today. Several people deserve special credits for their active involvement in the project: Juergen Hoeller, Colin Sampaleanu, Rod Johnson, Ben Hale, Christian Dupuis, J. Enrique Ruiz, C_sar Ordiñana, Rossen Stoyanchev, Jeremy Grelle, Rob Harrop, Seth Ladd, Colin Yates, Steven Devijver, Greame Rocher, Sam Brannen, Maxim Petrashev, Marten Deinum, and Dave Syer. It goes without saying that countless others also deserve credit.

Writing a book is a big undertaking that could not be done without the support and feedback of several people. I would like to thank Keith Donald for his invaluable input into Spring Web Flow, as well as for his review efforts. Karl David Moore also went above and beyond the call of duty, doing extraordinarily thorough chapter reviews and helping me polish my quirky English. The fine people at Apress that molded this book into its current form also deserve accolades: Heather Lang, Laura Cheu, and Kylie Johnston. Several other people also directly or indirectly contributed to this book: Philip Van Bogaert, Henri Shih, Kris Meukens, and Mike Seghers. Thank you all.

A very special thanks goes out to my wife Bieke. She tirelessly kept the kids busy and did more than her fair share while I slaved away at this book. I can honestly say that without Bieke's support, this book would not have been possible.

Introduction

When I started working on Spring Web Flow at the end of 2004, web applications already accounted for a large part of the Java enterprise development space. I had used Struts on several projects at that point but always felt something was missing. Working with a proprietary framework on a few projects in the financial industry sparked my interest. The framework I was using included a work flow engine, a fairly typical feature for frameworks targeted at high-end enterprise applications. What was novel about it, however, was that the work flow engine could also be used to define *page flows* in web applications. This brought a refreshingly intuitive approach to Java web application development.

Using a state diagram as the basis for page flows in web applications seemed much more natural than the request-centric solutions offered by the mainstream frameworks of the time. This was especially true for the more complex use cases that required the user to pass through a number of different steps in the completion of a business process. Over the course of 2004, I had been learning about the Spring Framework, which was gaining momentum at the time, and had been impressed by its design and implementation quality. I set out to add a page flow controller to the Spring Web MVC framework and created what would later become Spring Web Flow.

Initially, Spring Web Flow focused on using a state-diagram–based approach to make defining page navigation in web applications easy and intuitive. This gave web application developers a powerful way to express page navigation rules. Expressive page flow definitions also highlighted the need for better *navigational control*. The infamous Back button problem caused all sorts of difficulties in web applications that tried to control page navigation. Spring Web Flow clearly needed to address this issue.

Around that time, I read an article discussing the use of continuations to solve navigational problems in web applications. This struck me as fitting very well with Spring Web Flow's flow execution model and provided the missing link. Spring Web Flow now combined two very attractive and complementary features:

- An intuitive and easy to use method of *expressing page navigation rules* in web applications.
- A powerful and robust *navigational control* system.

Bringing a third attractive feature to the table required only a small step:

- Encapsulate page flows as black-box *application modules* with a well-defined input-output contract.

Spring Web Flow has come a long way in the last two years. It has grown from a simple "flow controller" for the Spring Web MVC framework into "a next generation Java web application controller framework that allows developers to model user actions as high-level modules called flows. The framework delivers improved productivity and testability while providing a strong solution to enforcing navigation rules and managing application state" (Johnson et al 2003). The next generation of the framework, Spring Web Flow 2, introduces exciting new features to help you build and run rich web applications. Clearly, Spring Web Flow is now a mature project that is used in many production deployments and has an active user community.

I wrote this book not only to teach you how to work with Spring Web Flow but also to help you understand the rationale and motivation behind the framework. I hope you enjoy reading this book and have fun working with Spring Web Flow!

About the Spring Web Flow Project

The original incarnation of the Spring Web Flow project was a small open source project started by Erwin Vervaet in October 2004 called Ervacon Spring Web Flow (`http://www.ervacon.com/products/springwebflow`). The project caught the attention of Keith Donald, one of the developers on the Spring Framework team, and became an official Spring Framework subproject in February of 2005. After almost two years of active development and a number of preview releases and release candidates, the project released its first production ready 1.0 version in October of 2006. This version is the subject of this book.

Building on the solid foundation set by Spring Web Flow 1, development continued on the next generation of the product. Spring Web Flow 2, released in June of 2008, underwent a few architectural changes allowing it to more seamlessly integrate into a rich web environment. It has impressive support for JSF and AJAX techniques and further simplifies the flow definition syntax.

Spring Web Flow uses the well-known Apache 2 license, a free/open source software license (Apache Software Foundation 2004). The Apache 2 license is used by many other open source projects, such as the Spring Framework itself and the Apache HTTP Server. It allows use of Spring Web Flow for any purpose, whether commercial or noncommercial, and even allows for modification and redistribution.

Spring Web Flow is sometimes mistakenly written as "Spring WebFlow." This confusion has its origin in the Spring Web Flow package name, `org.springframework.webflow`, which writes `webflow` in one word. "Spring Web Flow" is also often abbreviated as "SWF."

The official Spring Web Flow home page is located at `http://www.springframework.org/webflow`. It is an essential resource for Spring Web Flow users wanting to keep an eye on the evolution of the project. If you're still left with questions after reading this book,

you'll have a good chance of getting an answer on the very active Spring user forums: `http://forum.springframework.org`. The Ervacon Spring Web Flow Portal (`http://www.ervacon.com/products/swf`) also offers useful information such as Spring Web Flow tips and tricks and a practical introduction.

About This Book

This book aims to teach you how to work with Spring Web Flow. It covers both basic and advanced use cases and provides an in-depth reference to all features Spring Web Flow currently offers. You'll also learn to extend the framework to take it beyond its out-of-the-box feature set. Once you've finished this book, you'll be able to call yourself a Spring Web Flow expert!

Spring Web Flow 1 and Spring Web Flow 2

Before we continue, one important point needs to be clarified. This book deals with Spring Web Flow 1. The next generation of the framework, Spring Web Flow 2, is subject matter for another book.

Versions 1 and 2 are essentially two separate products. The core concepts are the same, but the two versions are quite different technically. As a result, Spring Web Flow 2 is not backward compatible with Spring Web Flow 1. Moving from version 1 to version 2 would be a migration rather than a simple upgrade. This book's Epilogue discusses the differences between the two versions in more detail and will help you decide which version is best for you.

Spring Web Flow 1 will be referred to as just "Spring Web Flow" in this book, omitting the version number. When talking about Spring Web Flow 2, the version number will be explicitly mentioned.

Target Audience

This book is intended to be a reference for both new and advanced Spring Web Flow users. If you're a new user, you'll learn how to get started using Spring Web Flow and leverage all of its powerful features. As an advanced user, you'll learn about extending the framework and many of its best practices, and you'll find this book provides very interesting insights into the design of Spring Web Flow.

Before reading this book, you should have a solid understanding of Java and Java web application development including topics such as servlets and JavaServer Pages (JSP). Many of the samples in this book use the Spring Web MVC framework. However, if you're

familiar with any web Model-View-Controller (MVC) framework (for instance Struts or WebWork), you should have no problem following along.

A basic knowledge of the Spring Framework and its guiding principles, such as the Inversion of Control pattern and dependency injection, is also assumed. You don't need to be a Spring expert to read this book, but you'll have an easier time if you have at least played around with Spring classes like `ApplicationContext` and `BeanFactory` and understand how Spring wires together beans.

Rather than bloating this book with a detailed description of Java web applications, Spring Web MVC, or even Spring in general, I refer you to the Apress books *Beginning Spring 2: From Novice to Professional* by Dave Minter (2005); *Pro Spring 2.5* by Jan Machacek, Jessica Ditt, Aleksa Vukotic, and Anirvan Chakraborty (2008); and *Pro Java™ EE Spring Patterns: Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework* by Dhrubojyoti Kayal (2008). Find other Apress books at `http://www.apress.com`.

Overview

This book provides both introductory material and in-depth coverage of Spring Web Flow. The following overview will help you focus on the chapters most relevant to you. You can also read this book from cover to cover, and I recommend doing so if you're new to Spring Web Flow. If you're already familiar with the framework, you can skip the first two chapters and head directly to Chapter 3.

Chapter 1: Introducing Spring Web Flow

This chapter takes a high-level view and examines the problem Spring Web Flow was designed to solve. It will explain the context in which Spring Web Flow lives and give you a conceptual understanding of what exactly Spring Web Flow is.

Chapter 2: Getting Started

After the broad introduction of Chapter 1, this chapter will help you to hit the ground running. It explains all the practical details to get started using and experimenting with Spring Web Flow. The environment setup in this chapter should enable you to easily follow along with the examples covered in later chapters and to *try things for yourself*.

Chapter 3: Spring Web Flow's Architecture

Chapters 1 and 2 superficially touch on some of the Spring Web Flow concepts. Chapter 3 digs a little deeper. It explains the Spring Web Flow architecture, giving you a detailed understanding of the different subsystems involved and setting the stage for an in-depth study of the Spring Web Flow feature set in the following chapters.

Chapter 4: Spring Web Flow Basics

This chapter covers basic Spring Web Flow features, needed in most, if not all, use cases. You'll learn how to design and implement a web flow using both the XML- and Java-based flow definition languages.

Chapter 5: Advanced Web Flow Concepts

As a follow up to the basic concepts covered in Chapter 4, this chapter will detail more advanced functionality. It explains how to reuse flows as subflows from inside other flows, realizing Spring Web Flow's promise of modularity. Handling HTML form data is also covered.

Chapter 6: Flow Execution Management

When working with Spring Web Flow, most of the development effort revolves around defining flows. Also very important, however, is understanding how Spring Web Flow manages flow executions and the associated data; these topics will be discussed in this chapter.

Chapter 7: Driving Flow Executions

This chapter focuses on integrating Spring Web Flow into hosting frameworks like Spring Web MVC and JSF and driving flow executions from those environments. Developing views for a web flow will also be discussed.

Chapter 8: Testing with Spring Web Flow

Unit testing Spring Web Flow applications is explained in this chapter. You'll learn how to perform integration tests with your flow definitions and how to test flow artifacts (such as actions) in isolation.

Chapter 9: The Sample Application

To give you an example of a nontrivial application that combines both free browsing and controlled navigation situations, this chapter will document a sample application: Spring Bank. Spring Bank is a simple electronic banking application that allows users to do things such as manage their bank accounts or enter payments. Use cases of this application will be used throughout this book to help explain and illustrate Spring Web Flow's feature set.

Chapter 10: Real-World Use Cases

This chapter covers some frequently asked questions related to use cases occurring in the real world. You'll learn about such things as securing your flows, tracking breadcrumbs, or stress testing Spring Web Flow applications.

Chapter 11: Extending Spring Web Flow

The last chapter covers extending and customizing Spring Web Flow. You'll also learn how to build Spring Web Flow from the sources.

Epilogue

To conclude this book, the epilogue leaves you with some parting thoughts and takes a look at what's new and improved in Spring Web Flow 2.

Typographical Conventions

This book uses simple and easy-to-understand typographical conventions. All text to be interpreted in a literal sense, like Java class names, code fragments, file names, or XML elements uses a `fixed width font`. *Italics* indicate topics of particular importance, and new or important pieces of a code fragment are highlighted in **`boldface fixed-width text`**. Commands to be entered on the command line also use a **boldface font**.

Many of the program listings presented in this book have been formatted for readability and to make them fit nicely on the page. In some cases, additional line breaks had to be introduced. A backslash (`\`) line continuation marker is used whenever an extra line break had to be added.

About the Examples

All the sample applications presented in this book use Java 5, Servlet API 2.4, and JSP 2.0. Make sure you have a Java 5 Development Kit (JDK) and an appropriate application server or Servlet engine installed on your computer (for instance, Tomcat 5 or Jetty 6).

Spring Web Flow 1.0.6, together with Spring 2.5.4, was used to develop the sample applications. Future Spring Web Flow 1.x and Spring 2.x versions will be compatible. Chapter 2 provides a detailed overview of setting up a build and development environment you can use to run the sample applications.

The Spring Bank sample application, together with the other samples discussed in this book, can be found in the public Ervacon Subversion repository at `https://svn.ervacon.com/public/spring`. You can browse the source code by just pointing your browser at this address. Alternatively, you can check out the entire source tree using any Subversion client, for instance, TortoiseSVN (`http://tortoisesvn.tigris.org`) if you are using Microsoft Windows. The source code for this book is also available to readers at `http://www.apress.com` in the Downloads section of this book's home page. Please feel free to visit the Apress web site and download all the code there.

CHAPTER 1

Introducing Spring Web Flow

Enterprise applications, and more specifically web applications, form a large part of all applications developed using Java. Most Java developers have worked on a Java web application at one point or another in their careers. As such, it comes as no surprise that there is a large variety of so-called web model, view, controller (MVC) frameworks to choose from. Well known examples include the following:

- Struts from the Apache Software Foundation (`http://struts.apache.org`).
- Spring Web MVC, the web MVC framework built on top of the Spring Framework (`http://www.springframework.org`).
- WebWork, a web framework developed by the OpenSymphony project (`http://www.opensymphony.com/webwork`). WebWork was used as the basis for Struts 2, and its development continues under that umbrella.

In the last few years, these classic request-based frameworks have gotten more and more competition from component-based web MVC frameworks. Key players in this arena follow:

- JavaServer Faces (JSF), a framework developed by the Java Community Process (JCP) as Java Specification Requests (JSRs) 127, 252, and 314 (`http://java.sun.com/javaee/javaserverfaces/`)
- Tapestry from the Apache Software Foundation (`http://tapestry.apache.org`)

Request-based frameworks treat the HTTP request as a first-class citizen. Request handling is typically done by application actions or controllers and results in the rendering of a new page. Component-based frameworks, however, abstract the HTTP request and encapsulate application functionality in reusable components. This approach is very similar to the one taken by desktop graphical user interface (GUI) toolkits. Web application components react to events and manipulate their own internal state, leaving the rendering of a page up to the controlling framework.

Despite their differences, all of these frameworks use the *Model, View, Controller* (MVC) design pattern to structure web applications and make them easier to understand and maintain. MVC was originally developed in the Smalltalk community to structure the GUI of desktop applications. It has also proven to be very effective in web application development.

The MVC pattern tries to separate the concerns of the *view* (user interface) from those of the *model* (domain or business model) by introducing the *controller* as an intermediary. In web applications, the controller processes incoming requests by delegating to business components (for instance, a service layer) and preparing model data for rendering by a selected view. The view is typically implemented using a templating system such as JavaServer Pages (JSP) or Velocity.

■ **Note** Web applications use a slight variation of the MVC design pattern sometimes called Model 2 architecture or web MVC. In the original MVC triad, the controller is not coupled to the view. Instead, the view acts as an observer of the model, receiving event notifications when the model is changed (Gamma et al 1995).

Most implementations of the MVC pattern in web applications introduce an additional component: the *front controller* (Fowler 2003). The front controller is technical in nature and coordinates request processing by enforcing a well defined request processing life cycle: it maps a request onto a particular controller and renders the view selected by that controller. The front controller can also manage common concerns like security and internationalization. Well known front controller implementations are the Struts `ActionServlet`, the Spring MVC `Dispatcher Servlet`, or the JSF `FacesServlet`. From the application's point of view, the controller is in the driver's seat. Once the front controller has selected the appropriate controller for a request, that controller decides how to interface with business components, which view to display, and the model data to render. This process is presented graphically in Figure 1-1.

Web MVC frameworks help us develop efficient, structured, and maintainable *web applications*: software applications that are usable on the World Wide Web (WWW). Such an application could, for instance, dynamically serve data coming from a database, in contrast to serving static HTML web pages.

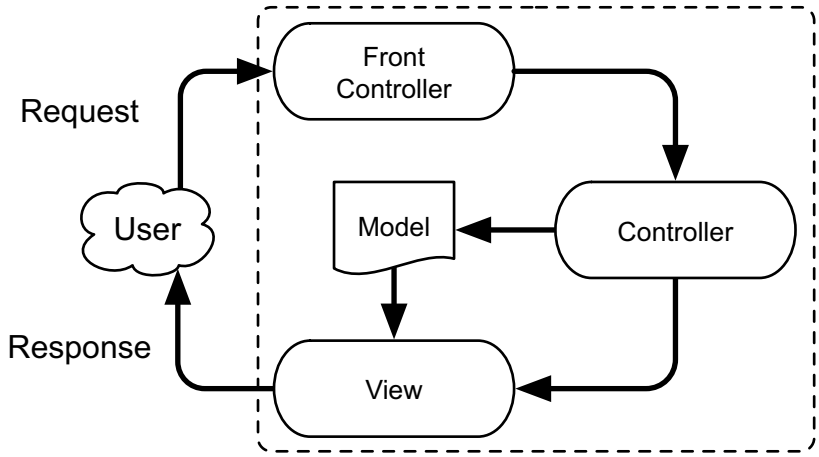

Figure 1-1. *Web MVC*

Web applications are pushing the boundaries of web technology. We have become accustomed to having complex conversations with these applications, requiring interactive dialogs spanning multiple requests. An airline ticket booking process is one example: you select a flight, indicate seating preferences, and enter payment information within the scope of an interaction with the application that spans several pages.

Java web application development based on the MVC design pattern is well understood. As always in the Java world, there is heated debate about the pros and cons of particular frameworks. The basics are agreed on by most people involved, however, and some of the older frameworks such as Struts have been used successfully in production deployments for several years. Given the enormous amount of effort that has already gone into the development of these frameworks, and the applications based on them, it comes as a huge surprise that not a single mainstream Java web application development framework offers a truly compelling way of implementing complex conversations.

Spring Web Flow aims to fill this void. It serves as a web application controller component focused entirely on the definition and execution of complex conversations in a web application. Instead of competing with the well established web MVC frameworks already available, Spring Web Flow integrates into those frameworks. It serves as the "C" in "MVC": a controller orchestrating interaction with business components, preparing model data, and selecting views (the role of Spring Web Flow as the controller has changed slightly in Spring Web Flow 2, which builds on top of Spring Web MVC).

To better understand the goals and benefits of Spring Web Flow, you must first understand the problems it tries to solve and the complexities involved in the solution. Let's investigate these next.

Free Browsing

Historically, the Web was designed as a system that allowed users to browse through informational pages linked together using hyperlinks. Information is annotated using Hypertext Markup Language (HTML) to allow correct presentation in a browser and embed anchors that link the information to other resources or pages. Each page is uniquely identified in this web using a Uniform Resource Identifier (URI).

To keep the system scalable and efficient, the World Wide Web was set up as a stateless client-server environment. Client browsers retrieve information from web servers using the GET method defined by the Hypertext Transfer Protocol (HTTP). This simple setup allows caching of information both on the client side and on intermediary proxy servers. The web servers themselves can be easily scaled horizontally, adding additional systems to form a cluster, because they do not have to maintain information associated with individual users.

This simple stateless architecture was clearly a great success. The World Wide Web has scaled to unprecedented size, and millions of people are now familiar with its concepts. Browsers allow you to efficiently move between pages, offering navigational aides such as a browsing history with Back, Forward, and Refresh buttons. Users can directly type semantically meaningful URIs like `http://en.wikipedia.org/wiki/Plato` into their browsers and bookmark URIs to easily return to those pages at a later time. Browsing is not constrained in any way, and users surfing the World Wide Web are limited only by their own interests, time, and imagination.

The architecture of the World Wide Web is often called *RESTful*, since it follows the Representational State Transfer (REST) style of software architecture for distributed hypermedia systems, as originally formulated by Roy Fielding in his doctoral dissertation (2000):

> *Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.*

Many technical publications also use the term "RESTful" to describe applications using simple web technologies such as HTML, HTTP, and URIs. Furthermore, meaningful, user-readable URIs are often referred to as *REST-style URIs* or *URLs*.

When the World Wide Web was gaining momentum, designers quickly realized that it would be far more compelling if it allowed user interaction. To accommodate this new requirement, core World Wide Web standards were enhanced with interactive elements:

- HTTP was extended with additional request methods: POST allows information to be submitted to the server. PUT and DELETE allow clients to add resources to the server or remove them again.

- Elements allowing user interaction were added to HTML. The best known example is the HTML `<FORM>` tag, which allows you to include parameter key-value pairs in a request sent to the server (typically in a POST request).

- Using technology such as cookies, servers can track a user through an entire HTTP session spanning multiple requests, thus working around the stateless nature of HTTP and allowing servers to customize content for each user.

All of these techniques and enhancements combine to result in a system capable of supporting *web applications*: useful, interactive computer applications that leverage the ease of use and ubiquity of the World Wide Web while realizing important benefits, such as these:

- There's no need to install custom software on the client, making large-scale deployment trivial.

- All clients always use the same, up-to-date version of the application.

- New subscription-based billing models are possible, opening up possibilities for new revenue streams.

- Users are familiar with the technologies involved—web browsers, HTML pages, and so on—often alleviating the need for training.

Of course, there is also a cost attached to these new possibilities:

- Having HTTP sessions makes scaling a web application more difficult than scaling a traditional web server. Simple solutions often cause *server affinity*, that is, require a user with an active session to always use one server from a cluster, thus preventing failover and effective load balancing. Still, this problem is on the server side, so it can be tackled by knowledgeable people, and the client browser can remain simple and lightweight.

- Web applications cannot offer the rich interactive user experience found in desktop GUI applications. This downside turned out to be somewhat of an advantage to users, since it forced web applications developers to create simple and intuitive interfaces for their applications.

With web developers, users, and businesses gaining more and more experience with web applications, the requirements of these applications become ever more challenging. Today we see web applications using techniques such as Asynchronous JavaScript and XML (AJAX) to offer a richer user experience. Web applications also go to great lengths to control a user's actions while interacting with the application. The next section explores this topic in detail.

Controlled Navigation

Most modern web applications not only require free browsing but also have more demanding use cases requiring *controlled navigation*: the users' navigational freedom needs to be constrained to carefully guide them to the completion of a business process. There are many, well known examples of such controlled navigation use cases.

- One example is an airline ticket booking process, involving steps such as flight selection, seating preferences, frequent flyer policies, and payment information entry. Then too, most travel agencies allow you to book not only a flight but also an associated hotel room and rental car.

- Also, electronic banking has become commonplace in the last few years, allowing customers to use web applications to make payments, buy stock options, or manage their credit cards.

- Many governments are using web technology and web applications to allow citizens to perform tasks electronically, for instance, submitting tax declarations. Such processes are bound by legislation and are often extremely complex, requiring the user to follow an exact navigational flow corresponding to his or her situation.

These web applications potentially deal with sensitive user data or even the user's own money. It is therefore of utmost importance that the applications are very stable, secure, and robust. Users should feel comfortable interacting with the application, trusting it to do the right thing, even in situations where a user has accidentally used the browser's Back or Refresh buttons.

All of the Java web application frameworks discussed previously allow you to develop simple web applications with relative ease. Most use the HTTP session to manage state associated with a particular user and make it easy to work with pages containing HTML forms using techniques such as automatic data binding. However, they do not go beyond these abilities to provide support to tackle the difficult use cases where navigation needs to be controlled and users need to be carefully guided to the completion of a certain task.

This shortcoming is unfortunate, since correctly implementing a completely controlled navigation in a web application is a difficult problem to solve. Three key issues need to addressed:

- Navigational control
- State management
- Modularity

Let's investigate each of these topics in a little more detail.

Navigational Control

The first question to be answered when implementing a controlled navigation in a web application is, "How do you control the user's navigational freedom?" As explained in the "Free Browsing" section, the World Wide Web was designed as a system promoting free browsing and unconstrained navigation. Browsers offer lists of favorite bookmarks, allowing a user to bookmark a page of particular interest and jump directly to that page again at later point in time. History lists maintained by browsers, and the Back, Forward, and Refresh buttons make it easy for users to go back to pages they visited earlier. A user can even open multiple browser windows or tabs at the same time to compare information, for instance.

You might think that all this functionality is just a convenience offered by modern browsers, but in reality, the free browsing spirit is engrained in the specifications of the core technologies powering the World Wide Web. The following extract from the HTTP 1.1 specification (section 13.13 of RFC 2616) suggests browsers should not reload a page from the server when the user accesses the browsing history, for instance, using the Back button:

> *User agents often have history mechanisms, such as navigation buttons and history lists, that can be used to redisplay an entity retrieved earlier in a session.*

> *History mechanisms and caches are different. In particular, history mechanisms should not try to show a semantically transparent view of the current state of a resource. Rather, a history mechanism is meant to show exactly what the user saw at the time when the resource was retrieved.*
>
> *By default, an expiration time does not apply to history mechanisms. If the entity is still in storage, a history mechanism should display it even if the entity has expired, unless the user has specifically configured the agent to refresh expired history documents.*

This definition suggests that the browser history is intended as a client-side navigational aid, enhancing the user's web surfing experience. At the time of this writing, most popular browsers, such as Mozilla Firefox (`http://www.mozilla.com/firefox`) and Microsoft Internet Explorer (`http://www.microsoft.com/ie`), do not strictly follow the specification and will use cache control settings to decide whether or not to reload a page from the history. The Opera browser (`http://www.opera.com`) is an example of a browser strictly complying with the specifications. When you click the Back button, Opera will redisplay the previous page exactly as it was shown before, without reloading it from the server.

It's clear that all of this free browsing support doesn't complement use cases requiring controlled navigation. Unfortunately, it is not generally possible to disable the browser's navigation history, its bookmarking capability, or its ability to open new windows or tabs. You are therefore forced to deal with the problem head-on: you must handle situations where the user uses free browsing conveniences when having a controlled conversation with a web application. Some of the situations that need to be handled follow:

- What happens when a user bookmarks a page in the middle of the conversation? We can't stop the actual bookmarking, but how should the application react when the user uses the bookmark to jump back into the conversation? In most cases, the answer will be that the application should produce an error informing the user that the conversation has expired or ended, possibly allowing the conversation to be restarted. The entry point into the conversation or task might be bookmarkable, but the internal pages typically are not. In other situations, it will be necessary to keep track of the conversation for a long period of time and allow users to jump back into it and continue where they left off.

- How does an application handle refresh requests or moving back or forward in the browsing history? Ideally, a refresh request is idempotent, not causing any side effects with repeated use and allowing the user to freely refresh pages. Handling back and forward navigation, however, is more difficult.

- A less common situation involves the user opening two browser windows on the same application, typically to compare results or evaluate alternatives. How does a web application deal with this? Care needs to be taken to avoid interference or double submits.

Applications can ignore these problems and just ask the user not to use the browser's Back button when starting a process requiring controlled navigation. This approach is obviously naive and brittle, as users are accustomed to surfing around the Internet, frequently clicking the Back or Refresh buttons. When a mistake is made, web applications should be able to handle it in a stable and predictable way.

Applications with a well known and controlled user group, like intranet applications, can sometimes avoid these problems altogether. By deploying a specialized or customized web browser, developers can completely disable all navigational aides. This is obviously not an option for web applications running on the Internet, where users use a wide variety of web browsers. Some Internet web applications try to simulate this by running the application in a special browser window that contains no button bar or other embellishments. This helps, but breaks easily, if for instance, the user presses the Backspace button or a special mouse button to navigate backward in the browser's history.

Incomplete navigational control and users accidentally using the navigational aids offered by the browser also cause another well known problem in web applications: the dangerous double submit.

The Double Submit Problem

You have already seen the two central request methods supported by the HTTP protocol used on the World Wide Web. A request method indicates the intention of a request to the server.

The GET method allows a client to retrieve a resource (page) from the server. HTTP defines GET to be safe, meaning that it should only ever be used for information retrieval and does not change the state of the server. In other words, a GET request is idempotent and can be repeated (refreshed) any number of times, always producing the same result. GET was the first request method supported by HTTP and was originally the only one. It is still the most frequently used request method today.

The POST method, on the other hand, allows a request to submit data to the server for processing, possibly causing side effects such as updating records in a database. A user submitting a POST request is responsible for the changes caused to the state of the server. POST requests are not safe or idempotent and therefore cannot be refreshed or bookmarked. Historically, POST requests are initiated by clicking buttons on web pages, in contrast with textual links used for GET requests. Using buttons visually differentiates POST requests from GET requests, highlighting their sensitive nature to the user.

A user using the browser navigation history or Refresh button can accidentally repeat a POST request, causing server-side processing to occur twice. This repeat processing has the potential to result in incorrect or unwanted side effects and is known as the *double submit* problem. Imagine a user resubmitting a "confirm purchase" request, resulting in two purchases! Web browsers are aware of this problem and will display a warning to the

user when they try to reissue a POST request, such as the following one from Internet Explorer:

In contrast, refreshing pages loaded using a GET request is allowed, and no warning will be shown. Although these warnings provide an extra layer of security, they will make an application feel brittle to the end user: accidentally clicking the Back button causes an unnerving message about possible duplicate transactions!

A real solution for the double submit problem is provided by the POST-REDIRECT-GET idiom (Jouravlev 2004). When a web application sends an HTML page in response to a POST request, that page will look like a normal web page to the user, causing the user to think it is bookmarkable and refreshable. Instead of sending the page data directly in reply to a POST request, the POST-REDIRECT-GET idiom states that a web application should issue a REDIRECT request in response to a POST request, causing a subsequent GET request. This is illustrated in Figure 1-2.

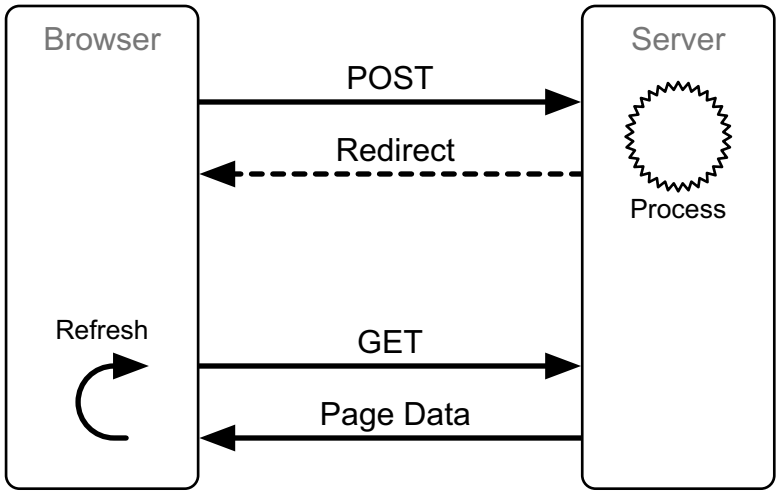

Figure 1-2. *The POST-REDIRECT-GET idiom*

A redirect instructs the web browser to look for the response elsewhere using a GET request. HTTP 1.1 defines result code 303, "See Other", for exactly this purpose (from section 10.3.4 of RFC 2616):

> *The response to the request can be found under a different URI and should be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource.*

For backward compatibility with older web browsers that do not support HTTP 1.1, many web applications don't use the 303 response and instead use the 302, "Found", code (from section 10.3.3 of RFC 2616):

> *The requested resource resides temporarily under a different URI.*

All web browsers in common use today react to the 302 code in exactly the same was as to the 303 code.

A browser receiving a redirect response will know that the requested resource is to be found at another location. As a result, it will not add the original request to its browsing history and instead add only the redirected GET request. This elegantly solves the double submit problem, since the redirected request can be safely refreshed (it's a GET request!). Furthermore, the navigational history will contain only idempotent GET requests, allowing the user to use the browser Back and Forward buttons or bookmark pages without causing alarming warning messages or side effects on the server.

The HTTP specification defines the semantics for GET and POST requests. Unfortunately, these semantics are enforced neither by the HTTP protocol nor by web servers. Because of this, many web sites and web applications abuse GET requests to submit data to the server and cause side effects. These compromises are often driven by visual requirements and a desire to make a web page look as beautiful as possible. JavaScript gives web developers a lot of freedom to post forms by clicking textual links or to get a resource from the server by clicking a button. Image buttons blur the presentational distinction between GET and POST requests even further, giving you complete visual freedom.

Since a GET request can also be used to submit data to the server for processing, possibly causing side effects, the POST-REDIRECT-GET idiom can be reformulated in a more general way as *redirect after submit*: a web application should issue a redirect in response to every submit, be it using a POST or a GET.

Redirecting after every submit allows you to build web applications that behave correctly even when the user accidentally uses functions such as the browser Back button. Another interesting problem is protecting the application against a malicious user trying to short-circuit a conversation with a web application.

Short-Circuiting Conversations

Most complex conversations in web applications require a linear progression, typically working toward the completion of a business process, such as filling in a tax form. It is

crucial that the user enters all the required data and does not jump ahead, skipping important information, or hack around sensitive checks. The server must carefully track a user's progression to ensure the task is completed as defined by the business requirements.

Having a client submit information about its location in a conversation is insufficient. A hacker could easily manipulate those request parameters to short-circuit the flow, possibly compromising the application and the data it manages.

Web applications have no control over the environment the client is using. Normal users will be using the popular browser variants. A malicious user, however, may be using other tools to craft special requests to trick the server into doing things it's not supposed to do. A well known rule among web application developers is that the server should always validate user input data, even when client-side (JavaScript) checks are used. This is also true of navigational control. The server needs to take responsibility and ensure the client follows the defined navigation rules.

In summary, controlling user navigation in a web application is clearly a hard problem to solve. The crux of the problem is the fact that we are trying to enforce navigational constraints in an environment that was explicitly designed for free browsing.

State Management

Complex conversations within web applications not only mandate a controlled navigation but also involve state management. These subjects are often intertwined: a user accumulates state while progressing through the navigational flow prescribed by a particular business process. When the browser's navigational facilities are used, the application needs to handle the possible impact on the data associated with the conversation:

- When using the Back button, should the application undo (or forget) the edits already done on later pages, or should that information be retained for reuse when the user progresses again? Both alternatives are quite common.

- As browsers such as Opera do not contact the server at all when going back in the navigation history, how do we keep server state synchronized with the client? The client could be seeing stale data that is different from the current values managed by the server, which could lead to confusing situations or possibly even data corruption.

- An application needs to ensure that two browser windows for the same conversation do not interfere with each other, potentially overwriting data entered in one window with the data captured in the other window.

- When a conversation ends, associated data should be cleaned up to avoid memory leaks and prevent the same task from being completed multiple times.