

# **Struts 2 from Square One**

## **Part 1 - Carpe Diem**

**Ted Husted**

---

# Struts 2 from Square One : Part 1 - Carpe Diem

Ted Husted

Copyright © 2006-2007 Husted dot Com, Inc.



## Note

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, Draft v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>). Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder. Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Open Publication License v1.0, 8 June 1999

### I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

Copyright (c) [year] by [author's name or designee]. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author(s) names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

### II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

### III. SCOPE OF LICENSE

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

**SEVERABILITY.** If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

**NO WARRANTY.** Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

### IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
-

- 
2. The person making the modifications must be identified and the modifications dated.
  3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
  4. The location of the original unmodified document must be identified.
  5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

## V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

## VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.



## Note

Some portions of this work are based on material provided under the Apache License [<http://apache.org/licenses/LICENSE-2.0.txt>] by the Apache Software Foundation. Such portions are Copyright (c) 2000-2007 Apache Software Foundation.

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

## TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

### 1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the

---

---

direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

1. You must give any other recipients of the Work or Derivative Works a copy of this License; and

2. You must cause any modified files to carry prominent notices stating that You changed the files; and

3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative

---

---

Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.



---

# Table of Contents

What do you want to deploy today? .....	xv
I. Carpe Diem .....	1
1. Apache Struts 2 From Square One .....	2
1.1. Why do we need Struts? .....	2
1.2. Is Struts the best choice for every project? .....	2
1.3. How does Struts make web development easier? .....	3
1.3.1. What architecture does Struts provide to applications? .....	3
1.3.2. What ties the Action and Result together? .....	4
1.3.3. What services do the Struts Tags provide? .....	6
1.4. What are the objectives of this book? .....	7
1.5. What audience does the book serve? .....	7
1.6. Are there any prerequisites? .....	7
1.7. How is the book organized? .....	8
1.7.1. Part 1 - Carpe Diem .....	8
1.7.2. Quick Reference .....	8
1.7.3. What will be covered by future editions? .....	8
1.8. How are the sessions organized? .....	11
1.8.1. What is the MailReader? .....	11
1.8.2. What does the MailReader workflow look like? .....	12
1.8.3. Does the MailReader use a Database? .....	13
1.8.4. What are use cases? .....	13
1.9. Is the source code available? .....	14
1.10. Review: Apache Struts 2 From Square One .....	14
1.10.1. Segue .....	15
1.11. Training Guide .....	15
1.11.1. Presentation - Struts 2 from Square One .....	15
1.11.2. Accomplishments .....	16
1.11.3. Use Case: Read Mail .....	16
2. Building Web Applications .....	17
2.1. Why build web applications? .....	17
2.1.1. Browsers? sites? applications? hypertext? Can we start from the beginning? .....	17
2.1.2. How is an intranet different than the Internet? .....	19
2.1.3. Are web applications so different? .....	19
2.1.4. How does the Common Gateway Interface (CGI) work? .....	20
2.1.5. How do we build web sites? .....	21
2.1.6. How much HTML do we need to know to create web applications? .....	23
2.1.7. Review: Why build web applications? .....	24
2.1.8. How are Java web applications organized? .....	25
2.1.9. Review: How are applications organized? .....	27
2.2. Review: Building Web Applications .....	27
2.2.1. Segue .....	28
2.3. Training Guide .....	28
2.3.1. Presentation - Building Web Applications .....	28

---

2.3.2. Coding Exercise - "Hello" .....	28
2.3.3. Prerequisites .....	29
2.3.4. Exercises .....	29
2.3.5. Accomplishments .....	29
2.3.6. Use Case: Hello (Setup development tools) .....	30
3. Building Struts 2 Applications .....	32
3.1. Can we use a conventional design? .....	32
3.1.1. Why bother? .....	35
3.1.2. Review: Can we use a conventional design? .....	36
3.2. Why use frameworks? .....	36
3.2.1. Review: Why use frameworks? .....	38
3.3. What value does Struts add? .....	38
3.3.1. What are the key workflow components? .....	39
3.3.2. How do we handle actions? .....	40
3.3.3. How do we handle results? .....	43
3.3.4. How do we display dynamic data? .....	43
3.3.5. What happens during the workflow? .....	44
3.3.6. Review: What are the key workflow components? .....	47
3.3.7. What are the key architectural components? .....	47
3.3.8. What are Interceptors? .....	48
3.3.9. Review: What are Interceptors? .....	56
3.3.10. What is the Value Stack? .....	56
3.3.11. Review: What is the ValueStack? .....	62
3.3.12. What is OGNL? .....	62
3.4. Review: Building Struts 2 Applications .....	67
3.4.1. Segue .....	67
3.5. Training Guide .....	67
3.5.1. Presentation - Building Struts 2 Applications .....	68
3.5.2. Coding Exercise - "Welcome" .....	68
3.5.3. Prerequisites .....	68
3.5.4. Exercises .....	68
3.5.5. Hint Code .....	69
3.5.6. Accomplishments .....	69
3.5.7. Extra Credit .....	69
3.5.8. Use Case: Welcome .....	69
4. Migrating to Apache Struts 2 .....	71
4.1. What are some frequently-asked questions about the Struts 2 release? .....	71
4.1.1. What is Struts 2? .....	72
4.1.2. How are Struts 1 and Struts 2 alike? .....	72
4.1.3. What's changed in Struts 2? .....	72
4.1.4. Is Struts 1 obsolete? .....	73
4.1.5. Is it difficult to migrate from Struts 1 to Struts 2? .....	73
4.1.6. Why are POJOs important? .....	74
4.1.7. Review: How are Struts 1 and Struts 2 alike? .....	74
4.2. Where should a migration begin? .....	74
4.3. Is the Struts 2 configuration file different? .....	77

---

4.4. Why are there so many changes to the Struts configuration? .....	80
4.5. Do the Action classes change too? .....	81
4.6. What about the tags? .....	83
4.7. Can we still localize messages? .....	84
4.8. How do we change Locales in Struts 2? .....	87
4.9. Does Struts 2 use the Commons Validator? .....	88
4.10. Is that all there is? .....	91
II. Quick Reference .....	93
A. Quick Reference .....	94
A.1. Maven Configuration .....	94
A.1.1. Maven Artifact ID .....	94
A.1.2. Maven Snapshot Repositories .....	94
A.2. Use the Source .....	94
A.2.1. Action.java .....	95
A.2.2. ActionSupport.java .....	95
A.2.3. Interceptor.java .....	95
A.2.4. Validator.java .....	95
A.2.5. ValidatorSupport.java .....	95
A.2.6. default.properties .....	96
A.2.7. default.xml   validators.xml .....	96
A.2.8. struts-default.xml .....	96
A.2.9. struts-message.properties .....	96
A.2.10. struts-portlet-default.xml .....	96
A.2.11. struts-tags.tld .....	96
A.3. XML Document Types (DocType) .....	97
A.3.1. Struts Configuration XML DocType reference .....	97
A.3.2. XWork Validator XML DocType Reference .....	97
A.4. Common Configurations .....	97
A.4.1. web.xml .....	97
A.4.2. struts.xml .....	98
A.4.3. struts-plugin.xml .....	98
A.5. Struts 2 web.xml elements .....	98
A.5.1. Typical Struts 2 web.xml .....	98
A.5.2. Fully-loaded Struts 2 web.xml with optional elements .....	98
A.6. Configuration Examples .....	100
A.6.1. struts.properties with alternate settings .....	101
A.6.2. Struts Configuration XML with HelloWorld action .....	101
A.6.3. Struts Configuration XML with Wildcards .....	101
A.7. Interceptors .....	102
A.8. Validators .....	102
A.8.1. Methods excluded from validation .....	103
A.8.2. Specifying the methods excluded from validation .....	104
A.9. Result Types .....	104
A.9.1. Setting a default result type .....	104
A.10. Exception Handling .....	105
A.10.1. Specifying a global exception handler .....	105
A.10.2. Inserting error and exception messages .....	105

---

A.11. OGNL .....	105
A.12. Static Content .....	107
A.12.1. Adding other static content locations .....	108
A.13. Struts Tags .....	108
A.14. Struts Tag examples .....	109
A.15. Key Best Practices .....	112

---

# List of Figures

1.1. Struts 2 Architecture .....	3
1.2. Page With Struts Tags (a complete form) .....	7
1.3. Say Howdy .....	12
1.4. Who-Who Are You? .....	12
1.5. Known Subject .....	13
2.1. The First Homepage .....	18
2.2. HTML (HyperText Markup Language) .....	24
2.3. Show me the Classes .....	25
3.1. Modeling the Problem Domain .....	32
3.2. Bridging the Model Gap .....	33
3.3. Conventional MVC .....	33
3.4. Multiple V's in MVC .....	34
3.5. Desktop MVC .....	34
3.6. Enterprise MVC .....	35
3.7. Struts 2 Workflow .....	44
3.8. Hello World! .....	46
3.9. Interceptor Gauntlet: Thank you sir! May I have another? .....	48
3.10. Value Stack: Last In First Out .....	60

---

# List of Tables

1.1. Part 2 - Plumbing Matters .....	9
1.2. Part 3 - Looking Good .....	9
1.3. Part 4 - Action Friends .....	10
1.4. Future Appendices and End Pieces .....	10
1.5. MailReader Use Cases .....	14
3.1. ActionContext properties .....	64
3.2. Expression Language Notations .....	65
3.3. OGNL Samples .....	65
3.4. OGNL Operators .....	66
A.1. key framework classes and configuration files .....	94
A.2. Essential configuration files .....	97
A.3. Default interceptor stack .....	102
A.4. Other bundled interceptors .....	102
A.5. Predefined Validators .....	102
A.6. Predefined result types .....	104
A.7. Default Result Names .....	105
A.8. Common OGNL Operators .....	106
A.9. Expression Language Notations .....	106
A.10. Control Tags .....	108
A.11. Data Tags .....	108
A.12. Form Tags .....	109
A.13. Non-Form Tags .....	109
A.14. General Tag Attributes .....	109
A.15. Tooltip Related Tag Attributes .....	109
A.16. Javascript Related Tag Attributes .....	109
A.17. Template Related Tag Attributes .....	109

---

# List of Examples

1.1. Logon XML Document .....	5
1.2. Logon Java Annotation .....	5
1.3. Page Without Tags (a partial form) .....	6
1.4. Page With Struts Tags (a complete form) .....	6
2.1. Creating dynamic content via code .....	21
2.2. Creating dynamic content via server pages .....	22
2.3. Creating dynamic content via server pages .....	22
2.4. Dynamic content via JavaScript .....	22
3.1. Welcome to the Jungle (Model 1) .....	35
3.2. Separating code and markup (Model 2) .....	35
3.3. struts.xml .....	40
3.4. HibernateAction.java .....	42
3.5. Hello.java .....	45
3.6. Hello.jsp .....	45
3.7. resources.properties .....	45
3.8. Welcome.jsp .....	46
3.9. TimerInterceptor.java .....	48
3.10. Preparable Interface and the PrepareInterceptor .....	49
3.11. Interceptor.java .....	50
3.12. Interceptor Idioms .....	53
3.13. Custom Interceptor .....	54
3.14. HibernateInterceptor.java .....	55
3.15. ModelDriven.java .....	61
3.16. ModelDrivenInterceptor.java .....	61
4.1. A web.xml with both Struts 1 and Struts 2 enabled .....	76
4.2. Struts 1 Configuration for our "Hello World" application .....	78
4.3. Struts 2 Configuration for our "Hello World" application .....	79
4.4. A Struts 1 <forward> element and a corresponding Struts 2 <result> element .....	80
4.5. Listing: Selecting a different result type .....	81
4.6. Struts 1 Hello ActionForm and Action class .....	82
4.7. Struts 2 Hello Action class (includes form properties) .....	82
4.8. Struts 1 "Hello" server page .....	83
4.9. Struts 2 "Hello" server page .....	84
4.10. Registering a Struts 2 message resource bundle via struts.properties .....	84
4.11. Hello World Resource bundles .....	85
4.12. Changes to Struts 1 Hello Action to enable internationalism .....	85
4.13. Changes to Hello Action to enable internationalism .....	86
4.14. Struts 1 "Hello" server page .....	86
4.15. Struts 2 "Hello" server page .....	87
4.16. Changing the locale in Struts 1 (LocaleAction) .....	87
4.17. Changing the locale in Struts 1 (JSP) .....	87
4.18. Changing the locale in Struts 2 (JSP) .....	87
4.19. The Struts 1 validation.xml configuration file for HelloForm (validations.xml) .....	88

4.20. The Struts 2 validation file for Hello (actions/Hello-validation.xml) .....	89
4.21. Adding an input form to Hello World for Struts 1 .....	89
4.22. A Struts 1 input form (HelloInput.jsp) .....	90
4.23. Adding an input form to Hello World for Struts 2 .....	90
4.24. A Struts 2 input form (Hello_input.jsp) .....	90
4.25. Resource bundle with localized error messages .....	91
A.1. Validator examples .....	103

---

# What do you want to deploy today?

Is there an application that you want to put out on the web? Perhaps an intranet application serving the company's internal network? Or maybe a high-volume Internet application available to the general public? Either way, it's sure to be something that you want on the web. Now!

*Struts 2 from Square One* is designed for people who want to create Java web applications, not just quickly, but correctly. Internet time has come and gone. Today's developers live in enterprise time. We need to create web applications that sizzle, but we also need to create web applications that we can maintain, release after release after release.

The Struts framework has proven the test of time. With Struts 2, we enter a new era of elegance and extensibility. Struts is the most popular Java web application framework, and Struts 2 is the best Struts yet.

Build! Deploy! Maintain! *And seize the day!*



## Note

Part 1 of the book ("Carpe Diem") is being made available as an early release. We hope to make the other parts available as single volumes as each part is completed. In the meantime, we have engineered Part 1 so that developers can still be up and running with Struts 2 as soon as possible.

### About the Author

Ted Husted is a senior member of the Apache Struts development team and served as release manager for the Struts 2 release.

Ted's speciality is building agile web applications with open source products like Struts, Spring, and iBATIS and helping others do the same. His books include *JUnit in Action*, *Struts in Action*, and *Professional JSP Site Design*. Ted provides onsite training to software development teams through the United States. Visit <http://www.StrutsMentor.com> for details.

---

# Part I. Carpe Diem

We hit the ground running by building, extending, and testing a simple "hello world" application.

<i>Apache Struts 2 From Square One</i>	We overview the Struts framework and introduce the course materials.
<i>Building Web Applications</i>	We start from square one and discuss why we build web applications and why building web applications is such a chore.  In the coding exercise, we explore and extend a simple "Hello World" application that demonstrates the basics of web application infrastructure.
<i>Building Struts 2 Applications</i>	We overview the framework in broad strokes and explore how using Struts 2 changes how we write web applications.  In the coding exercise, we implement the first use case of the MailReader application, displaying a Welcome page with links to other actions.
<i>Migrating to Apache Struts 2</i>	We compare and contrast Struts 1 and Struts 2 by migrating and extending a simple application. The sessions contain a series of mini-exercises that step through moving a Struts 1 application to Struts 2.



## Tip

Visit the Struts Mentor website [<http://www.StrutsMentor.com/>] to schedule a live training course at your location.

---

---

# Chapter 1. Apache Struts 2 From Square One

*Apache Struts 2 from Square One* is designed as a companion guide to a live training course, but you do not need to attend the course to enjoy the book. All materials provided by the course are also available to readers of the book.

Questions readers might have about the materials include:

- Why do we need Struts?
- Is Struts the best choice for every project?
- How does Struts make web development easier?
- What are the objectives of this book?
- What audience does the book serve?
- Are there any prerequisites?
- How is the book organized?
- How are the sessions organized?
- Is the source code available?

Let's cover each point in turn.

## 1.1. Why do we need Struts?

Nowadays, with the help of integrated development environments, like Eclipse, IDEA, or NetBeans, creating a simple Java web application is almost easy. But creating enterprise-ready web applications is still hard work. Java provides a broad and powerful platform for application development, both for the desktop and for the web, but, there are missing pieces. Struts provides the "gluecode" that developers need to build large, high-performance Java web applications that can be maintained and extended over the long term, either by a sole developer or by a team of developers.

## 1.2. Is Struts the best choice for every project?

If you need to write a very simple application, with a handful of pages, then you might consider a "Model 1" solution that uses only JavaServer Pages and the Java Standard Tag Library, or even JavaServer Faces. See the [java.sun.com](http://java.sun.com) website for more about these technologies.

If you are writing a complicated application, with dozens of pages, that needs to be maintained over time, then Struts can help.

## 1.3. How does Struts make web development easier?

Apache Struts 2 makes enterprise-ready web development easier in three ways:

1. by providing a flexible and extensible application architecture,
2. by providing labor-saving custom tags, and
3. and by providing a way to configure common workflows within an application.

### 1.3.1. What architecture does Struts provide to applications?

In a web application, everything revolves around the request and response protocol that drives the Internet.<sup>1</sup> Let's look at the framework from the viewpoint of handling an incoming request.



#### Note

If terms like "request and response protocol" seem confusing, skip ahead to the "Building Web Applications" session before reading the rest of this section.

**Figure 1.1. Struts 2 Architecture**



As shown by the figure *Struts 2 Architecture*, the high-level view of the framework is simple and elegant. We can summarize the request processing cycle into seven stages: Accept Request, Select Action, Push Interceptors, Invoke Action, Select Result, Pop Interceptors, and Return Response.

---

<sup>1</sup> Portions of this section are based on material provided by the Apache Software Foundation under the Apache License and modified for inclusion here.

1. *Accept Request.* First, a web browser makes a request. The request might be for a resource like a HTML page, a PDF, or (among other things) a special "Action" resource that can create a dynamic response.
2. *Select Action.* The incoming request is transferred to a Struts 2 component called a "Filter Dispatcher". Whenever this request is for a Struts "Action", our dispatcher passes the request through to the rest of the framework. (Otherwise, the request just passes through our dispatcher, untouched.)
3. *Push Interceptors.* Whenever a request enters the framework, we might want to validate input, or transfer values to an internal object, or upload a file, or any combination of similar operations. (In other words, we want to "intercept" the request to do some housekeeping before going on to the main event.) The framework provides a component called an "Interceptor" to handle operations that need to be applied to more than one request.
4. *Invoke Action.* Aside from the "Interceptor" operations, we will also want to invoke an operation unique to the request (the "main event"). For example, we might want to work with a database to store or retrieve information. After a request passes through the outer layer of Interceptors, a method on an Action class can be called to handle any special processing for the request.
5. *Select Result.* After the Action method fires, we need to tell the rest of the framework what to do next. We've handled the request, and now we need to create the response (or "result").

To bootstrap the response process, the Action method returns a string token representing the outcome of the request. Typical outcomes are terms like "success", "failure", "cancel", or something more specific, like "logon". The framework matches that token with the name of a Result component.

The Result is responsible for handling the response, either by rendering output for the browser, or by transferring control to another resource that will render the response. Most often, "another resource" will be a JavaServer Page formatted with Struts Tags.

6. *Pop Interceptors.* After the Result handler fires, the request passes back through the set of Interceptors. Some Interceptors work with incoming requests, some work with outgoing requests, and some work with both.
7. *Return Response.* The request exits the framework, and the web server sends our response back to the web browser.

## 1.3.2. What ties the Action and Result together?

A web application uses a deployment descriptor to initialize resources like filters and listeners. The web deployment descriptor is formatted as a XML document. The web container reads the XML document and creates an internal configuration object. When the container starts up, it loads and configures the components specified by the deployment descriptor.

Likewise, Struts uses an internal configuration object to tie together various parts of the application, like the Action and the Result. An application can specify the Struts configuration by using Java annotations or by providing one or more XML documents.

The example *Logon XML Document* shows a typical XML-style configuration for a logon action:

### Example 1.1. Logon XML Document

```
<struts>
  <package name="default"
    extends="struts-default">
    <action name="Logon"
      class="mailreader2.Logon">
      <result type="redirect-action">MainMenu</result>
      <result name="input">/pages/Logon.jsp</result>
      <result name="cancel"
        type="redirect-action">Welcome</result>
      <result name="expired"
        type="chain">ChangePassword</result>
    </action>
  </package>
</struts>
```

The example *Logon Java Annotation* shows the corresponding annotations for the same logon action:

### Example 1.2. Logon Java Annotation

```
@Results({
    @Result(name="success", value="MainMenu"),
    @Result(name="input", value="pages/Logon.jsp"),
    @Result(name="cancel", value="Welcome",
      type="redirect-action"),
    @Result(name="expired", value="ChangePassword",
      type="chain")})
public class Logon extends ActionSupport {
    // ....
}
```



#### Tip

Configuration by Java annotation or by XML document are not mutually exclusive. We can use either or both in the same application.

In either case, we configure the framework to invoke a certain Action class in response to a certain web address (URI). After invoking the Action, the configuration tells the framework which type of result to use for the response. The typical result is to merge a JavaServer Page template with dynamic data obtained by the Action class.

### 1.3.3. What services do the Struts Tags provide?

The Struts Tags help us create rich web applications with a minimum of coding. Often, much of the coding effort in a web application goes into the pages. The Struts Tags reduce effort by reducing code.

The example *Page Without Tags* shows what it is like to write a conventional JavaServer Page.

#### Example 1.3. Page Without Tags (a partial form)

```
<% User user = ActionContext.getContext() %>
<form action="Profile_update.action" method="post">
  <table>
    <tr>
      <td align="right"><label>First name:</label></td>
      <td><input type="text" name="user.firstname"
        value="<%=user.getFirstname() %> /></td>
    </tr>
    <tr>
      <td>
        <input type="radio" name="user.gender" value="0"
          id="user.gender0"
          <% if (user.getGender()==0) { %>
            checked="checked" %> } %> />
        <label for="user.gender0">Female</label>
      </td>
    </tr>
  </table>
  ...
</form>
```

Looking over the markup, it's easy to see why "plain vanilla" Java web development is so much work! So far, we've only coded two controls, and there are six more to go! Let's finish the form using Struts Tags.

#### Example 1.4. Page With Struts Tags (a complete form)

```
<s:actionerror/>
<s:form action="Profile_update" validate="true">
  <s:textfield label="Username" name="username"/>
  <s:password label="Password" name="password"/>
  <s:password label="(Repeat) Password" name="password2"/>
  <s:textfield label="Full Name" name="fullName"/>
  <s:textfield label="From Address" name="fromAddress"/>
  <s:textfield label="Reply To Address" name="replyToAddress"/>
  <s:submit value="Save" name="Save"/>
  <s:submit action="Register_cancel" value="Cancel" name="Cancel"
    onclick="form.onsubmit=null"/>
</s:form>
```

**Figure 1.2. Page With Struts Tags (a complete form)**



In about the same amount of code as two conventional controls, the Struts Tags can create an entire data-input form with eight controls. Not only is there less code, but the code is easier to read and maintain. The tags also support validation and localization as a first-class features. So not only is there less code and less maintenance, there is *more* utility.

## 1.4. What are the objectives of this book?

The training course presented by this book teaches developers how to use Apache Struts 2 quickly and effectively. Both practice and theory are covered in the context of working examples.

The objectives of this book are to

- Present and apply MVC application architecture
- Build a working web application
- Explore key best practices

After completing the course, learners will be equipped to create and maintain basic Struts applications, including all necessary client- side and server-side programming.

## 1.5. What audience does the book serve?

This book is intended for developers and programmers who want to learn about web application development using Apache Struts 2.

## 1.6. Are there any prerequisites?

For participants that are new to Java or web development, the session *Building Web Applications* reviews the underlying technologies.

Ideally, participants should have experience programming with Java along with a basic knowledge of HTML and HTTP. Experience with a modern Java IDE, such as Eclipse, IDEA, or NetBeans, is also beneficial.

## 1.7. How is the book organized?

The book is organized into four parts, each representing a full day of training.



### Note

Part 1 of the book is made available as an early release. We hope to make the other parts available as single volumes as each part is completed. In the meantime, we have engineered Part 1 so that developers can be up and running with Apache Struts 2 as soon as possible.

### 1.7.1. Part 1 - Carpe Diem

In *Apache Struts 2 from Square One: Part 1 - Carpe Diem*, we hit the ground running by building, extending, and testing a simple "hello world" application.

- *Apache Struts 2 From Square One*
- *Building Web Applications*
- *Building Struts 2 Applications*
- *Migrating to Apache Struts 2*

### 1.7.2. Quick Reference

We collect together the essential, everyday details that Struts developers need at their fingertips.

### 1.7.3. What will be covered by future editions?

Other parts planned for future editions of Struts 2 from Square One include: *Plumbing Matters* (Part 2), *Looking Good* (Part 3), and *Action Friends* (Part 4).

In *Part 2 - Plumbing Matters*, now that we are up and running, we will ask "How do we get there from here?"

**Table 1.1. Part 2 - Plumbing Matters**

<i>Jumpstarting JUnit</i>	The framework encourages a layered architecture, which implies a layered approach to testing. We look at interacting with the business layer through unit tests.
<i>Capturing Input</i>	Forms play a strong role in most web applications. How does Struts 2 help us harvest data submitted by a form?
<i>Validating Input</i>	Now that we are up and running, we ask "How do we get there from here?"
<i>Test-Driving Web Development</i>	Once we know the business layer is sound, we can test the presentation layer too.

In *Part 3 - Looking Good*, we will focus on the glitz and glamour of the user interface.

**Table 1.2. Part 3 - Looking Good**

<i>Mapping Workflows</i>	The action mappings work closely with the result types to guide clients through the application.
<i>Localizing Content</i>	To reach the broadest audience, some applications support more than one language. Even if the application supports one language, the message resources are an effective tool for managing shared phrases throughout the application
<i>Displaying Dynamic Content</i>	Creating interactive screens can be the most rewarding and the most frustrating part of web development. Struts Action relies on JSP tags to display application state.

No application is an island. In *Part 4 - Action Friends*, we will look at popular Struts 2 extension for creating menus and reports.

**Table 1.3. Part 4 - Action Friends**

<i>Coding Logic</i>	In most applications, the Actions either do the heavy lifting, or know someone who does.
<i>Composing Pages</i>	Most pages are composed of common characteristics that we can code once and share using SiteMesh
<i>Struts Menu</i>	As we add features to an application, we need to organize operations into a friendly set of menus. With Struts Menu, we can keep our application organized with drop-down menus, tabbed displays, and more.
<i>JasperReports</i>	As an application's user base grows, we need to know more about who is using the application and how. Let's use JasperReports to create a set of administrative reports.

Struts is the framework of choice for enterprise teams. New appendices will help cross-train workers for both HTML and Java, and new end-pieces will make it easier for newcomers to get started.

**Table 1.4. Future Appendices and End Pieces**

<i>HTML for Java Developers</i>	How much do Java developers need to know about markup? We introduce the essential HTML that a Java developer needs to know to get through the day on a typical web application project.
<i>Java for HTML Developers</i>	How much do HTML designers need to know about Java? We outline the essential Java that a HTML designer needs to know to get through the day on a typical web application project.
<i>MailReader Tour</i>	The coding exercises build the MailReader example application, step by step. The tour is a "walk-through" of the completed application.
<i>Glossary</i>	Capsule definitions of core technologies.
<i>Index</i>	Instant access to key terms and concepts.



## Tip

The companion guide may be a work-in-progress, but the onsite training course is available now. Visit Struts Mentor [<http://www.StrutsMentor.com/>] to schedule a course at your location.

Thank you for supporting the early release of Part 1, which helps to make release of the other parts possible.

# 1.8. How are the sessions organized?

Training sessions include a technology primer and a training guide. In the live training course, the primer is a slide presentation, and the guide is a handout. In the book, the primer is the main text of each chapter, and the training guide is presented as the last section of each chapter.

- The *technology primer* introduces any background theory or underlying dependencies we need to code the exercise outlined by the training guide: *What do we need to make it work?*
- The *training guide* outlines the coding exercise for each session. *How do we put it all together, exactly?*

The training guide reviews session highlights, outlines prerequisites, provides hint code, and describes the coding exercise by way of a formal use case.

- The *coding exercise* is one component of a working application. The technology primer, training guide, and use case, are all designed to support coding the exercise.
- The *use case* introduces the exercise from the client's perspective: *Why do we need to do what the exercise does?*
- Over the course of the exercises, students build and extend a "best practices" web application, based on the Struts MailReader example application.

## 1.8.1. What is the MailReader?

MailReader is an example application included with Struts framework. The example represents the first iteration of a web application that can read various POP3 and IMAP accounts. Visitors can register with the web site and setup a list of email accounts. To serve a wider audience, the application is internationalized.

The example demonstrates navigation, localization, business facade, data access objects, CRUD operations, and managing parent/child records.

## 1.8.2. What does the MailReader workflow look like?

Initially, the MailReader presents a Welcome page, as shown in the figure *Say Howdy*.

**Figure 1.3. Say Howdy**



If the visitor selects "Log on", the system prompts for credentials, as shown in the figure *Who-Who Are You?*.

**Figure 1.4. Who-Who Are You?**



Upon a successful login, the system presents a profile page for the subscriber, including the account setup and the list of email accounts, as shown in the figure *Known Subject*.

**Figure 1.5. Known Subject**

If a visitor did not have an account, then he or she could a new profile and start adding email accounts.

### 1.8.3. Does the MailReader use a Database?

The MailReader demonstrates data persistence, but it does not use a JDBC SQL database. The account and subscription information are persisted as a XML document. However, all the same design principles apply, regardless of how the data is stored.

The MailReader account and information data is represented by a set of data access objects. The objects take care of accessing the database "behind the scenes", so that the web application can focus on the user interface. There are data access objects for the database, Users, and Subscriptions.



#### Tip

The appendix *MailReader Tour* walks through the implementation of the complete MailReader example.

### 1.8.4. What are use cases?

Over the course of the book, we build the MailReader example, use case by use case.

- Use cases are a stylized narrative that tell us how the system is suppose to behave.
- Use cases capture contracts between the various stakeholders of a system.
- Use cases describe how actors achieve goals.
- Use cases are requirements, but not all of the requirements



#### Tip

An example use case is shown in the *Training Guide* section of this chapter.

The table "MailReader Use Cases" lists the use cases that describe the MailReader application.

**Table 1.5. MailReader Use Cases**

Goal	Actor	Story or Brief
Read Mail	Visitor and Subscriber	Visitors can Browse or Register. Subscribers can Login to a Profile and Subscribe to various mail servers (external to the system), Retrieve the mail for subscriptions, and Compose email messages. Data is retained between sessions.

Our Use Case format follows the style described by *Writing Effective Use Cases* by Alistair Cockburn [ISBN 0201702258].

## 1.9. Is the source code available?

Example source code can be downloaded and browsed at the Google Code site [<http://code.google.com/p/sq1-struts2/>].

## 1.10. Review: Apache Struts 2 From Square One

At the beginning of this session, we set out to answer several key questions and lay the foundation for the rest of the book. Let's review some takeaways.

- Struts is an elegant, extensible framework that makes it easier to build enterprise-ready Java web applications.
- This volume is the part of a four-part book project.
- The book is designed as a companion guide to a training course, but it designed to be useful on its own.
- Each session includes a primer (written chapter) and coding exercise. The exercises are based on use cases and include an implementation guide. During the course, we build the MailReader example application, step by step.
- All course materials are available from our Google Code site [<http://code.google.com/p/sq1-struts2/>].

## 1.10.1. Segue

*Struts 2 from Square One* is designed for people who want to create Java web applications quickly and correctly. In the session *Building Web Applications*, we cover any knowledge gaps by reviewing the framework's prerequisite technologies. (If frameworks help us create better applications faster, it is by standing on the shoulders of giants.)

# 1.11. Training Guide

The "Square One" session includes a brief overview of the course.

## 1.11.1. Presentation - Struts 2 from Square One

We introduce the workshop outline and related materials.

- How does Struts make web development easier?
- What are the objectives this course?
- How are the sessions organized?

### 1.11.1.1. About the Presentations

Each session begins with a general presentation that covers the exercise prerequisites. The training course includes a slide presentation. The book includes a written chapter. Both the slides and the chapters are considered to be "technology primers" in different formats.

### 1.11.1.2. About the Exercises

Each exercise implements a use case from a working example application, the "MailReader". Each exercise builds on the prior workshop, so that we have a complete, functioning application at the end. The initial workshops cover the simplest use cases. Later workshops cover more complicated topics.

The exercises are supported by training guide, which includes a formal use case.

#### 1.11.1.2.1. About the Training Guide

The training guide outlines coding the exercise, cite prerequisites, and even provide "hint code" as appropriate.

#### 1.11.1.2.2. About the Use Cases

The use case describes the exercise from the client's perspective. Each exercise represents a feature in the example application.

The *Read Mail* use case attached to this section covers in broad strokes the entire MailReader application.

## 1.11.2. Accomplishments

- Introduced the Struts 2 framework (from 30,000 feet)
- Introduced the course and training materials

## 1.11.3. Use Case: Read Mail

Visitors can login or register with the system, maintain a set of subscriptions to various mail servers (external to the system), retrieve the mail for subscriptions, and compose email messages.

*Narrative:*

Trillian is traveling without a laptop and would like to check her mail. Using the hotel's business center, she goes to the MailReader website, browses a few pages, and then he creates an account. Trillian enters the details for her email subscription, marking it for Auto Check. Trillian opens the check mail page and selects the Check Mail command. She reviews the mail for her account, replies to one of the emails, and composes a new email. The system retains her registration for a period of time, so Trillian can login again later, or the next time she is away.

*Elements:*

Goal:	Read Mail (User Goal)
Trigger:	Someone has mail they would like to read
Primary Actor:	Visitor or Subscriber

*Main Success Scenario (MSS):*

Step	Action
1	Visitor opens Welcome page
2	Visitor selects the login command
3	Subscriber selects the check mail command
4	Subscriber browses mail
5	Subscriber composes messages and replies

---

# Chapter 2. Building Web Applications

In this session, we step through the process of building plain-old web applications (without the Struts 2 framework). Experienced web developers might want to skip ahead to *Building Struts 2 applications*. Or, just settle back and enjoy the review (you might even find it interesting!). If you are new to web development, no worries, mate. We cover all the essentials, stem to stern.



## Note

In the companion lab exercise, we will be setting up the infrastructure for a Java web application. In the next session, we build on this infrastructure to create a simple Struts 2 application.

In exploring the process of building web applications, we will address two key questions:

- Why build web applications?
- How are Java web applications organized?

In the session *Building Struts 2 Applications*, we look at what the framework adds to the baseline Java platform.

Each key question is followed by a review section that reinforces take- away points.

## 2.1. Why build web applications?

Most often, we put applications on the web so that it's easier for people to find and use our applications. Nowadays, computers come with web browsers already installed. If someone can access a web site, they can access our web applications too. No fancy setup programs for people to run, or desktops for support departments to maintain. Anyone with a browser can just open up the web address and start using our latest hypertext creation.

### 2.1.1. Browsers? sites? applications? hypertext? Can we start from the beginning?

Hypertext is a modern word, but an ancient concept. Since time immemorial, we've created maps to help us visualize the world around us. Sailors use maps to navigate treacherous waters. Generals use maps to position troops and study supply lines. In the 1940s, people began to envision electronic maps with embedded cross references. If we wanted more detail about a tank or a plane, we could select the cross-reference, and instantly view details of the weaponry, fuel range, and crew profiles. From dreams like these, evolved the

notion of "hypertext", a system where we can select a reference in one document to access another.

In the early years, the Internet was a data wilderness, and finding information was a challenge best left to native guides. In 1991, a group of programmers and physics working at the European Laboratory for Particle Physics (CERN) in Switzerland, lead by Tim Berners-Lee, created an hypertext system with the intended use of indexing the Internet. The system became known as the "World Wide Web", and the web has grown beyond anyone's imaginings.

The software that people use to read hypertext documents on the web is called a "browser". Most often, the content is formatted as HTML, or "Hypertext Markup Language". HTML mixes special "markup" characters in with the regular content, to make it easy for authors to format text and include images. For example, a HTML tag like `` will insert a picture file into the text at the point where the tag appears.

Browsers interpret the HTML and display the text using various fonts and other special effects. The content is organized into a set of pages, which authors link together into a web site. In fact, the first web site [<http://www.husted.com/sq1-struts2/>] ever posted is still up and running, and now memorized as our figure *The First Homepage*.

**Figure 2.1. The First Homepage**



Of course, today, the original WWW site is an amusing historical curiosity. To find "what's out there" now, people turn to "search engines" like Yahoo! and Google. Our search queries are answered by huge database systems, perpetually updated.

Marshall McLuhan teaches that "the medium is the message". The web is a dynamic medium, and people have come to expect that web content be current. However, ordinary web sites are not designed to capture live data. We have to write special software that can obtain the latest data and create the web content "on the fly". When we extend a web site with dynamic content, we cross over to a web application.

We might want to deploy an intranet application serving the company's internal network, or it might be a high-volume Internet application available to the general public. But, it's something that we want on the web. Now.

## 2.1.2. How is an intranet different than the Internet?

The Internet is often called a "network of networks", which begs the question "what is a computer network?".

Ironically, people were the first computers. During World War II, legions of women armed with calculating machines computed tables of artillery trajectories. Down the hall, a team of scientists and engineers worked feverishly on an electronic computer that could automate such calculations, and, maybe one day, even predict the weather. The "Electronic Numerical Integrator and Computer" was not finished in time to help the Allies, and the weather is still hit-and-miss, but since Eniac's debut in 1946, computers have taken the world by storm.

Soon after we invented computers, we invented computer networks so that the machines could exchange information directly. (Otherwise, we would still be running punchcards from machine to machine!) Just like a telephone network, we can create a single connection to each machine and use a switching mechanism to enable each machine to contact any other machine "on the line".

In the late 1950s, we began linking the networks together and eventually created the "Internet" -- a network of networks. The original use case for an internet was to allow workers to share expensive special-purpose computers via remote connections. Today, the popular use case is electronic chat and mail. Not only can we send individual messages to each other, we can join topical mailing lists that route messages to any number of subscribers.

Today, the Internet is a publicly accessible network and open to all comers. Organizations world-wide find the Internet to be an excellent way to share information, mainly via electronic mail and the World Wide Web.

But, sometimes, the Internet is it a bit too excellent. Sometimes, we want to share information within our organization, but not necessarily with the whole world. An "intranet" is a closed system that is designed to "look and feel" like the Internet. Many companies create intranets to provide internal services to employees. In order for us to use an intranet, our computer must first be connected to the local network. Most web applications written today are intranet applications designed for internal use only.

Of course, intranet applications are not any easier to write. Colleagues can be our toughest critics. Building any useful web application is a complex task. We need to create both a web site and the equivalent of a desktop application, and merge the two into a seamless whole.

## 2.1.3. Are web applications so different?

Simple web sites are easy to create, which is one reason the web is so popular. When it comes to presenting static content to anyone, anyplace, the web platform reigns supreme.

A conventional web site serves up stored pages on demand, and it can do so at lightening speed. It's the 21st century answer to a basement printing press. Samizdat gone wild.

But, for some purposes, a conventional web site can be too static or too convenient. We may need to create content "on the fly" by incorporating information from a database system. We may want to restrict access to all or part of the site, either to preserve confidentiality -- or just to make a buck!

A web application uses a web site as the front-end to another layer of software, so that we that can do things like keep track of who is using the application, access a database, customize pages and graphics on the fly, and handle errors gracefully. A web application is a cybernetic troubadour, printing press, and personal valet, all rolled into one.

Since many of us need to augment web sites with dynamic media and custom logic, there is a standard "Common Gateway Interface" (CGI) that connects web sites with web-savvy computer software. A web application uses CGI, or the equivalent, to create a dynamic web site with custom content. To the web browser, the front-end appears to be an ordinary set of pages. But, appearances can be deceiving.

## 2.1.4. How does the Common Gateway Interface (CGI) work?

Conventional computer programs are loaded on demand, perform their appointed task, and vanish, until called again. Other programs run continuously as a "service" and "listen" for requests. When a request arrives, it is handled by the service program, and the "server" computer continues to listen for the next request.

A conventional web request refers to a file, or "page", stored somewhere in the server's file system. The browser doesn't need to know exactly where, because the server maps the web request to an internal location. We might request "index.html", and the server might send out the file stored at "C:\InetPub\default\index.html".

Of course, it wasn't long before we wanted web servers to do more than regurgitate static pages. We wanted to create dynamic pages, often based on database lookups. Instead of loading a page, we wanted to load a special computer program, and let the program send back the page instead.

In 1993, several workers collaborating via an Internet mailing list created the "Common Gateway Interface" (CGI) specification, which defined how a web server could invoke another computer program. To do this, we map some locations to the CGI program instead of ordinary files. When the server sees a request for "index.cgi", instead of retrieving the file, it runs the "index.cgi" as a program. The server passes a standard set of environment settings to the CGI program, which it can use to create an appropriate response.

All of the CGI-related work happens on the server side. The browser making the request doesn't know whether the page is a conventional static file or a dynamic response created by the CGI program. Either way, the browser receives back ordinary HTML.

Some platforms, including Java, offer more efficient alternatives to conventional CGI, but the principle remains the same. When a request is made for a Java resource, it is handled like a standard service, rather than a custom order, so that the response can be returned as quickly as possible.

## 2.1.5. How do we build web sites?

Originally, many web applications were built by using conventional programs to generate HTML with ordinary text-output statements. The example *Creating dynamic content via code* shows an "old-school" web application.

### Example 2.1. Creating dynamic content via code

(TODO)

That worked just fine, until someone wanted to change the look of page. Of course, when it comes to web sites, change is the only constant. Maintaining HTML in programming statements was not the way to go.

#### **A Short History of Java and the Web**

Java and the web grew up together. Just as Tim Berners-Lee was announcing the World Wide Web project, another developer, James Gosling, was creating a new programming language that would come to be called "Java". There are many different types of computers, and, generally, a program built on one kind of computer, cannot be run on another kind of computer. A key feature of Java is "Write Once, Run Anywhere", where a Java program built on one kind of computer could be run on any other kind of computer that supported Java.

Java created a lot of excitement for itself, and the web, by creating the "HotJava" web browser. Aside from rendering HTML pages, HotJava could also run Java "applets", which offered capabilities above and beyond ordinary web surfing. Other browsers soon added Java applet support. As browsing became more interesting, Java became more popular.

In 1997, Java introduced a web server along with a new technology called "servlets". Applets had been designed to be run inside of another application (the browser). Servlets were designed to be run inside of a web server, or, more specifically, a servlet container. Servlets fill the same niche as CGI programs, but servlets are faster and more powerful.

Java Server Pages arrived as an extension to servlets in November 1998. Developers can create a JSP template that looks much like a HTML page, but the template can use Java programming statements to do things like access database systems. The servlet container compiles the JSP template into a servlet, for speed and efficiency.

So, we turned the problem inside-out and created "server pages". Instead of putting HTML into the programming statements, we put programming statements into the HTML.

The example *Creating dynamic content via server pages* demonstrates the server page approach.

### Example 2.2. Creating dynamic content via server pages

( TODO )

When a request for a server page arrives, the programming statements are processed before the text is returned. In the *server pages* example, the programming statement is replaced with its output, the current time. The browser receives the page as ordinary HTML, which it renders in the ordinary way. The example *server page as rendered by a browser* shows the final result.

### Example 2.3. Creating dynamic content via server pages

( TODO )

Another way to create dynamic pages is to use "client-side" programming (as opposed to "server-side" programming). Today, most browsers understand a programming language called JavaScript. We can mix JavaScript statements with HTML to create special effects.

### Example 2.4. Dynamic content via JavaScript

( TODO )

The example *Dynamic content via JavaScript* shows the same page redesigned to use JavaScript.



#### Tip

JavaScript is (now) an extension of a standard scripting language called ECMAScript. Other ECMAScript dialects include JScript (Microsoft) and ActionScript (Adobe Flash). JavaScript was invented first, and the ECMAScript standard was created so different implementations could share a common feature set. (The Java and JavaScript programming languages are not directly related, except that they have similar names.)

JavaScript was originally named LiveScript. It was created by Brendan Eich and included in version 2.0 of the Netscape web browser, released in 1995.

Many web applications use both server-side programming and client-side scripting. For example, a client-side script might ensure that we enter into a form both a username and a password. Once the form passes validation, a server-side program can check the database for a matching account.



#### Tip

Business logic is the part of the application that actually solves the problem at hand. In a email program, an example of business logic is filtering messages

into various folders. In a spreadsheet, an example of business logic is summing a column. The user interface presents data. The business logic determines what data to present. The business logic will often work closely with data access logic, to query or update a database. The user interface will often work with a facade, which hides both the business logic and data access logic behind a single consistent interface.

A key difference between a web site and a web application is that an application can validate input, execute business logic, and create dynamic content on the server. We don't have to do it all on the web page. A web application can create dynamic content in several ways: via code, via server pages, via client-side scripting, and all three in any combination.

## **2.1.6. How much HTML do we need to know to create web applications?**

To get started, Java developers need to know only the basics of HTML. Often, a team building a web application will have people who focus on the Java programming, and other people who focus on the HTML mechanics. Other times, the Java web developer has to be a "jack of all trades".

### What is HTML, anyway?

HTML, short for HyperText Markup Language, is the predominant markup language for the creation of web pages. It provides a means to describe the structure of text-based information in a document "by denoting certain text as headings, paragraphs, lists, and so on" and to supplement that text with interactive forms, embedded images, and other objects. HTML is written in the form of labels (known as tags), created by greater-than signs (>) and less-than signs (<). HTML can also describe, to some degree, the appearance and semantics of a document, and can include embedded scripting language code which can affect the behavior of web browsers and other HTML processors.

— Wikipedia, the free encyclopedia  
[<http://en.wikipedia.org/wiki/HTML>]

**Figure 2.2. HTML (HyperText Markup Language)**



We can use HTML to gather and display information almost as well as a desktop application. But, turning a conglomeration of web pages and images into a coherent web application is still a challenge.

Today, every popular language has extensions designed to ease web development. One of the leading platforms for web application development is Java. By following a set of design conventions, the Java platform makes it easy to combine JavaServer Pages and compiled Java classes into a web application. Java classes can handle server-side logic, while JavaServer Pages provide the user interface.

## 2.1.7. Review: Why build web applications?

Let's pause for a moment, and review a key take-away point:

- In a web application, client data input executes \_\_\_\_\_ on the server.

### 2.1.7.1. Answer: Why build web applications?

- In a web application, client data input executes *business logic* on the server.

## 2.1.8. How are Java web applications organized?

Java and the Internet grew up together. The HotJava browser and Java applets helped popularize the web by extending the capabilities of early browsers beyond what we could provide with HTML. Soon, we were not only using Java to surf the web, we were using Java to create web applications.

Early CGI programs gave web sites access to databases and other software components, but CGI can be slow and limiting. So, Java engineers created a "container" that we could add to a web server. The web server still handles the standard web requests, but other requests can be processed by the Java components, in the same way that some requests were handled by CGI.

Sometimes a container will be used as an extension to a conventional web server. The conventional web server handles the static pages and images but forwards to the container requests for Java resources (including JavaServer Pages). Today, most containers also include a basic web server, making the product an all-in-one solution.

Essentially, containers are service providers, and applications are service consumers. Applications request services via a standard configuration file and an infrastructure directory with other special folders and files. When the container starts, it reads the configuration file and loads the application.

The configuration file tells the container what bootstrap classes to load on startup. The standard folders provide a place to add custom classes, specific to an application. The figure *Show me the Classes* is a typical file layout under the Apache Tomcat container.

**Figure 2.3. Show me the Classes**



The root infrastructure folder is `WEB-INF`, which is directly off the website's document root. Everything under `WEB-INF` is "protected" by the container so that these resources cannot be directly accessed by a browser. Under `WEB-INF`, we find three standard components: `web.xml`, the `lib` folder, and the `classes` folder.

The `web.xml` file is also known as the *Web Deployment Descriptor*. When the container starts up, it reads the `web.xml` to find out what servlets and other components to load for an application. If these "bootstrap" components cannot be loaded, then the container will usually refuse to load the application. In this way, the application can rely its own bootstrap components being available.



### Tip

A useful feature of Java web containers is that each application runs in its own "space". If one application fails to load, any others using the container can still load normally. If an application doesn't load, most containers maintain log files that capture error and status messages.

The `classes` folder holds the application's Java classes. The classes start out as human-readable Java source code. To help the application run faster, we compile the source code into a binary format. Every resource under the `classes` folder is put on the container's search path, or *classpath*". When the application is looking for classes or other resources, the `classes` folder is checked first. Accordingly, we might also find configuration files and message files under `classes` that the applications wants available on the classpath.

The `lib` folder hosts collections of other classes and resources needed by an application. Each collection is contained in its own Java Archive, or JAR, file. The JAR uses a special file format so that it can contain any number of other files and folders. JARs can be used to share resources between applications. The Apache Struts 2 framework, for example, is distributed as a JAR.



### Tip

The JAR format is actually the same format used by the popular "zip" software. Any program that can access ZIP files can be used to peek inside a JAR.

If an application uses JavaServer Pages, it may also use tag libraries. JSP tags look like HTML tags but act like program. Most JSP tags will take input from the page or container and replace the tag with dynamic output. The syntax used by JSP tag is defined by a Tag Library Descriptor (TLD) file. A TLD is usually packaged in a JAR, but it may also be found under `WEB-INF` or `WEB-INF/tags/`.

The Struts 2 framework bundles a comprehensive tag library, designed to make web development faster, easier, ... and snazzier! In the session *Building Struts 2 Applications*, we will look at the components that Struts 2 brings to Java web applications, including the tag library.

## 2.1.9. Review: How are applications organized?

Match each numbered item with its definition.

1. CGI
2. WEB-INF
3. web.xml
4. JAR
5. TLD

Definitions to match:

- a. Tag Library Descriptor
- b. Java Archive
- c. Web Application Infrastructure
- d. Computer Gateway Interface
- e. Web Application Deployment Descriptor

### 2.1.9.1. Answer: How are applications organized?

1. CGI = *(d) Computer Gateway Interface*
2. WEB-INF = *(c) Web Application Infrastructure*
3. web.xml = *(e) Web Application Deployment Descriptor*
4. JAR = *(b) Java Archive*
5. TLD = *(a) Tag Library Descriptor*

## 2.2. Review: Building Web Applications

At the beginning of this session, we set out to answer two key questions:

- Why build web applications?
- How are Java web applications organized?

Along the way, we learned that web applications are

- Easier to find, use, and deploy

- Browsers pre-installed
- No desktop to maintain
- Suitable for intranet or Internet

We also learned that building web applications is difficult because

- Standard HTML controls are plain vanilla
- HTML/HTTP is a stateless protocol
- The database, server, and browser are all separate products

One way to make web applications easier to build is to use the Java web platform. Java web applications run with a "container" and are organized in a particular way. All of the required Java resources can be found under a WEB-INF folder, which contains a classes folder, a lib folder, and a web.xml configuration file.

- WEB-INF
  - web.xml
    - classes
    - lib

## 2.2.1. Segue

Apache Struts 2 is an elegant, extensible framework that builds on the Java web platform. In the session *Building Struts 2 Applications*, we build on the foundation material covered in this session, and explore the Struts 2 architecture.

## 2.3. Training Guide

We start from square one and discuss why we build web applications and why building web applications is such a chore.

### 2.3.1. Presentation - Building Web Applications

Today, the web is everywhere, and bundled with everything. Where did it all start, and how do we jump on the bandwagon?

- Why build Web applications?
- Why is building applications such a chore?
- How do we build web sites?
- How are Java web applications organized?

### 2.3.2. Coding Exercise - "Hello"

Story: The application will be deployed for the web on the Java platform.

Setup essential software tools, so we can deploy a simple "Hello" application that we extend in the main workshop.

- Install Development Software
- Setup Environment
- Test Environment

### **2.3.3. Prerequisites**

- CD source code or high-bandwidth Internet connection

### **2.3.4. Exercises**

- Install Development Software
  - Java
  - Maven
  - Jetty
  - Integrated Development Environment (IDE)
    - Eclipse, IDEA, or NetBeans
  - MailReader-QuickStart.zip
- Setup Environment
  - Build Mailreader-Hello with Maven
  - Create a "MailReader" project with Maven plugin
- Test Environment
  - Create a JUnit task and run the unit test
  - Create a Tomcat task and run the web application
  - Confirm that page says "Congratulations. Struts is up and running."

### **2.3.5. Accomplishments**

- Installed Java, Maven, Jetty, IDE
- Setup starter web application for IDE
- Ran tests and sample application to verify infrastructure

## 2.3.6. Use Case: Hello (Setup development tools)

Building Java web applications requires several tools to be setup and configured to work together.

*Narrative:*

Ford is ready to get started on the MailReader project, but he needs to setup a development environment first. To build the MailReader, Ford installs the Java platform, a Java container, and an integrated development environment (IDE). To test the environment, Ford also installs a "quick start" Struts 2 application, runs a unit test, and deploys the "Hello" web application. All systems are green, so Ford moves onto the client's first use case.



*Elements:*

Goal:	Install development platform (Developer Goal)
Trigger:	Development on a new workstation
Primary Actor:	Developer

*Main Success Scenario (MSS):*

<b>Step</b>	<b>Action</b>
1	Download and install latest Java 5 from <a href="http://java.sun.com">java.sun.com</a>
2	Download and install latest Tomcat 5 ZIP distribution from <a href="http://tomcat.apache.org">tomcat.apache.org</a> . Extract to /usr/local/tomcat so that startup.bat is at /usr/local/tomcat/bin/startup.bat.
3	Download and install latest IDEA 6 from <a href="http://www.jetbrains.com">www.jetbrains.com</a> .
4	

*Preconditions and Guarantees:*

Preconditions:	Hardware configuration supports IDE requirements
----------------	--

---

# Chapter 3. Building Struts 2 Applications

In this session, we look at building web applications with the Apache Struts 2 framework. If you are new to web development, be sure to review the *Building Web Applications* session first, which covers the foundation material.



## Note

In the companion lab exercise, we will create a simple Struts 2 application.

In exploring the framework, we will address three key questions:

- Can we use a conventional design?
- Why use frameworks?
- What value does Struts 2 add?

Each key question is followed by a review section that reinforces take-away points.

## 3.1. Can we use a conventional design?

Once upon a time, in 1978, Trygve Reenskaug, a visiting scientist to Xerox PARC, was working on a SmallTalk application that helped people design and build ships. The users of the application needed to inspect and manipulate a large and complex data set. Reenskaug struck upon the idea of separating the application into three areas of concern: the Model, View, and Controller (MVC).

**Figure 3.1. Modeling the Problem Domain**



"The essential purpose of MVC," says Reenskaug, "is to bridge the gap between the human user's mental model and the digital model that exists in the computer. The ideal MVC

solution supports the user illusion of seeing and manipulating the domain information directly." (<http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>)

The figure *Bridging the Model Gap* illustrates Reenskaug's original conception.<sup>1</sup>

### Figure 3.2. Bridging the Model Gap



The notion of MVC has evolved over the years, but the essential separation of concerns lives on, even into web applications. In a desktop application, MVC is often depicted as a triangle or loop, as shown in the figure *Conventional MVC*. Here, the View talks to the Controller, and the Controller consults with Model and selects the next View. To close the loop, the View updates itself from the Model.

### Figure 3.3. Conventional MVC



In concrete terms, the View might be a dialog box. The Controller accepts input from the form and updates the Model (a database). The Views (plural) observe the change to the Model and update their individual representations of the Model. A key notion behind MVC is that there might be multiple views, or representations, of the same data. The figure *Multiple V's in MVC* illustrates.

<sup>1</sup> Image used with permission.

**Figure 3.4. Multiple V's in MVC**



In the original desktop implementation, Views are expected to "observe" the Model. When the Model changes, it alerts the Views, and the Views refresh themselves. The figure *Desktop MVC* outlines the original roles and workflow.

**Figure 3.5. Desktop MVC**



Over the years, MVC became a popular paradigm for designing interactive software applications. But, the idea of "observing" state changes to the Model doesn't work well for web applications. The web is designed to be a disconnected environment, and the Observer pattern needs a live connection to work well.

So, for the web, we modify the pattern. Web MVC looks less like a loop and more like a horseshoe or bridge. The figure *Enterprise MVC* outlines the modified workflow.

**Figure 3.6. Enterprise MVC**

In Desktop MVC, the View pulls state changes directly from the Model. In Enterprise MVC, the Controller pushes state changes to the View - so that the View does not need a direct connection to the Model. The Controller becomes a go-between for the View and Model. The presentation layer exchanges input with the control layer, the control layer exchanges input with the application logic, the application logic queries or updates the database.

With this small change, we find that MVC works well with web applications too. In practice, the larger an application, the more an application benefits from the separation of concerns that MVC provides.

### 3.1.1. Why bother?

Sure, MVC looks cool on paper, but don't we end up with more pieces to puzzle? If we can put Java code in server pages, why not put everything into the page, and be done with it?

Good question. (Been there, did that.) For small applications with five or ten pages, a "page-centric" approach works just fine. But, as applications grow, more and more pages are able to share more and more code. Meanwhile, the links between the pages become more complex. Changing the page flow can mean touching several pages. Since (in this design) each page mixes code with markup, each page is fragile and easy to break.

The example *Welcome to the Jungle* shows a server page that mixes code and markup. In Java vernacular, we call this design "Model 1".

#### **Example 3.1. Welcome to the Jungle (Model 1)**

(TODO)

By contrast, the example *separating code and markup* moves the code into a separate object. In Java vernacular, we call this design "Model 2".

#### **Example 3.2. Separating code and markup (Model 2)**

(TODO)

Again, for a small application, Model 1 works just fine. But as applications grow, Model 2 becomes more and more attractive.

When we build web applications, we can choose our own internal architecture. We can use Model 1 in one application, and Model 2 in the next. But, under the hood, all Java web applications still share a common infrastructure.

## 3.1.2. Review: Can we use a conventional design?

Match each numbered item with its definition.

- Model
- View
- Controller

*Definitions to match:*

- a. Renders Model
- b. Selects View
- c. Retains State

### 3.1.2.1. Answer: Can we use a conventional design?

- Model = (c) *Retains State*
- View = (a) *Renders Model*
- Controller = (b) *Selects View*

## 3.2. Why use frameworks?

OK, but do we need to use a "framework" to implement a MVC design? Is it that complicated?

While it is not hard to build an application using a MVC architecture, it is harder to abstract common features of an application into a framework. There are several advantages to using an application framework.

- Code reuse
- Incremental development
- Long-term maintenance by a team
- Separation of Concerns
- Don't Repeat Yourself

*Code reuse.* Most of us don't create just one application. Each application we build is different, but most share common characteristics. Rather than reimplement each reusable

feature from scratch, we can put all the reusable parts together into a "semi- complete application" to use on our next project. Or, in other words, a framework.<sup>2</sup>

*Incremental development.* Many applications are created in fits and starts. A framework can help guide us down a common development path. Good frameworks help developers explore an application, making it easier to find where we left off. By definition, frameworks are built for extensibility, making it easier to write one stage of the application, pause, and then go on to the next.

*Long-term maintenance by a team.* A complex, MVC application is often developed by a team. A framework defines a common, well-understood structure that everyone on the team can follow. Extensible frameworks also let one member of a team plugin a custom component for other team members to use. Team members can come and go, but still find an application easy to maintain, if it uses a familiar framework.

*Separation of Concerns.* By definition, a MVC application separates the concerns of presentation, navigation, and business logic. A good MVC framework helps us keep these concerns separated, so that we can get the full benefit of the MVC approach.

### **Separation of Concerns**

The term "Separation of Concerns" is credited to Edsger W. Dijkstra from his paper "On the role of scientific thought".

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained - on the contrary! - by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.<sup>3</sup>

In this author's opinion, "Separation of Concerns" is the one true best practice, and all other best practices are specific instances of SoC.

---

<sup>2</sup> <http://st-www.cs.uiuc.edu/users/johnson/frameworks.html>

<sup>3</sup> <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>

*Don't Repeat Yourself.* A good framework helps us normalize an application in the same way we normalize a database, and for the same reason. Ideally, we should be able to express a fact once so that any component that needs that fact can fetch the canonical reference. In a database, facts are items like "Last Name = Beeblebrox". In an application, facts are also items like "Date Format = d MMM yyyy". Without a framework, we often end up sharing code through cut and paste. Meaning that if the code changes later, we have to update a fact in multiple places, risking an "update anomaly".

As before, the larger the application, the more valuable these benefits become. In a small application, there is less code to reuse and fewer concerns to separate. But, as an application grows, so does the value of a MVC framework.

## 3.2.1. Review: Why use frameworks?

Let's fill in the blanks to review some key take-away points.

- Code \_\_\_\_\_
- \_\_\_\_\_ development
- Long-term \_\_\_\_\_ by a team
- \_\_\_\_\_ of Concerns
- Don't \_\_\_\_\_ Yourself

### 3.2.1.1. Answer: Why use frameworks?

Reasons that frameworks are useful include:

- Code *reuse*
- *Incremental* development
- Long-term *maintenance* by a team
- *Separation* of Concerns
- Don't *Repeat* Yourself

## 3.3. What value does Struts add?

So, if the Java web platform is so popular, why do we need to add anything at all? Why can't we just use Java out of the box?

While Java provides several essential services to web applications, there are a number of "missing links" between the services. When Java is found wanting, open source frameworks like Struts rush in to provide the gluecode we need to put it all together.

### Why do we need to add anything?

Java provides several essential services

- Servlets
- Server Pages
- Session Tracking
- Localization

that most enterprise applications need. While helpful, all of these services come up short at implementation time.

*Servlets.* Filters and Servlets make it easy for a Java application to handle the request in an object-orientated way. Servlets do their job well, but in practice they tend to be heavyweight, hard to configure, and web bound.

*Server Pages.* JSP templates are a very powerful tool. Anything we can do with a servlet, we can do with a JavaServer Page ... mainly because JSPs are a special kind of servlet. But, with great power comes great responsibility. If used exclusively, JSPs tend to encourage developers to mix Java code with HTML markup. In practice, we've found that mixing Java and HTML creates fragile applications that are easy to break and hard to maintain.

*Session tracking.* The Java web platform has built-in session tracking, but it's not foolproof. To ensure that it works correctly, we sometimes need to include special codes in hyperlinks, to retain the session identifier between requests. Encoding links is a headache ... unless a frameworks like Struts 2 automatically encodes links as needed.

*Localization.* The Java platform includes a localization library to help us internationalize our applications. But localization support is not built into servlets. The JavaServer Pages Standard Tag Library (JSTL) provides a message tag support but much is still left to the developer.

The framework adds value in two ways:

- Workflow Components
- Elegant Architecture

First, the framework provides key workflow components that help developers solve everyday use cases. Second, the framework provides an elegant architecture so that we can easily extend and customize our web applications.

## 3.3.1. What are the key workflow components?

The Apache Struts framework exposes three key components

- An action handler
- A result handler
- A custom tag library

Both Struts 1 and Struts 2 rely on these components. Here, we will only discuss only the Struts 2 versions.

## 3.3.2. How do we handle actions?

The action handler processes input and interacts with other layers of the application.

An application can map an action handler to logical resource names, like `Welcome.action` or `Account_insert.action`. When a browser submits a request for one of these resources, the framework selects an Action class to handle the exchange.

The input values from the request (if any) are transferred to properties on an Action class (or some other class of your choosing). The framework then calls a method on the Action to invoke the business logic associated with the request.

The class can return a string token to indicate the outcome of the transaction. Usually, these tokens are generic terms like "success", "failure", "input", "error", or "cancel".

The example *struts.xml* shows a handler for a simple "Hello" action with a single "success" result. The "success" result type is the default, and an unnamed result is deemed to be "success".

### Example 3.3. struts.xml

```
<struts>
  <package name="default"
    extends="struts-default">

    <action name="Hello" class="Hello">
      <result>/pages/Hello.jsp</result>
    </action>

    <!-- Add your actions here -->
  </package>
</struts>
```

From a MVC perspective, the Action represents the Model, since it is responsible for managing application state.

### 3.3.2.1. Can an Action class interact with Hibernate?

The most common use of an Action class is to process business logic and access a database. Most often, a database is accessed through a data access framework, like Hibernate. The Hibernate site provides an example of a Struts2/WebWork Action.<sup>4</sup>

A complex application might want to create a business facade that would hide Hibernate from an Action class, but in a simple web application, it may make sense to define

---

<sup>4</sup> <http://www.hibernate.org/51.html>

a persistence-aware superclass so that Action classes can use Hibernate directly. A full-featured superclass will handle logging, exceptions, and Session instantiation.

**Example 3.4. HibernateAction.java**

```
public class HibernateAction extends ActionSupport {

    private static final Log LOG = LogFactory.getLog(HibernateAction.class);
    private Session session;

    protected String invokeCommand() throws Exception {
        LOG.trace("Opening Hibernate session");
        SessionFactory sf = (SessionFactory) ActionContext.getContext()
            .getApplication()
            .get("SessionFactory");
        if (sf==null) sf = buildSessionFactory();
        session = sf.openSession();
        session.setFlushMode( getFlushMode() );

        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            LOG.trace("Invoking Action command");
            super.invokeCommand();
            LOG.trace("Committing Hibernate Transaction");
            tx.commit();
        }
        catch (Exception e) {
            LOG.error("Exception invoking Action command " +
                "or committing Hibernate Transaction", e);
            if (tx!=null) tx.rollback();
            throw e;
        }
        finally {
            session.close();
        }
    }

    protected final Session getSession() {
        return session;
    }

    protected FlushMode getFlushMode() {
        return FlushMode.AUTO;
    }

    private SessionFactory buildSessionFactory() {
        // Create a SessionFactory and store it in Application context
    }
}
```

The same sort of base Action class shown in the *HibernateAction.java* example could be written for iBATIS or Cayenne, or your own data access framework.



## Note

Since a new Action class instance is created for each request (unlike Struts 1), we can subclass *HibernateAction* and add the properties needed by a data-entry form. The Action class can pass those properties directly to Hibernate, or retrieve properties from Hibernate to populate the form.

### 3.3.3. How do we handle results?

The result handler either creates output or dispatches to another resource, like a server page, to create the output.

There are several result handlers bundled with the framework and several more are available as plugins. For example, there are plugins for rendering reports with JFreeChart or JasperReports. Another plugin converts output to JSON for AJAX applications. Yet another lets Struts 2 applications utilize JavaServer Faces components as a result. Other custom results can be created and plugged into the framework as needed.

Each action handler can have one or more result handlers, mapped to whatever result types are needed. Regardless of what type of result is being generated, the action handler only needs to return a logical name. The action does not need to know how the response is being handled.

In the example *struts.xml*, the default `dispatch` result would be used. The framework transfers control the `/pages/Hello.jsp` resource, and lets the container render the JSP.

From a MVC perspective, the Result represents the Controller, since it selects the next view.

### 3.3.4. How do we display dynamic data?

The Struts Tags render dynamic content into HTML so that it can be presented by a standard browser.

The tag library includes a set of "UI" tags that reduce the amount of markup needed to create data entry forms. The UI tags utilize templates and style sheets to create a table around a form and automatically present any error or validation messages. Of course, the markup generation can be switched off to give us total control over the markup when needed.

There is also a set of advanced tags for creating novel controls, like an "option transfer" control, where selected items can be transferred from one list to another.

The Struts Tags work with JSP, Freemarker, and Velocity, so that we have our choice of presentation layers. The example *Hello.jsp* shows a JavaServer Page using the Struts Tags.

From a MVC perspective, the tags represent the view, since they display application state.

### 3.3.5. What happens during the workflow?

The figure *Struts 2 Workflow* depicts the components in action.

**Figure 3.7. Struts 2 Workflow**



A typical request workflow runs something like this:

- The web browser requests the page.
- The framework looks at the request and determines the appropriate Action.
- The framework automatically applies common functionality to the request like property population, validation, authorization, and file uploading.
- The Action method executes, usually storing and/or retrieving information from a database.
- The Result either renders the output (images, PDF, JSON) or dispatches to another resource, like a JavaServer Page, to complete the response.
- The server page utilizes the Struts Tags to output dynamic data.

#### 3.3.5.1. What resources does the workflow utilize?

Let's step through the resources we would need to create a simple Hello World application, that could be localized for different languages.

The example *struts.xml* showed the configuration for our Hello action. The configuration file is loaded from the classpath. (An easy way for an application to place a resource on the classpath is to store it under `WEB-INF/classes`.)

The example *Hello.java* is an Action class that just forwards to the default result type "SUCCESS".

### Example 3.5. Hello.java

```
import com.opensymphony.xwork2.ActionSupport;
public class Hello extends ActionSupport {
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

Truth be told, we don't even need a Java class for this example. For the action mapping, we could have written

```
<action name="Hello">
    <result>/pages/Hello.jsp</result>
</action>
```

since the default execute method on the default Action class returns "success" too.

The example *Hello.jsp* uses the "text" tag to display an localized message.

### Example 3.6. Hello.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
    <title>Hello</title>
</head>
<body>
<h2><s:text name="hello.message"/></h2>
</body>
</html>
```



#### Tip

Even if an application is not being localized, it's a good practice to store key phrases in a message resource file, so that they can be reused. The example *resources.properties* shows our message file, which is also loaded from the classpath.

### Example 3.7. resources.properties

```
hello.message = Congratulations! Struts is up and running ...
# Add your messages here ...
```

If messages for another languages were needed, we could add a resource file for that language, and the framework would use it automatically. The default language is taken from the client's browser and can also be set progmatically.

The figure *Hello World!* shows the page displayed by our "Hello.action" workflow.

**Figure 3.8. Hello World!**

The example *Welcome.jsp* shows a slightly more complicated page that includes links to other actions.

**Example 3.8. Welcome.jsp**

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<body>
<h3>Options</h3>
<ul>
<li><a href="<s:url action="Register"/>">
  Register</a></li>
<li><a href="<s:url action="Logon"/>">
  Sign On</a></li>
</ul>
</body>
</html>
```

On a day-to-day basis, these are the four resources that most Struts developers might touch every day.

- the action mappings,
- the Action class,
- the server page, and
- the message resources.

Happily, even with Struts 2 in the picture, the simple things are still simple. We can add the properties we need to an Action class, the Action can use those properties to interact with the business layer, and the framework can output the result. *Click. Done.*

Of course, in real life, users demand more than simple pages with simple workflows. To help us deal with complex workflows, Struts 2 is backed by an elegant, extensible architecture, that is powerful, fast, and flexible.

### 3.3.6. Review: What are the key workflow components?

- A\_\_\_\_\_ h\_\_\_\_\_
- R\_\_\_\_\_ h\_\_\_\_\_
- C\_\_\_\_\_ t\_\_\_\_\_
- The Struts 2 default configuration file is named \_\_\_\_\_ .xml
- To reference the Struts 2 taglib, use the URI "/\_\_\_\_\_ - \_\_\_\_\_"

#### 3.3.6.1. Answer: What are the key workflow components?

- *Action handler*
- *Result handler*
- *Custom tags*
- The Struts 2 default configuration file is named *struts.xml*
- To reference the Struts 2 taglib, use the URI *"/struts-tags"*

### 3.3.7. What are the key architectural components?

The workflow components make coding web applications easier. But, to make it that easy a lot has to go on behind the scenes.

For example, we may need input values from the request to be mapped to properties on Java class, we may need to validate the input, we may want a client to be authenticated and authorized before executing the action, we may need to expose the Action properties to the Struts Tags, and we might even want to monitor the application's response time.

In a larger application, we might want to customize what happens to an incoming request. Some requests might need more handling, some requests might need less handling. For example, AJAX requests should be handled as quickly as possible.

To keep request processing fast, flexible, and powerful, the Struts 2 architecture utilizes three key components.

- Interceptors
- Value Stack
- Expression Language

Interceptors provide services like validation and authorization. The Value Stack exposes properties from the Action and other objects to the Expression Language. The Expression Language can be used to present properties found on the Action and other objects.

Let's look at each architectural component in turn.

### 3.3.8. What are Interceptors?

Services like property population, validation, and authorization are provided by components called "Interceptors", which can fire before and after the Action executes. Most of the framework's utility is provided by Interceptors. All requests can be handled by a single set of Interceptors, or different sets of Interceptors can handle different requests.

The set of Interceptors acts as a "gauntlet" for the request. The request object passes through each Interceptor in turn. The Interceptor can ignore the request, act on the request data, or short-circuit the request and keep the Action class method from firing.

**Figure 3.9. Interceptor Gauntlet: Thank you sir! May I have another?**



The figure "Interceptor Gauntlet" depicts the Interceptor component in action.

The simplest example may be the `TimerInterceptor`, which logs how long it takes to execute an action.

#### Example 3.9. `TimerInterceptor.java`

```
public String intercept(ActionInvocation invocation)
    throws Exception {
    if (log.isInfoEnabled()) {
        long startTime = System.currentTimeMillis();
        String result = invocation.invoke();
        long executionTime = System.currentTimeMillis() - startTime;
        String namespace = invocation.getProxy().getNamespace();
        // ... generate logger message string
        doLog(getLoggerToUse(), message.toString());
        return result;
    }
}
```

In the `TimerInterceptor.java` example, the code first obtains the current time, and then lets the rest of the Interceptor stack run by calling `String result =`

`invocation.invoke()`. When control returns to the `TimerInterceptor`, we compute and log the execution time. The result code is returned, and the process repeats with the next `Interceptor` (if any).

OK, that's great for timing throughput. But, what about tasks that might need to stop the request? Like, say, because validation fails?

To keep the `Action` method from executing, we can just return a result code without calling `invocation.invoke`. If we skip the `invoke`, the call begins to return. The "backside" of any `Interceptors` already consulted will be still be called. But any `Interceptors` positioned later in the stack will not fire, and neither will the `Action` class.

The `DefaultWorkflowInterceptor` uses this technique to return the "input" result code if validation fails. When an `Interceptor` "short-circuits" the stack, it can return whatever result code is appropriate. In the case of validation, the `Interceptor` returns "input". When control returns, the "input" result is selected, just as if the `Action` method had returned the code instead.

Another good `Interceptor` example is the `PrepareInterceptor`, which works hand-in-glove with the `Preparable` interface.

### Example 3.10. Preparable Interface and the PrepareInterceptor

```
public interface Preparable {
    void prepare() throws Exception;
}

public class PrepareInterceptor extends AbstractInterceptor {
    public String void intercept(ActionInvocation invocation)
        throws Exception {
        Object action = invocation.getAction();
        if (action instanceof Preparable) {
            ((Preparable) action).prepare();
        }
        return invocation.invoke();
    }
}
```

In the example *Preparable Interface and the PrepareInterceptor*, we see a common Struts 2 pattern. The `Preparable` interface defines a `prepare` method. If the `Action` underlying the request implements `Preparable`, the `Interceptor` invokes the `prepare` method. Otherwise, the request is ignored.

Note that the `PrepareInterceptor` does nothing to incoming requests. All the work happens after the `Action` method executes (if it executes).



### Note

In WebWork 2, there was `AroundInterceptor` that provided before and after methods, and called `invocation.invoke` in between. But, the Apache Struts

group decided it was more trouble than it was worth, and we removed the class in Struts 2.

### 3.3.8.1. How do we add custom Interceptors?

To create your own Interceptor, create a Java class that implements the interface shown in the listing *Interceptor.java*. (Note that it is an XWork interface.)

#### Example 3.11. Interceptor.java

```
package com.opensymphony.xwork2.interceptor;

import com.opensymphony.xwork2.ActionInvocation;
import java.io.Serializable;

public interface Interceptor extends Serializable {
    void destroy();
    void init();
    String intercept(ActionInvocation invocation) throws Exception;
}
```

Like Struts 1 Actions, Interceptors are instantiated once by the framework, and so must be thread-safe. (Conversely, Struts 2 Actions are instantiated for each request and do not need to be thread-safe.) When the Interceptor is first instantiated, the `init` method is called. Any resources the Interceptor needs can be allocated within the `init` method. Just remember that all requests passing through the Interceptor will share access to class resources.

## Thread Safety

Most of us can't leap into action the moment the alarm clock rings. We need to get up, get dressed, have coffee, drive to work, find our desk, and then, finally, we are ready to have at it. Computers work much the same way. When a request comes into a web server, there are a number of resources that need to be allocated to the request before it can be handled. Java web servers eliminate some of that overhead by sharing resources between multiple requests.

In computer science parlance, we put each request on a thread and execute several threads at once in a shared address space. Without multi-threading, the server would spend more time allocating resources, and less time handling the actual request.

Within the Struts framework, Interceptors are multithreaded. Two requests might be passing through the Interceptor at the same time (or very close to it), and, depending on the circumstances, request B might pass through the Interceptor before request A completes. If the Interceptor uses class-scope variables, one thread passing through the class could change the variable, and another thread could change it again, overwriting the value set by the "first" thread.

Each thread does get its own stack frame. In practice, to keep Interceptors thread safe, use only variables that are placed on the thread's stack frame. In practice, this means variables passed into the method and variables declared in the body of the method.

*Wrong:*

```
long startTime; // Class-scope! Wrong!
public String intercept(ActionInvocation invocation)
    throws Exception {
    if (log.isInfoEnabled()) {
        startTime = System.currentTimeMillis();
        // ...
    }
}
```

*Right:*

```
public String intercept(ActionInvocation invocation)
    throws Exception {
    if (log.isInfoEnabled()) {
        long startTime = System.currentTimeMillis(); // Right!
        // ...
    }
}
```

So long as Interceptors are thread-safe and return the result of `invocation.invoke()`, the sky's the limit!

The `intercept` method is invoked for each request passing through the Interceptor. As with the `Action.execute` method, the `intercept` method returns a result code. The framework uses the code to select an result type.

Conceptually, Interceptor methods have a "before" and "after" part, either of which is optional. The dividing line between "before" and "after" is a call to `invocation.invoke()`.



## Note

All Interceptors should include code that calls `invocation.invoke()` and returns the result code on a nominal outcome. The Interceptor might return its own code, or change the code being returned, but, to preserve the integrity of the Interceptor stack, all Interceptors must return a result code.

There are three Interceptor coding idioms.

1. A "before" Interceptor executes logic before the Action, but nothing afterwards. The `PrepareInterceptor` is a "before" Interceptor.
2. An "after" Interceptor executes logic after the Action, but nothing before.
3. A "before and after" Interceptor executes logic both before and after the Action.

The example *Interceptor Idioms* shows the skeleton code for each type.

**Example 3.12. Interceptor Idioms**

```
public class BeforeInterceptor extends AbstractInterceptor {
    public String intercept(ActionInvocation invocation)
        throws Exception {
        // Code before logic here
        return invocation.invoke();
    }
}

public class AfterInterceptor extends AbstractInterceptor {
    public String intercept(ActionInvocation invocation)
        throws Exception {
        var string result = invocation.invoke();
        // Code after logic here
        return result;
    }
}

public class BeforeAndAfterInterceptor
    extends AbstractInterceptor {
    public String intercept(ActionInvocation invocation)
        throws Exception {
        // Code before logic here
        var string result = invocation.invoke();
        // Code after logic here
        return result;
    }
}
```

To configure an Interceptor, first define it as an interceptor element, and then add it to an interceptor-stack element.

### Example 3.13. Custom Interceptor

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//
  DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
  <package name="struts-custome">
    <interceptors>
      <interceptor name="custom"
        class="interceptors.CustomInterceptor"/>
      <interceptor-stack name="customStack">
        <interceptor-ref name="defaultStack"/>
        <interceptor-ref name="custom"/>
      </interceptor-stack>
    </interceptors>
    <default-interceptor-ref name="customtStack"/>
  </package>
</struts>
```

#### 3.3.8.2. Can we use an Interceptor to access Hibernate?

One good use of a custom Interceptor is to access a data access framework, like Hibernate. Usually this means defining a Hibernate Session Factory that can be injected into Actions as a property. The *HibernateInterceptor.java* listing shows one example.

**Example 3.14. HibernateInterceptor.java**

```
package interceptors;
import com.opensymphony.xwork.ActionInvocation;
import com.opensymphony.xwork.interceptor.;
import javax.servlet.ServletException;
import org.hibernate.SessionFactory;
import org.hibernate.StaleObjectStateException;
// The application must implement a Hibernate SessionFactory
import hibernate.Factory;
public class HibernateOpenSessionInViewInterceptor extends Interceptor {
private SessionFactory hibernateSessionFactory;
public void init() {
    System.out.println("Initializing " +
        "HibernateOpenSessionInViewInterceptor interceptor, " +
        "obtaining Hibernate SessionFactory from FactoryHibernate");
    hibernateSessionFactory = Factory.getSessionFactory();
}
public void destroy() { }
public String intercept(ActionInvocation invocation) throws Exception {
    hibernateSessionFactory.getCurrentSession().beginTransaction();
    var string result = invocation.invoke();
    try {
        hibernateSessionFactory.getCurrentSession().
            getTransaction().commit();
    } catch (StaleObjectStateException staleEx) {
        System.err.println("This interceptor does not implement optimi
+ "concurrency control!");
        System.err.println("Your application will not work until you a
+ "compensation actions!");
        throw staleEx;
    } catch (Throwable ex) {
        ex.printStackTrace(); // Rollback only
        try {
            if (hibernateSessionFactory.getCurrentSession()
                .getTransaction().isActive()) {
                System.out.println("Trying to rollback database transactio
+ "after exception");
                hibernateSessionFactory.getCurrentSession().
                    getTransaction().rollback();
            } } catch (Throwable rbEx) {
                System.err.println("Could not rollback transaction after exc
+ " - " + rbEx);
            } throw new ServletException(ex); // Let others handle it
        } return result;
    }
}
```

As with the HibernateAction, the same sort of Interceptor could be written to use iBATIS, Cayenne, or any data access framework.

## 3.3.9. Review: What are Interceptors?

Let's fill in the blanks to review some key take-away points.

- Interceptors inject custom \_\_\_\_\_ into the request processing pipeline.
- Much of the core \_\_\_\_\_ of the framework is implemented as Interceptors.
- Custom Interceptors are (hard/easy) to add.

### 3.3.9.1. Answer: What are Interceptors?

- Interceptors inject custom *logic* into the request processing pipeline.
- Much of the core *functionality* of the framework is implemented as Interceptors.
- Custom Interceptors are *easy* to add.

## 3.3.10. What is the Value Stack?

A key reason we create web application is so that we can create content "on the fly", usually by incorporating information from a database system. Of course, not all the content is dynamic. In practice, a web page will be a mix of static content and dynamic values.

For example, after we log into an application, the system might show our name at the top of the page. The page we see might say "Welcome, Trillian!" But, the markup for the original page might read `Welcome <c:out value="{userName}" />`. Before presenting the page, the framework substitutes "Trillian" for `Welcome <c:out value="{userName}" />`. If a someone else was logged in, the system might instead substitute "Zaphod" or "Jeltz".

### 3.3.10.1. How does the system send dynamic values out to the page?

Since dispatching values to a page is a common need, unsurprisingly, the Java servlet framework provides a ready-made answer. In a web application, components are handed a "request" object that includes a place where we could store a variable like "userName". The system can lookup "userName" in the request and merge it into the JSP template.

In fact, just to cover all the bases, the Java platform provides four storage scopes: application, session, request, and page. When we request a value like "userName", a JSP tag can check each scope looking for a attribute called "userName". The page scope is checked first, then the request, session, and application scopes. If an attribute is found, the search stops, and the corresponding value is returned.

Sounds easy enough ... and it is. But, in practice, we have *a lot* of these values. To preserve our sanity, we usually organize the values into one or more data objects. In fact, Java

even defines a special object for the express purpose of storing values, cleverly called a *JavaBean*.

## JavaBeans

Environmentalists implore us to "reduce, reuse, recycle". We often use that same mantra in software development. Most projects require us to write a lot custom code<sup>5</sup>, and we are always looking for ways to "reduce, reuse, and recycle" the code we write.

JavaBeans are an approach to creating reusable software components. Most software components boil down to events acting on data. The JavaBean specification defines an common pattern for naming event-handling methods and data-storage methods. The data- storage methods are called properties. JavaBean properties can also be persisted, or "serialized", so that the state of the "bean" can be reused at another time. We can define a JavaBean for use in one part of the program, and reuse it in another, or define a JavaBean that can be reused between programs. Standalone JavaBean components can be used to extend another application. For example, a data input control might be defined as a JavaBean component, and plugged into a visual tool for building user interfaces.

Web frameworks like Apache Struts use JavaBeans to carry data between layers in the application. A Struts Action might collect input in a JavaBean object and pass it to a database framework, like iBATIS. Instead of designing Struts and iBATIS to work together, we can design both to work with JavaBeans.

One reason JavaBeans are so popular is because they are easy to create. Simple JavaBeans don't rely on a special interface or base class. To qualify as a simple JavaBean, an object need only follow a three design conventions.

The Simple JavaBean conventions are:

- Implements java.io.Serializable interface
- Provides a no argument constructor
- Provides "getter" and "setter methods for accessing properties

See the figure "PersonBean" for a coding example.

```
public class PersonBean implements java.io.Serializable {
    private String name;
    private boolean deceased;
    // No-arg constructor (takes no arguments).
    public PersonBean() { }
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }
    // Different semantics for a boolean field (is vs get)
    public boolean isDeceased() { return this.deceased; }
    public void setDeceased(boolean deceased)
        this.deceased = deceased; }
}
```

More complex beans can also define *Bound*, *Constrained*, and *Index* properties. For more about Complex JavaBeans, see the Sun JavaBean Trail.<sup>6</sup>

<sup>5</sup> Or not: <http://www.sqlonrails.org/>

<sup>6</sup> <http://java.sun.com/docs/books/tutorial/javabeans/index.html>

In a typical application, usually we would not look for a standalone *userName* variable, but for a *userName* property on a JavaBean object. A common reference might be `<c:out value="\${user.userName}"/>`, where *user* is a JavaBean, and *userName* is a property on the bean.

In Struts 1, the framework even designates a special JavaBean to act as an input buffer, the "ActionForm". The framework catches input from a form, puts it into an ActionForm, and then validates the input values. If the values fail validation, the ActionForm is used to repopulate the form, so that original input is not lost.

Because the ActionForm is used to validate input, Struts 1 expects ActionForms to have all String properties. Otherwise, if someone typed "tangerine" into a numeric field, we would not be able to redisplay the invalid input. The system would try to convert "tangerine" into a number, fail, and have nowhere to store the invalid string (because "tangerine" is not a number).

### 3.3.10.2. What can we do to simplify binding and validating values?

Looking back over the workflows, we can identify two key problems. First, we have to couple a server page tag to both a JavaBean name *and* a property name. (To get one value, we have to specify two attributes.)

Second, a JavaBean property can have only one type. If the input doesn't match the type, then there is no place to store the original input. Data is lost.

Hmmmm. What's needed is place where we can obtain a value just by specifying a property name, and a place where we can store values with different types under the same property name.

The Struts 2 Value Stack combines the notion of searching a sequence of scopes with the notion of an LIFO stack (last-in, first-out). We can put any number of JavaBeans onto the stack, and then ask it for a property, like "userName". The Value Stack looks through its set of objects, one by one, until it finds one with a "userName" property. The last object put on the stack will be the first one searched, and so the most "recent" object wins, as shown by the figure *Value Stack: Last In First Out*.

### Figure 3.10. Value Stack: Last In First Out



In practice, the Value Stack delivers exactly what a server page wants. The page just wants the value of the "userName" property. It doesn't care whether "userName" is on the "address" bean or the "customer" bean or the "profile" bean. It just wants the *userName*, without the red tape.

During a validation workflow, if the input doesn't match the target property type, we can push the invalid input value directly onto the Value Stack. When the input form is repopulated, the Struts tag will find and redisplay the property we pushed. The other property is still on the stack, in its own *JavaBean*, but its earlier on the stack, so the validation property wins.

Meanwhile, if the input is valid, the system will automatically convert strings to the target types, and vice versa. If a form is repopulated, some values will come from the target *JavaBean*, and others may come from a validation property pushed on to the stack afterwards. But, either way, the tag only has to ask for the property by name. The Value Stack resolves which instance of the property is returned.

#### 3.3.10.3. Are there other use cases for the Value Stack?

The *iterate* tag makes good use of the Value Stack. A dynamic page will often loop through a set of entries to create a list of items. For each pass through the loop, the Struts *iterate* tag will push the next value onto the Value Stack. At the end of the loop, the *iterate* tag will pop that property, so that the value is only available within the scope of the tag.

Another common use is transient properties. Often, the application's data model is represented as a *JavaBean*. Since Struts 2 will automatically convert data types, a web application may be able to use the data model beans directly. However, web applications often use properties that are not part of the data model. For example, when creating a password, we often have the user enter the password twice, to validate the data entry. The confirmation password is not part of the data model, but it is still a property that we need to collect and validate.

When there are properties absent from the data model, we can define these "transient" properties on the Action class. If both the model bean and the Action are on the stack, the

tags can access properties on either object interchangeably. The properties reside on two different objects, but to the tag, it's all one namespace.

### 3.3.10.4. How do we get our own properties onto the Value Stack?

The easiest way to get a property onto the Value Stack is to add it to the Action class. The Action is automatically pushed onto the stack. Once that happens, there is nothing more for us to do, and our Action properties can be accessed directly by name.

### 3.3.10.5. How do we get another JavaBean onto the Value Stack?

If you already have a JavaBean with the properties, the framework provides an interface/Interceptor duo that makes it easy to add another data object to the Value Stack.

The *ModelAware* interface defines a *getModel* property. Just provide an instance of the JavaBean via the *getModel* property, and the rest is automatic.

#### Example 3.15. ModelDriven.java

```
package com.opensymphony.xwork2;
public interface ModelDriven<T> {
    T getModel();
}
```

The framework will obtain and push the JavaBean onto the stack. Then the Struts Tags, like *property* and *iterator*, can work with the POJO properties directly.

#### Example 3.16. ModelDrivenInterceptor.java

```
public class ModelDrivenInterceptor extends AbstractInterceptor {

    public String intercept(ActionInvocation invocation) throws Exception {
        Object action = invocation.getAction();

        if (action instanceof ModelDriven) {
            ModelDriven modelDriven = (ModelDriven) action;
            ValueStack stack = invocation.getStack();
            if (modelDriven.getModel() != null) {
                stack.push(modelDriven.getModel());
            }
        }
        return invocation.invoke();
    }
}
```

As shown in the listings *ModelDriven.java* and *ModelDrivenInterceptor.java*, all we need to do is obtain a reference to the stack via *ValueStack stack = invocation.getValueStack()* and then use *stack.push()*.

To make it easy to obtain framework internals from anywhere in an application, the framework exposes a context object via thread local. An Action can obtain the context object, which contains references to other framework members, including the *ActionInvocation*.

```
ActionContext.getContext().getInvocation().getValueStack().push(o);
```

OK, so it's not *that* easy, but we can still do it in one line of code!



### Note

Pushing an object on the stack from the Action works fine for read-only properties, but a read-write object needs to be added by an Interceptor. Why? Because autopopulation (via the *ParametersInterceptor*) takes place before the Action executes. To add multiple read-write beans, copy and revise the *ModelInterceptor* and add the customized version to the Interceptor stack.

## 3.3.11. Review: What is the ValueStack?

- The ValueStack builds a \_\_\_\_\_ of objects
- Objects are examined to find property \_\_\_\_\_
- The ValueStack allows the \_\_\_\_\_ to find property values across multiple objects

### 3.3.11.1. Review: What is the ValueStack?

- The ValueStack builds a *stack* of objects
- Objects are examined to find property *values*
- The ValueStack allows the *expression language* to find property values across multiple objects

## 3.3.12. What is OGNL?

In its purest form, software is a mechanism that turns input into output.

In a web application, input will come in the form of a request from a browser. Ultimately, output is rendered back to the browser in the form of HTML. The challenge is injecting dynamic values into the static markup, so that we can customize the output.

Another term for injecting values is "binding". If we are using JavaServer Pages, we can create variables on the page. But to inject a dynamic value into the variable, we need to bind it with a JavaBean that we pass to the page.

The conventional JSP/Java approach is to put the JavaBean into one of the standard scopes (request, session, application). A JSP scriptlet can then bind a page variable to the JavaBean, and output the value.

```
<%  
User user = (User) session.get('user');  
String username = user.getUsername();  
%>  
<p>Username: <%= username %> </p>
```

To bind one property, we wrote three lines of Java code! Java might be a great way to code business logic, but it's a verbose and error-prone way to merge simple variables into HTML markup!

Java is designed as a general purpose language that can be used to create enterprise software. Customizing markup is a task best left to a language that specializes in easy access to data stored in pre-existing objects: an expression language.

To expose a simple JavaBean property using the Struts 2 expression language, we could write

```
<p>Username: <s:property name="#session.user.username" /> </p>
```

Here, "#session.user.username" does the same work as "((User) session.get("user")).getUsername()".

There are several expression languages being used today. The expression language used by Struts 2 is known as OGNL, or "Object Graph Navigation Language". The acronym is pronounced as a word, like the last syllables of a slurred pronunciation of "orthogonal."

By default, most attributes passed to a Struts 2 tag are interpreted as OGNL expressions. If we code a form tag, like

```
<s:textfield name="username" />
```

We are saying "check the Value Stack for a "username" property. If the property was not found, the tag would use the static value instead.



## Note

Native OGNL doesn't understand about the Value Stack. The framework "tricks" OGNL into thinking the Value Stack is a single JavaBean.

You may have noticed that in the first example, we used a pound sign (#) in our OGNL expression, but in the second example we did not. The reason relates to the OGNL notion of a "root" object.

An OGNL expression is evaluated in a context that may contain several objects. One of those objects can be designated the root. When we refer to a property of the root object, we don't need to reference the object, we can just reference the property directly. (The root object acts a bit like the Value Stack, except that in stock OGNL, the root is a single object.) To refer to other objects in the OGNL context, we can prefix the object's name with the pound sign ("#").

The framework automatically sets the Value Stack as the OGNL root object. Accordingly, Value Stack properties can be referenced directly, without any redtape or semantic sugar.

The OGNL context is set to use the framework's own Action Context. This context contains references to the standard request and session objects and other key resources. The Action Context resources are available to an OGNL expression via the # operator, but these objects are not necessarily on the Value Stack. (The Value Stack is actually part of the Action Context.)

The table *ActionContext properties* lists some of the properties available through the Action Context include:

**Table 3.1. ActionContext properties**

application	ServletContext as a Map
locale	Java locale object for this client
name	Name of the action
parameters	HttpServletRequest parameters as a Map
session	HttpSession as a Map

### 3.3.12.1. Are there other ways to refer to OGNL Expressions?

If a property is stored in the Value Stack and in a standard scope, there are several different ways to refer to the same property. The table *Expression Language Notations* summarizes various uses of the  $\${}$ ,  $\%{}$ , and  $\#{}$  notations.

**Table 3.2. Expression Language Notations**

<code>&lt;p&gt;Username: \${user.username}&lt;/p&gt;</code>	A JavaBean object in a standard context in Freemarker, Velocity, or JSTL EL (Not OGNL).
<code>&lt;s:textfield name="username" /&gt;</code>	A username property on the Value Stack.
<code>&lt;s:url id="es" action="Hello"&gt;   &lt;s:param name="request_locale" value="es" /&gt; &lt;/s:url&gt; &lt;s:a href="%{es}"&gt;Español&lt;/s:a&gt;</code>	Another way to refer to a property placed on the Value Stack.
<code>&lt;s:property name="#session.user.username" /&gt;</code>	The username property of the User object in the Session context.
<code>&lt;s:select label="FooBar" name="foo" list="#{'username':'trillian','username':'zaphod'}" /&gt;</code>	A static Map, as in <code>put("username","trillian")</code> .

### 3.3.12.2. What are some other examples of OGNL expressions?

OGNL is easy to learn by example, yet powerful in its application. An advantage is that the same expression is usually used for both getting and setting properties. The table *OGNL samples* demonstrates some common expressions.

**Table 3.3. OGNL Samples**

OGNL Expression	Java Equivalent
<code>user.name</code>	<code>getUser.getName()</code>
<code>user.toString()</code>	<code>getUser().getName()</code>
<code>item.categories[0]</code>	Accesses first element of Categories collection
<code>@com.example.Test@foo()</code>	Calls the static <code>foo()</code> method on the <code>com.example.Text.class</code>
<code>name in {null,"fred"}</code>	Calls <code>getName()</code> on each category in the collection, returning a new collection (projection)
<code>if test="task=='Edit'"</code>	<code>(task.equals("Edit"))</code>

The table *OGNL operators* summarizes the most common OGNL operators.

**Table 3.4. OGNL Operators**

Assignment operator	$e1 = e2$
Conditional operator	$e1 ? e2 : e3$
Logical or operator	$e1    e2, e1 \text{ or } e2$
Logical and operator	$e1 \&\& e2, e1 \text{ and } e2$
Equality test	$e1 == e2, e1 \text{ eq } e2$
Inequality test	$e1 != e2, e1 \text{ neq } e2$
Less than comparison	$e1 < e2, e1 \text{ lt } e2$
Less than or equals comparison	$e1 <= e2, e1 \text{ lte } e2$
Greater than comparison	$e1 > e2, e1 \text{ gt } e2$
Greater than or equals comparison	$e1 >= e2, e1 \text{ gte } e2$
List membership comparison	$e1 \text{ in } e2$
List non-membership comparison	$e1 \text{ not in } e2$

### 3.3.12.3. Is OGNL just something that we use with the Struts Tags?

The framework embeds OGNL everywhere, including the configuration files. In the *Expression Validator* example, we use a OGNL expression with a validator element.

```
<validator type="expression">
  <param name="expression">password eq password2</param>
  <message key="error.password.match" />
</validator>
```

If the passwords are not equal, then validation fails.

Another use of OGNL is with message resources.

```
requiredstring = ${getText(fieldName)} is required.
```

We can use OGNL expressions to customize messages by accessing properties and calling methods on Value Stack objects.

### 3.3.12.4. Review - OGNL

- OGNL stands for O\_\_\_\_\_ G\_\_\_\_\_ Navigation L\_\_\_\_\_
- OGNL is an expression and \_\_\_\_\_ language for getting and setting properties of Java objects Within the framework, OGNL is embedded everywhere -- views, ValueStack, xml configuration files.
- Within the framework, OGNL is \_\_\_\_\_ everywhere - views, ValueStack, XML configuration files.

### 3.3.12.4.1. Answers - OGNL

- OGNL stands for *Object Graph Navigation Language* .
- OGNL is an expression and *binding* language for getting and setting properties of Java objects Within the framework, OGNL is embedded everywhere -- views, ValueStack, XML configuration files.
- Within the framework, OGNL is *embedded* everywhere - views, ValueStack, xml configuration files.

## 3.4. Review: Building Struts 2 Applications

At the beginning of this session, we set out to answer three key questions:

- Can we use a conventional design?
- Why use frameworks?
- What value does Struts 2 add?

We saw that a conventional Model-View-Controller design works well with applications, so long as we forgo a direct connection between model and view in favor of using the controller as a "go-between".

We learned that frameworks provide teams with better code reuse, encourage incremental development, ease long-term maintenance. Most importantly, frameworks help us separate concerns and keep us from repeating ourselves.

We explored the key features of the Struts framework, including action handlers, result handlers, and the custom tags. Peering closer, we examined the framework's internal components: interceptors, the value stack, and the expression language.

### 3.4.1. Segue

Apache Struts 2 follows classic design patterns, deftly extending MVC to support enterprise-ready web applications. In the session *Migrating to Apache Struts 2*, we convert an application from Struts 1 to Struts 2, highlighting the architectural improvements made by the latest version.

## 3.5. Training Guide

What role does a framework play in web application development?

## 3.5.1. Presentation - Building Struts 2 Applications

We overview the framework in broad strokes.

- Can we use a conventional design?
- Why use frameworks?
- What value does Struts 2 add?

## 3.5.2. Coding Exercise - "Welcome"

Story: Visitors to the MailReader application can select tasks available to them from the default Welcome menu.

- Create action mapping
- Create server pages with JSP tags
- Configure default action

## 3.5.3. Prerequisites

- Hello
  - web.xml, resources.properties, index.html
  - struts.xml, Hello.action, HelloTest
  - Hello.jsp, taglib, text tag
- Supporting Technologies
  - HTML, HTTP, Java, Servlets (hopefully all understood)
  - JavaBeans, XML configuration, JSP Taglibs (cover for servlet based teams)
- Welcome
  - default-action
  - url tag

## 3.5.4. Exercises

- Utilize module created in Preliminary Workshop
- Review "Welcome" use case

- Create a "Missing" JSP template with the text
  - "This feature is under construction. Please try again in the next iteration."
- Create an action mapping for Missing, making it the default
- Create a Welcome page and action to realize the Welcome user case
- Refer to "Missing" in the links
- Update `{{index.html}}` to load "Welcome.action"
- Play test the page and links

### 3.5.5. Hint Code

Linking to other resources through Uniform Resource Identifiers (URIs) is a vital part of coding web applications. Struts 2 provides a tag that specializes in rendering URIs.

```
<a href="<s:url action="Register"/>">Register with MailReader</
```

Aside from action, the uri tag can also link to local and remote pages.

### 3.5.6. Accomplishments

- Created action mapping
- Created server pages with JSP tags
- Configured default action

### 3.5.7. Extra Credit

- Replace the literal text in Welcome page with messages

Do the same with Missing

### 3.5.8. Use Case: Welcome

Visitors can select tasks available to them from the default Welcome menu.

*Narrative:*

Trillian is traveling without a laptop and would like to check her mail. Using the hotel's business center, she goes to the MailReader website. The default page asks Trillian if she would like to logon, create a new account or display the page in another language.
--

<i>MailReader Options</i>
---------------------------

- Register with MailReader
- Log into MailReader

*Language Options*

- English
- Japanese
- Russian
- Vogon

*Elements:*

Goal:	Select top-level command (User Goal)
Trigger:	Visitor opens default Welcome page
Primary Actor:	Visitor

*Main Success Scenario (MSS):*

<b>Step</b>	<b>Action</b>
1	Visitor (or Subscriber) opens Welcome menu
2	Welcome presents Register, Login, and Localize commands
3	Visitor or Subscriber selects a command
4	System invokes selected command

*Preconditions and Guarantees:*

Preconditions:	Client is not authenticated
----------------	-----------------------------

---

# Chapter 4. Migrating to Apache Struts 2

## *A tutorial for Struts 1 Developers*

This tutorial helps Struts 1 developers become knowledgeable Struts 2 developers as quickly as possible by leveraging our hard-earned experience.



### Note

While included in *Apache Struts 2 from Square One*, the tutorial chapter is designed so that it can also be read on its own.

Over the course of the tutorial, we will migrate a Struts 1 application to Apache Struts 2, pointing out the similarities and differences as we go.

The tutorial is intended for readers who have already developed a web application with Struts 1, or a similar framework, like Spring MVC. A working knowledge about how Java web applications are built is needed to follow the material presented in this tutorial.

## 4.1. What are some frequently-asked questions about the Struts 2 release?

Before we dig into the code, let's set the scene by answering a few baseline questions:<sup>1</sup>

- What is Struts 2?
- How are Struts 1 and Struts 2 alike?
- What's changed in Struts 2?
- Is Struts 1 obsolete?
- Is it difficult to migrate from Struts 1 to Struts 2?
- Why are "plain old Java objects" important?

With introductions out of the way, we will dig in and start migrating Struts 1 code to Struts 2. In this session, we'll keep it simple with a "Hello World" example. In later sessions, we move up to a realistic, data-driven application.

Almost all of the source code is presented as we go. The complete source code for the tutorial is available from our Google Code site [<http://code.google.com/p/sq1-struts2/>].

---

<sup>1</sup> Portions of this section are based on material provided by the Apache Software Foundation under the Apache License and modified for inclusion here.

## 4.1.1. What is Struts 2?

Apache Struts is a free open-source framework for creating Java web applications. Today, the Apache Struts Project offers two major versions of the Struts framework.

Struts 1 is recognized as the most popular web application framework for Java. The 1.x framework is mature, well-documented, and widely supported. More teams now use Struts 1 than all other Java web frameworks combined. Struts 1 is the best choice for teams who value proven solutions to common problems.

Struts 2 was originally known as WebWork 2. After working independently for several years, the WebWork and Struts communities joined forces to create Struts 2. The new framework is the best choice for teams who value elegant solutions to difficult problems.

## 4.1.2. How are Struts 1 and Struts 2 alike?

Both versions of Struts provide three key components.

1. A "request" handler that maps Java classes to web application URIs.
2. A "response" handler that maps logical names to server pages, or other web resources.
3. A tag library to help us create rich, responsive, form-based applications.

In Struts 2, all three components have been redesigned and enhanced, but the same architectural hallmarks remain.

## 4.1.3. What's changed in Struts 2?

Struts 2 is designed to be simpler to use and closer to how Struts was always meant to be. Some key changes are:

### 4.1.3.1. Smarter!

- *Improved Design* - All Struts 2 classes are based on interfaces. Core interfaces are HTTP independent.
- *Intelligent Defaults* - Most configuration elements have a default value that we can set and forget.
- *Enhanced Results* - Unlike ActionForwards, Struts 2 Results can actually help prepare the response.
- *Enhanced Tags* - Struts 2 tags don't just output data, but provide stylesheet-driven markup, so that we can create consistent pages with less code.
- *First-class AJAX support* - The AJAX theme gives interactive applications a significant boost.

- *Stateful Checkboxes* - Struts 2 checkboxes do not require special handling for false values.

### 4.1.3.2. Easier!

- *Easy-to-test Actions* - Struts 2 Actions are HTTP independent and can be tested without resorting to mock objects.
- *Easy-to-customize controller* - Struts 1 lets us customize the request processor per module, Struts 2 lets us customize the request handling per action, if desired.
- *Easy-to-tweak tags* - Struts 2 tag markup can be altered by changing an underlying style sheet. Individual tag markup can be changed by editing a FreeMarker template. No need to grok the taglib API! Both JSP and FreeMarker tags are fully supported.
- *Easy cancel handling* - The Struts 2 Cancel button can go directly to a different action.
- *Easy Spring integration* - Struts 2 Actions are Spring-aware. Just add Spring beans!
- *Easy plugins* - Struts 2 extensions can be added by dropping in a JAR. No manual configuration required!

### 4.1.3.3. POJO-ier!

- *POJO forms* - No more ActionForms! We can use any JavaBean we like or put properties directly on our Action classes. No need to use all String properties!
- *POJO Actions* - Any class can be used as an Action class. We don't even have to implement an interface!

## 4.1.4. Is Struts 1 obsolete?

As cool as Struts 2 may be, there is a robust and vibrant community of developers using Struts 1 in production, and we expect that thousands of teams will continue to base new projects on Struts 1, and continue to support existing projects, for many, many years to come.

Of course, if you are starting a new project, and you have your choice of versions, this would be an excellent time to consider whether you would like to continue to use Struts 1 or whether it's time to try Struts 2.

## 4.1.5. Is it difficult to migrate from Struts 1 to Struts 2?

It's somewhat difficult to migrate from Struts 1 to Struts 2. The work itself is not difficult, but the amount of effort is non-trivial. As this tutorial will show, migrating Actions and pages does take time and effort. For existing applications that will not change, or change

much, migration may not be worth the effort. But, for applications that will continue to change and grow, the time invested will be well spent. Struts 2 is smarter, easier, and best of all, POJO-ier!

### 4.1.6. Why are POJOs important?

Being able to use Plain Old Java Objects with Struts 2 means that we don't have to use "extra" objects just to placate the framework. One good example is testing Struts Action classes outside of a container. In Struts 1, if an Action class uses servlet state, we have to use a special mock object to simulate the servlet. In Struts 2, we can simulate servlet state with a plain old HashMap.

Another good example is transferring form input to an object. In Struts 1, the Actions and tags expect us to use ActionForms with all String properties. In Struts 2, we can use whatever object we want to capture input, including the Action class itself. Best of all, non-String properties are not a problem.

With Struts 2, it's easy to use dependency injection to create POJOs or Actions. Internally, Struts uses Google Guice to create objects, but plugins make it easy to use other dependency injection systems, including Spring and Plexus.

### 4.1.7. Review: How are Struts 1 and Struts 2 alike?

Let's reinforce some key take-away points.

- A "r\_\_\_\_\_" handler maps Java classes to web application URIs
- A "r\_\_\_\_\_" handler maps logical names to web resources
- A \_\_\_ library creates rich, form-based applications

#### 4.1.7.1. Answer: How are Struts 1 and Struts 2 alike?

- A *"request"* handler maps Java classes to web application URIs
- A *"request"* handler maps logical names to web resources
- A *"response"* handler maps logical names to web resources
- A tag library creates rich, form-based applications

## 4.2. Where should a migration begin?

When we have an existing Struts 1 application to migrate, the simplest approach is to add the Struts 2 JARs and migrate the application a page at a time. We can use both versions

in the same web application together. The configuration and the tags have been overhauled for Struts 2, but web application architectures can remain the same. Many changes are just a matter of removing Struts 1 red tape that is no longer needed. Other changes are just a matter of swapping one tag for another.

To get started, let's migrate a "Hello World" application from Struts 1 to Struts 2. Initially, the application simply displays a welcome message. As part of the migration, we will internationalize the application so that it can display a message in two languages. Then, we will extend it again to add a data entry form so we can input our own message and validate the data entry. (Remember, the complete source code for the tutorial is available from the Struts University site.)

First, we should add the Struts 2 JARs to our Struts 1 application's `WEB-INF/lib` folder, and the corresponding elements to the application's `web.xml` file.

The JARs we need to add are

1. `struts-core-*.jar`
2. `struts-api-*.jar`
3. `commons-logging-*.jar`
4. `freemarker-*.jar`
5. `ognl-*.jar`

The elements we need to add are

1. The Struts 2 Filter
2. A filter mapping

The listing shows a complete example.

**Example 4.1. A web.xml with both Struts 1 and Struts 2 enabled**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <!-- Struts 2 -->
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <!-- Struts 1 -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
      org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>
        /WEB-INF/classes/struts-config.xml
      </param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!-- Either version -->
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```



## Tip

Tip: With filters, it's better for us to use `/*` as the URI pattern. The default extension for Struts 2 actions can be set in the `struts.properties` file (new to Struts 2). Prefix mapping is not supported in the default distribution.

Once the Struts 2 elements are added to a Struts 1 application, we can use both versions together. The Struts 1 actions can handle the `*.do` URIs, and the Struts 2 actions can handle the `*.action` URIs. So we can migrate one to the other, until there's nothing left to "do"!

## 4.3. Is the Struts 2 configuration file different?

Compared to Struts 1, the Struts 2 configuration file is streamlined. Fewer elements are configured, and the remaining elements need fewer attributes. Even the element names are shorter. The factory default name for Struts 1 is `struts-config.xml`. The default for Struts 2 is just `struts.xml`.

Let's compare the configurations for our simple Hello World application by putting the listings end to end.

## Example 4.2. Struts 1 Configuration for our "Hello World" application

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//
  DTD Struts Configuration 1.3//EN"
  "http://struts.apache.org/dtds/struts-config_1_3.dtd">
<struts-config>

  <form-beans>
    <form-bean name="HelloForm"
      type="forms.HelloForm">
    </form-bean>
  </form-beans>

  <action-mappings>
    <action path="/Hello"
      name="HelloForm"
      type="actions.HelloAction"
      validate="false">
      <forward name="success"
        path="/HelloPage.jsp" />
    </action>
  </action-mappings>

  <message-resources parameter="resources" />

</struts-config>
```

### Example 4.3. Struts 2 Configuration for our "Hello World" application

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//
    DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
  <include file="struts-default.xml" />

  <package name="hello-default"
    extends="struts-default">

    <action name="Hello"
      class="actions.Hello">
      <result>/Hello.jsp</result>
    </action>

  </package>
</struts>
```

As a Struts 1 application goes, the first listing seems concise. But, even so, it's still longer than the Struts 2 listing.

In the Struts 1 listing, we map both an Action and ActionForm. In the Struts 2 listing, we map only the Action class. In Struts 2, the Action and ActionForm are combined. Instead of declaring an ActionForm, we can place the message property directly on the Struts 2 Action class.

To convert a copy of our Hello World configuration from Struts 1 to Struts 2, first we

1. Replace the DTD
2. Change `<struts-config>` to `<struts>`
3. Add `<include file="struts-default.xml"/>`
4. Remove the `<form-beans>` element
5. Change `<action-mappings>` to `<package name="hello-default" extends="struts-default">`
6. Update each `<action>` element

To update each `<action>` element, we

1. Remove from `<action>` the "name" attribute

2. Change the `<action>` "path" attribute to "name", and "type" to "class"
3. Change the `<forward>` element into a `< result>` element.

## 4.4. Why are there so many changes to the Struts configuration?

There are three key reasons why the Struts configuration has changed so much:

- Obsolescence
- consistency, and
- comprehension.

*Obsolescence.* Some elements, like `<form- beans>`, are obsolete in Struts 2. We just plain don't need them anymore.

*Consistency.* Attribute names are applied consistently across the framework. For example, the attribute "class" is used to indicate a true Java class, as it does in the Spring and iBATIS frameworks. The attribute "type" identifies a "nickname" for a class. Some confusing attribute names, like "path", are avoided altogether.

*Comprehension.* Other elements are streamlined to be more concise and easier to understand. Verbose elements with redundant attributes take more effort to create now, and they will be harder to understand later.

Under the heading "comprehension", a very big change in Struts 2 is the notion of "Intelligent Defaults". Take for example the Struts 1 `<forward>` element and the corresponding Struts 2 `<result>` element shown in the listing.

### Example 4.4. A Struts 1 `<forward>` element and a corresponding Struts 2 `<result>` element

```
<forward name="success" path="/Hello.jsp"/>
<result>/Hello.jsp</result>
```

In the case of `<result>`, there are three "Intelligent Defaults" being set.

1. The `<result>` name defaults to "success".
2. The `<result>` content defaults to "location" (e.g. path).
3. The `<result>` type defaults to "dispatch".

In this one example, Intelligent Defaults save sixteen characters out of forty-three (a 37% reduction). But, the real bonus is readability. At a glance, the `<result>` element is much easier to read and understand.

Of course, we can override the defaults as needed. If the Action returned "cancel", and we wanted to redirect instead of forward, we can specify an alternate name and result type.

### **Example 4.5. Listing: Selecting a different result type**

```
<result name="cancel" type="redirect">/Hello.jsp
</result>
```

As shown by the listing, with the added information, the result element becomes more verbose. But that's OK. The important thing is that the most common `<result>` is the easiest to specify.



### **Tip**

*The Doctrine of Intelligent Defaults.* When an essential value is omitted, the system automatically provides a predefined value, eliminating the need to explicitly qualify each and every aspect of a declaration.<sup>2</sup>

Intelligent defaults make the Struts configuration both more concise and easier to understand.

## **4.5. Do the Action classes change too?**

Like the Struts 2 configuration, the Action classes become simpler in most ways. S2 Action classes can combine the utility of a Struts 1 ActionForm, so sometimes we will add properties to capture form input.

Let's look at the ActionForm and Actions for our simple Hello World application.

**Example 4.6. Struts 1 Hello ActionForm and Action class**

```
package actions;
import javax.servlet.http.*;
import org.apache.struts.action.*;

public class HelloAction extends Action {

    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        HelloForm input = (HelloForm) form;
        input.setMessage(MESSAGE);
        return mapping.findForward(SUCCESS);
    }

    public static final String MESSAGE = "Hello World!";
    public static final String SUCCESS = "success";

}
```

**Example 4.7. Struts 2 Hello Action class (includes form properties)**

```
package actions;
import com.opensymphony.xwork2.ActionSupport;

public class Hello extends ActionSupport {

    public String execute() throws Exception {
        setMessage(MESSAGE);
        return SUCCESS;
    }

    public static final String MESSAGE = "Hello World!";

    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

}
```

As seen in the listings, migrating an Action from Struts 1 to Struts 2 is mainly a matter of simplifying the old code. Instead of storing the Message property on a separate object, we can just add the property to the Action class.



### Tip

Unlike Struts 1 Actions, Struts 2 Actions are not singletons. Struts 2 Actions do not need to be thread-safe, and we can use member variables, if desired.

To convert our Hello World Action class from Struts 1 to Struts 2, first we

1. Update or remove imports. (A well-factored Struts 2 Action should not need to refer to anything from the `javax.servlet.http` package.)
2. Move the Message property from the ActionForm to the Action, and remove the ActionForm object completely. It is obsolete.
3. Change the Struts 2 Action to extend ActionSupport. Extending ActionSupport is optional, but it is usually helpful. In this case, we are just using the predefined SUCCESS token.
4. Remove the ActionForm cast and reference the Message property directly, since it is now a property on the Action class.
5. Remove the SUCCESS static, since it is already defined on ActionSupport.

## 4.6. What about the tags?

Unsurprisingly, the Struts 2 tags also become simpler and require fewer attributes.

### Example 4.8. Struts 1 "Hello" server page

```
<%@ taglib prefix="bean"
  uri="http://struts.apache.org/tags-bean" %>
<html>
  <head>
    <title>Hello World!</title>
  </head>

  <body>
    <h2><bean:write name="HelloForm"
      property="message" />
    </h2>
  </body>
</html>
```

### Example 4.9. Struts 2 "Hello" server page

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h2><s:property value="message" /></h2>
  </body>
</html>
```

To convert from Struts 1 to Struts 2 the Hello World server pages shown in the listings, first we

1. Replace the `<%@ taglib %@>` directive
2. Change the `<bean:write ... />` tag to a `< s:property ... />` tag

As with the configuration and Action class, most of the changes help streamline the server page. The `<bean:write>` tag needs to know the name of the object. The `<s:property>` tag finds the property automatically. The name of the ActionForm is no longer coupled to the page.

See the Quick Reference Guide in the appendix for a table listing Struts 1 tags and Struts 2 equivalents. In most cases, the Struts 2 tags are not just simple equivalents. The new tags provide significant improvements over the Struts 1 libraries.

## 4.7. Can we still localize messages?

Both Struts 1 and Struts 2 use standard message resource bundles. If we have an existing bundle, it can be registered in the `struts.properties` file.

### Example 4.10. Registering a Struts 2 message resource bundle via `struts.properties`

```
struts.custom.i18n.resources = resources
```

The listing indicates that we will have a resource bundle with the root name "resources.properties" and alternate files named in the style "resource\_es.properties", where "es" is a standard locale code. Other valid codes might include "ja" or "ru", or even "fr\_CA", for Canadian French.

Both the resource bundle and the `struts.properties` file can be placed in the classes directory of a web application, so that they can be read from the classpath. (Other configurations are possible.)

For Hello World, the resource bundles, shown by the listing, are simple enough.

**Example 4.11. Hello World Resource bundles**

```
resources.properties
  message = Hello World
```

```
resources_es.properties
  message = Hola Mundo!
```

Since we are setting the "Hello" message in our Action, we can lookup the appropriate message by key, and set the localized message to our form property.

**Example 4.12. Changes to Struts 1 Hello Action to enable internationalism**

```
package actions;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import forms.HelloForm;

public class HelloAction extends Action {

    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        HelloForm input = (HelloForm) form;
-       input.setMessage(MESSAGE);
+       input.setMessage(
+           getResources(request).getMessage(MESSAGE));
        return mapping.findForward(SUCCESS);
    }

-   public static final String MESSAGE = "Hello World!";
+   public static final String MESSAGE = "message";
    public static final String SUCCESS = "success";

}
```

In the listing, the lines with "-" in the margin are being removed. The lines marked "+" are being added.

For Struts 2, as the listing shows, the i18n syntax is a tad cleaner.

**Example 4.13. Changes to Hello Action to enable internationalism**

```

public class Hello extends ActionSupport {

    public String execute() throws Exception {
-       setMessage(MESSAGE);
+       setMessage(getText(MESSAGE));

        return SUCCESS;
    }

-   public static final String MESSAGE = "Hello World!";
+   public static final String MESSAGE = "message";
    }

```

**Tip**

If Struts 1 can't find a message, it returns null. If Struts 2 can't find a message, it returns the message key.

As with Struts 1, some of the Struts 2 tags will output localized messages, based on a key passed to the tag. Right now, our page has "Hello World" embedded in the title. Let's change the title so that it looks up the message, just like the Action does.

**Example 4.14. Struts 1 "Hello" server page**

```

<%@ taglib prefix="bean"
    uri="http://struts.apache.org/tags-bean" %>
<html>
<head>
-   <title>Hello World!</title>
+   <title><bean:message key="message"/></title>
</head>

    <body>
        <h2><bean:write name="HelloForm"
            property="message" /></h2>
    </body>
</html>

```

**Example 4.15. Struts 2 "Hello" server page**

```

<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head>
-   <title>Hello World!</title>
+   <title><s:text name="message"/></title>
  </head>
  <body>
    <h2><s:property value="message"/></h2>
  </body>
</html>

```

Now we can change the title and body text in one place.

## 4.8. How do we change Locales in Struts 2?

The Locale is stored in the user's session with the Java container. To change locales in Struts 1, most often, we cobble up some kind of LocaleAction class. The listing shows a typical implementation.

**Example 4.16. Changing the locale in Struts 1 (LocaleAction)**

(TODO)

To kick off changing the locale, we need to add a link to our Struts 1 Locale action. The listing shows a typical locale link.

**Example 4.17. Changing the locale in Struts 1 (JSP)**

```

<li>
  <html:link action="/Locale?language=es">Español</html:link>
</li>

```

In Struts 2, we just add the appropriate "magic" links to a page, and the framework will change the locale automatically. An additional Action class is not needed.

**Example 4.18. Changing the locale in Struts 2 (JSP)**

```

<li>
  <s:url id="es" action="Hello">
    <s:param name="request_locale">es</s:param>
  </s:url>
  <s:a href="%{es}">Español</s:a>
</li>

```

From the listing, the key takeaway is that we need to pass "request\_locale=es" as a request parameter, where "es" can be any standard locale code. The framework sees the "magic" parameter and updates the locale for the instant session.

## 4.9. Does Struts 2 use the Commons Validator?

Struts 1 uses the Commons Validator framework. Struts 2 uses a similar but different validation framework. The Struts 2 validation framework is part of Open Symphony XWork. (Open Symphony was the original host of WebWork 2, which became Struts 2.)

Validation comes into play when we submit an input form for processing. Before we add an input form, let's look at how both versions of Struts configures its respective validation framework.

In Struts 1, we can add a `validations.xml` file to the application with a stanza that describes how to validate our "HelloForm", as shown by the listing.

### Example 4.19. The Struts 1 `validations.xml` configuration file for HelloForm

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
  "-//Apache Software Foundation//
  DTD Commons Validator Rules Configuration 1.3.0//EN"
  "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
  <formset>
    <form name="HelloForm">
      <field property="message" depends="required">
        <arg key="message"/>
      </field>
    </form>
  </formset>
</form-validation>
```

In a larger application, we would continue to add `<form>` elements for each `ActionForm` to be validated.

In Struts 2, the configuration files are finely-grained. We create a `validation.xml` file for each object being validated. In this case, since we want to validate the `actions.Hello` class, we create a `Hello-validation.xml` file in the `actions` package, like the one shown by the listing.

**Example 4.20. The Struts 2 validation file for Hello (actions/Hello-validation.xml)**

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE validators PUBLIC
  "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
  "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>
  <field name="message">
    <field-validator type="requiredstring">
      <message key="requiredstring"/>
    </field-validator>
  </field>
</validators>
```

Right now, our Hello Actions are providing a default message. Let's add a data entry form and validate the input to be sure there is message, then we wouldn't need to set the message in Action class.

In Struts 1, the usual approach to adding an input form is to add another mapping.

**Example 4.21. Adding an input form to Hello World for Struts 1**

```
<!-- ... -->

  <action-mappings>

    <action path="/HelloInput"
      name="HelloForm"
      forward="/HelloInput.jsp"
      validate="false" />

    <action path="/Hello"
      name="HelloForm"
      forward="/HelloPage.jsp"
      validate="true"
      input="/HelloInput.do" />

  </action-mappings>
</struts-config>
```

Since we don't need the Action class anymore, we also changed the "Hello" mapping in the listing to forward directly to our display page. Following the new mappings, we can add a HelloInput.jsp to carry our form. The listing shows our form markup for a HelloInput page.

**Example 4.22. A Struts 1 input form (HelloInput.jsp)**

```

<html:errors/>
<html:form action="/Hello" focus="message"
  onsubmit="return validateRegistrationForm(this);">
<table>
  <tr><td>
    Message:
  </td><td>
    <html:text property="message" />
  </td></tr>
</table>
</html:form>

```

**Tip**

We are using the Struts 1 HTML taglib on this page, so we need to import the taglib into the JSP.

In Struts 2, it's easy to reuse the same action mapping for the input form. Built into Struts 2 is the capability to call an alternate method. In Struts 1, we can do this by using some type of "dispatch" Action. In Struts 2, calling an alternate method is a first-class feature of the framework. The listing shows how we can call alternate methods using wildcards.

**Example 4.23. Adding an input form to Hello World for Struts 2**

```

<action name="Hello_*" method="{1}" class=
"actions.Hello">
  <result>/Hello.jsp</result>
  <result name="input">/Hello_input.jsp</result>
</action>

```

In the Struts 2 rendition, when we call "Hello\_input.action", the mapping will not invoke the usual "execute" method. Instead, it will call the "input" method by replacing the "{1}" with the characters matching the wildcard "\*" in the action name. We called "Hello\_input", so the framework invokes the "input" method.

As with the Struts 1 rendition, we can provide an input page to go with the Struts 2 mapping. Since the Struts 2 tags provide their own markup, the input form is concise, as shown by the listing.

**Example 4.24. A Struts 2 input form (Hello\_input.jsp)**

```

<s:actionerror/>
<s:form action="Hello">
  <s:textfield label="Message" name="message"/>
</s:form>

```

In the Struts 2 action mapping, we make use of wildcards to call an alternate method. The same sort of thing can be done with Struts 1, but it's more complicated. One complication is that we need to turn validation off for an input action, but we need validation turned on for a submit action.

In Struts 2, the base `ActionSupport` class has a special input method that does nothing but return the token "input". If we call an "input" method, by design, the framework will bypass validation. Usually, the only thing that happens on input is that the framework prepares and displays an "input" page.

Most often, the input page will post back to the same action, but to the default "execute" method this time. Some developers might use a "save" or "submit" method instead of "execute", but the notion is the same. We use one Action class method for input and another for submit.

Finally, we should update the resource bundle with localized error messages.

### **Example 4.25. Resource bundle with localized error messages**

```
resources.properties
prompt = Enter Message
message = Hello World!
# Struts 1
errors.required={0} is required.
# Struts 2
requiredstring = ${getText(fieldName)} is required.

resources_es.properties
prompt = Entre el mensaje
message = Hola Mundo!
# Struts 1
errors.required={0} se requiere.
# Struts 2
requiredstring = ${getText(fieldName)} se requiere.
```

As shown in the listing, Struts 1 and Struts 2 take a different tact to using replacement tokens in messages. In Struts 1, a numeric parameter is used (up to 4). In Struts 2, we have access to the same expression language and symbolic parameters that we use with the tags.

## **4.10. Is that all there is?**

We weren't able to cover everything in this brief tutorial, but we did cover the basics: mappings, Actions, tags, localization, and validation.

We converted a simple Hello World application from Struts 1 to Struts 2 by moving Actions, pages, and configuration elements over, one at a time. We were also able to share essential resources, like message bundles, between the versions.

Struts 2 is a giant leap forward, but, for Struts 1 developers, it's a learning curve we can walk, one step at a time.

---

## Part II. Quick Reference

We collect together the essential, everyday details that Struts developers need at their fingertips.



### Tip

Visit the Struts Mentor website [<http://www.StrutsMentor.com/>] to schedule a live training course at your location.

---

# Appendix A. Quick Reference

## A.1. Maven Configuration

What is the Maven Artifact ID? Where can we get Maven snapshot releases?<sup>1</sup>

### A.1.1. Maven Artifact ID

```
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-core</artifactId>
  <version>2.0.7-SNAPSHOT</version>
</dependency>
```

### A.1.2. Maven Snapshot Repositories

```
<repositories>
  <repository>
    <id>ASF_SNAPSHOT_M2</id>
    <name>ASF Maven 2 Snapshot</name>
    <url>
      http://people.apache.org/repo/m2-snapshot-repository
    </url>
  </repository>
</repositories>
```

## A.2. Use the Source

What are the key framework classes and configuration files? How do we use these resources in our applications? Where can we find the source? What goes where in the project tree?

**Table A.1. key framework classes and configuration files**

Action.java	ActionSupport.java
Validator.java	ValidatorSupport.java
Interceptor.java	default.properties
default.xml   validators.xml	struts-default.xml
struts-message.properties	struts-portlet-default.xml
struts-tags.tld	

---

<sup>1</sup> Portions of this section are based on material provided by the Apache Software Foundation under the Apache License and modified for inclusion here.

## A.2.1. Action.java

- Actions are the framework's basic unit of work. Most framework requests are handled by an Action, with help from Interceptors, Results, and the Struts Tags.
- To create your own Action, implement the Action interface or extend ActionSupport (preferred).
  - Much of Struts 2 extends the XWork framework, ActionSupport included.
- XWork interface [`com.opensymphony.xwork2.Action`]

## A.2.2. ActionSupport.java

- ActionSupport is a default implementation of an Action class that provides all the functionality that most applications need.
- Extend as the base of your own Actions or Action support class.
  - Surprisingly, ActionSupport is another XWork class.
- ActionSupport is the default action class. If an action class is not specified, ActionSupport is used. To specify a different default class, use the `default-action-ref` package element.
- XWork class [`com.opensymphony.xwork2.ActionSupport`]

## A.2.3. Interceptor.java

- Interceptors can invoke code before and after the Action executes. Interceptors encapsulate utility that can be shared between Actions into a reusable object.
- To create your own interceptor, implement the Interceptor interface and declare the interceptor in the `struts.xml` configuration file.
- XWork interface [`com.opensymphony.xwork2.interceptor.Interceptor`]

## A.2.4. Validator.java

- Validators vet incoming data and generate an error message when input is invalid.
- The framework provides all the validators that most applications will need. To create a custom validator, implement the Validator interface, copy the `default.xml` from the XWork source code, edit it, and bundle the new `default.xml` with your web application under `WEB-INF/classes`.
- XWork interface [`com.opensymphony.xwork2.validator.Validator`]

## A.2.5. ValidatorSupport.java

- Abstract implementation of the Validator interface suitable for subclassing.
- XWork class [`com.opensymphony.xwork2.validator.validators.ValidatorSupport`]

## A.2.6. default.properties

- Factory defaults for various framework settings.
- To override one or more settings, create a `struts.properties` file and specify the new setting value.
- properties file [`org/apache/struts/default.properties` ]

## A.2.7. default.xml | validators.xml

- Standard set of validators.
- The standard set of validators is declared in the `default.xml` file.
- To override, provide a *replacement* `validators.xml` file at the *root* of the classpath (e.g. under `WEB-INF/classes` in a web application). All validators to be used must be specified in the `validators.xml` .
  - Hint: Extract the `default.xml` from the XWork JAR, rename it as `validators.xml`, and modify it.
- XWork XML document [`com/opensymphony/xwork2/validator/validators/default.xml` ]

## A.2.8. struts-default.xml

- Default configuration packaged in the `struts2-core.jar` and automatically included in application configuration.
- XML document [`/struts-default.xml`]

## A.2.9. struts-message.properties

- Default messages used by the framework, with support for English, German, and Dutch.
- To add another locale to the bundle, create a `org/apache/struts2` folder under `classes` and provide the appropriate property file.
- properties file [`/struts-messages>`]

## A.2.10. struts-portlet-default.xml

- Default settings for Struts 2 portlets.
- To override, place a replacement file under `WEB-INF/classes`
- properties file [`/struts-portlet-default.xml`]

## A.2.11. struts-tags.tld

- Taglib descriptor for the Struts Tags.
- The TLD is an internal framework class and should not need to be changed.

- TLD [META-INF/struts-tags.tld]

## A.3. XML Document Types (DocType)

What XML DocTypes do we use with Struts 2?

### A.3.1. Struts Configuration XML DocType reference

```
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//
  DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">
```

For a complete listing, download the DTD via a web browser.<sup>2</sup>

### A.3.2. XWork Validator XML DocType Reference

```
<!DOCTYPE validators PUBLIC
  "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
  "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
```

For a complete listing, access the DTD via a web browser.<sup>3</sup>

## A.4. Common Configurations

What are the essential configuration files that most applications need?

**Table A.2. Essential configuration files**

web.xml	struts.xml
struts-plugin.xml	

### A.4.1. web.xml

- Web deployment descriptor to include all necessary framework components.
- Implements Web Application 2.3 DTD (Servlet 2.3 Specification)
- WEB-INF/web.xml
- *required*

<sup>2</sup> <http://struts.apache.org/dtds/struts-2.0.dtd>

<sup>3</sup> <http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd>

## A.4.2. struts.xml

- Main configuration, contains result/view types, action mappings, interceptors, and so forth.
- Custom framework settings (in <constant> section)
- Implements `struts-2.0.dtd`
- *required*

## A.4.3. struts-plugin.xml

- Optional configuration file for plugins. Each plugin JAR may have its own.
- Implements `struts-2.0.dtd`
- Place at the root of a plugin JAR.
- *optional*

## A.5. Struts 2 web.xml elements

How is the standard `web.xml` configured? What about adding optional elements, like SiteMesh? What is the format for the Spring configuration file?

### A.5.1. Typical Struts 2 web.xml

```
<web-app>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
  </filter>
  <filter-mapping><filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <!-- ... -->
</web-app>
```

### A.5.2. Fully-loaded Struts 2 web.xml with optional elements

```
<web-app>
  <filter>
    <filter-name>struts2-cleanup</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.ActionContextCleanUp
    </filter-class>
  </filter>
  <filter>
```

```
<filter-name>struts2</filter-name>
<filter-class>
    org.apache.struts2.dispatcher.FilterDispatcher
</filter-class>
</filter>
<filter>
    <filter-name>sitemesh</filter-name>
    <filter-class>
        com.opensymphony.module.sitemesh.filter.PageFilter
    </filter-class>
</filter>

<!-- SiteMesh for FreeMarker (use instead)
<filter>
    <filter-name>sitemesh</filter-name>
    <filter-class>
        org.apache.struts2.sitemesh.FreeMarkerPageFilter
    </filter-class>
</filter>
-->

<!-- SiteMesh for Velocity (use instead)
<filter>
    <filter-name>sitemesh</filter-name>
    <filter-class>
        org.apache.struts2.sitemesh.VelocityPageFilter
    </filter-class>
</filter>
-->

<filter-mapping>
    <filter-name>struts2-cleanup</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>sitemesh</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

```
<!-- MyFaces (JSF)
<listener>
  <listener-class>
    org.apache.myfaces.webapp.StartupServletContextListener
  </listener-class>
</listener>
-->

<!-- DWR (1 of 2)
<servlet>
  <servlet-name>dwr</servlet-name>
  <servlet-class>
    uk.ltd.getahead.dwr.DWRServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>>true</param-value>
  </init-param>
</servlet>
-->

<!-- JSF
<servlet>
  <servlet-name>faces</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>faces</servlet-name>
  <url-pattern>*.action</url-pattern>
</servlet-mapping>
-->

<!-- DWR (2 of 2)
<servlet-mapping>
  <servlet-name>dwr</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
-->
</web-app>
```

## A.6. Configuration Examples

What are some common configuration settings?

## A.6.1. struts.properties with alternate settings

```
struts.devMode = true
struts.enable.DynamicMethodInvocation = false
struts.action.extension=action,do
struts.custom.i18n.resources=alternate,alternate2
```

## A.6.2. Struts Configuration XML with HelloWorld action

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//
  DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
  <package name="package" extends="struts-default">
    <action name="HelloWorld" class="example.HelloWorld">
      <result>/example/HelloWord.jsp</result>
    </action>
    <!-- Add actions here -->
  </package>
</struts>
```

## A.6.3. Struts Configuration XML with Wildcards

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//
  DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
  <package name="actions" extends="struts-default">
    <global-results>
      <result name="error">/Error.jsp</result>
      <result>{1}/execute.jsp</result>
      <result name="input">{1}/input.jsp</result>
    </global-results>
    <global-exception-mappings>
      <exception-mapping
        result="error" exception="java.lang.Throwable"/>
    </global-exception-mappings>
    <action name="*/*" class="actions.{1}" method="{2}" />
  </package>
</struts>
```

## A.7. Interceptors

What interceptors come preconfigured? What other interceptors are bundled with the framework?

**Table A.3. Default interceptor stack**

exception	alias	servlet-config	prepare
il8n	chain	debugging <i>(new)</i> <sup>a</sup>	profiling <i>(new)</i>
scoped-model-driven	model-driven	fileUpload	checkbox <i>(new)</i>
static-params	params	conversionError	validation
workflow	roles <i>(new)</i>		

<sup>a</sup> *(new)* = New in S2

**Table A.4. Other bundled interceptors**

autowiring	createSession	execAndWait	external-ref
logger	scope	sessionAutowiring	store
timer	token	token-session	

## A.8. Validators

What Validators come with the framework? What methods are excluded from Validation?



### Tip

Data conversion is automatic. Input does not need to be verified for a target format, like `date` or `double`, unless the value must also fall within a certain range. As a rule, only required fields and special cases need to be validated.

**Table A.5. Predefined Validators**

required	requiredstring	int	double
date	expression	fieldexpression	email
url	visitor	conversion	stringlength
regex			

## Example A.1. Validator examples

```
<validators>
  <field name="username">
    <field-validator type="required">
      <message>username must not be null</message>
    </field-validator>

    <field-validator type="requiredstring">
      <param name="trim">false</param>
      <message>username is required</message>
    </field-validator>
  </field>

  <field name="birthday">
    <field-validator type="date">
      <param name="min">01/01/1990</param>
      <param name="max">01/01/2000</param>
      <message>Birthday must be within
        ${min} and ${max}</message>
    </field-validator>
  </field>

  <validator type="expression">
    <param name="expression">password eq password2</param>
    <message>Password and confirmation password
      must match</message>
  </validator>

  <field name="email">
    <field-validator type="email">
      <message>Must provide a valid email</message>
    </field-validator>
  </field>
</validators>
```

## A.8.1. Methods excluded from validation

input, back, cancel, browse

## A.8.2. Specifying the methods excluded from validation

To change the list of methods, copy the default Interceptor stack and modify the setup for the Validator interceptor.

```
<interceptor-stack name="defaultStack">
  <!-- ... -->
  <interceptor-ref name="validation">
    <param name="excludeMethods">
      input,back, cancel, browse</param>
    </interceptor-ref>
  <interceptor-ref name="workflow">
    <param name="excludeMethods">
      input,back, cancel, browse</param>
    </interceptor-ref>
</interceptor-stack>
```

## A.9. Result Types

What result types are preconfigured? What are the default result codes?

**Table A.6. Predefined result types**

chain	dispatcher <i>(default)</i>	freemarker	httpheader
plaintext	redirect <i>(popular)</i>	redirect- action <i>(popular)</i>	stream
tiles <i>(new!)</i>	velocity	xslt	

### A.9.1. Setting a default result type

```
<result-types>
  <result-type name="dispatcher"
    class="org.apache.struts2.dispatcher.ServletDispatcherResult"
    default="true"/>
</result-types>
```

**Table A.7. Default Result Names**

SUCCESS	"success"
NONE	"none"
ERROR	"error"
INPUT	"input"
LOGIN	"login"

## A.10. Exception Handling

How do we specify a global exception handler? How do we insert the exception messages into a page?

### A.10.1. Specifying a global exception handler

```
<global-results>
  <result name="Error">/Error.jsp</result>
</global-results>

<global-exception-mappings>
  <exception-mapping exception="java.lang.Throwable"
    result="Error"/>
</global-exception-mappings>
```

### A.10.2. Inserting error and exception messages

```
<s:actionerror />
<p>
  <s:property value="%{exception.message}"/>
</p>
<hr/>
<h3>Technical Details</h3>
<p>
  <s:property value="%{exceptionStack}"/>
</p>
```

## A.11. OGNL

What are the OGNL operators? What are the different OGNL notations? What are some common OGNL expressions?

**Table A.8. Common OGNL Operators**

Assignment operator	<code>e1 = e2</code>
Conditional operator	<code>e1 ? e2 : e3</code>
Logical or operator	<code>e1    e2, e1 or e2</code>
Logical and operator	<code>e1 &amp;&amp; e2, e1 and e2</code>
Equality test	<code>e1 == e2, e1 eq e2</code>
Inequality test	<code>e1 != e2, e1 neq e2</code>
Less than comparison	<code>e1 &lt; e2, e1 lt e2</code>
Less than or equals comparison	<code>e1 &lt;= e2, e1 lte e2</code>
Greater than comparison	<code>e1 &gt; e2, e1 gt e2</code>
Greater than or equals comparison	<code>e1 &gt;= e2, e1 gte e2</code>
List membership comparison	<code>e1 in e2</code>
List non-membership comparison	<code>e1 not in e2</code>

**Table A.9. Expression Language Notations**

<code>&lt;p&gt;Username: \${user.username}&lt;/p&gt;</code>	A JavaBean object in a standard context in Freemarker, Velocity, or JSTL EL (Not OGNL).
<code>&lt;s:textfield name="username" /&gt;</code>	A username property on the Value Stack.
<code>&lt;s:url id="es" action="Hello"&gt;   &lt;s:param name="request_locale" value="es" /&gt; &lt;/s:url&gt; &lt;s:a href="%{es}"&gt;Español&lt;/s:a&gt;</code>	Another way to refer to a property placed on the Value Stack.
<code>&lt;s:property name="#session.user.username" /&gt;</code>	The username property of the User object in the Session context.
<code>&lt;s:select label="FooBar" name="foo" list="#{'username':'trillian','username':'zaphod'}" /&gt;</code>	A static Map, as in <code>put("username","trillian")</code> .

```
# Embedded OGNL expression in validation message
requiredstring = ${getText(fieldName)} is required.
```

```
<!-- Referencing an Action property from a tag -->
<s:property value="postalCode" />
```

```
<!-- Referencing a non-root object in the ActionContext -->
<s:property value="#session.mySessionPropKey" /> or
```

```
<s:property value="#session["mySessionPropKey"]"/> or
<s:property value="#request["mySessionPropKey"]/>

<s:property value="%{#application.myApplicationAttribute}" />
<s:property value="%{#session.mySessionAttribute}" />
<s:property value="%{#request.myRequestAttribute}" />

<s:property value="%{#parameters.myParameter}" />
<s:property value="%{#myContextParam}" />
<s:property value="#session.mySessionPropKey"/> or
<s:property value="#session["mySessionPropKey"]"/> or
<s:property value="#request["mySessionPropKey"]/>

<s:property value="%{#application.myApplicationAttribute}" />
<s:property value="%{#session.mySessionAttribute}" />
<s:property value="%{#request.myRequestAttribute}" />

<s:property value="%{#parameters.myParameter}" />
<s:property value="%{#myContextParam}" />

<!-- Passing a static List to a tag -->
<s:select label="label" name="name"
    list="#{'foo':'foovalue', 'bar':'barvalue'}" />

<!-- Passing a static Map to a tag -->
<s:select label="label" name="name"
    list="#{'foo':'foovalue', 'bar':'barvalue'}" />

<!-- Using an expression to set the label -->
<s:textfield label="%{getText("postalCode.label")}"
    name="postalCode"/>

<!-- Passing a literal value to a textfield -->
<s:textfield key="state.label" name="state"
    value="%{'CA'}" />

<!-- Evaluating a boolean (static) -->
<s:select key="state.label" name="state"
    multiple="%{true}"/>

<!-- Evaluating a boolean (property) -->
<s:select key="state.label" name="state"
    multiple="allowMultiple"/>
```

## A.12. Static Content

How can we refer to static content that doesn't need to be processed by the framework?

- Implicit path to static resources: `/struts/$package-or- folder`
  - e.g., `/struts/default.css` maps to `/WEB-INF/classes/template/default.css`
- Or any other "template" folder on the classpath (e.g. in a JAR)

## A.12.1. Adding other static content locations

```
<filter>
  <filter-name>struts2</filter-name>
  <filter-class>
    org.apache.struts2.dispatcher.FilterDispatcher
  </filter-class>
  <set-param name="packages" value="styles,images" />
</filter>
```

With this change,

- `/struts/default.css` will also find `/WEB-INF/classes/styles/default.css`
- `/struts/image.png` will also `/WEB-INF/classes/images/image.png`

Or any other "styles" or "images" folder on the classpath (e.g. in a JAR).

## A.13. Struts Tags

What are the various Struts Tags? Are there common attributes?

**Table A.10. Control Tags**

<code>if</code>	<code>elseif</code>	<code>else</code>	<code>append</code>
<code>generator</code>	<code>iterator</code>	<code>merge</code>	<code>sort</code>
<code>subset</code>			

**Table A.11. Data Tags**

<code>action</code>	<code>bean</code>	<code>debug</code>	<code>i18n</code>
<code>include</code>	<code>param</code>	<code>push</code>	<code>set</code>
<code>text</code>	<code>url</code>	<code>property</code>	

**Table A.12. Form Tags**

checkbox	checkboxlist	combobox
datepicker	doubleselect	head
file	form	hidden
label	optiontransferselect	optgroup
password	reset	select
submit	textarea	textfield
timepicker	token	updownselect

**Table A.13. Non-Form Tags**

a	actionerror	actionmessage	component
date	div	fielderror	panel
table	tabbedPanel	tree	treenode

**Table A.14. General Tag Attributes**

cssClass	cssStyle	title
disabled	label	labelPosition
requiredposition	name	required
tabIndex	value	

**Table A.15. Tooltip Related Tag Attributes**

tooltip	jsTooltipEnabled	tooltipIcon	tooltipDelay
---------	------------------	-------------	--------------

**Table A.16. Javascript Related Tag Attributes**

onclick	ondbclick	onmousedown	onmouseover
onmouseout	onfocus	onblur	onkeypress
onkeyup	onkeydown	onselect	onchange

**Table A.17. Template Related Tag Attributes**

templateDir	theme	template	
-------------	-------	----------	--

## A.14. Struts Tag examples

What are some common tag usages?

```
<!-- A simple page with one form and one text field -->
```

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <body>
    <s:actionerror/>
    <s:form action="Hello">
      <s:textfield label="Message" name="message"/>
    </s:form>
  </body>
</html>

<!-- A form with localized labels -->
<s:actionerror />
<s:form action="Login" validate="true">
  <s:textfield key="username"/>
  <s:password key="password" showPassword="true"/>
  <s:submit key="button.save"/>
  <s:reset key="button.reset"/>
  <s:submit action="Login_cancel" key="button.cancel"
    onclick="form.onsubmit=null"/>
</s:form>

<!-- Using POJO input field -->
<s:textfield label="fullName" name="user.fullName"/>

<!-- Using a button to submit to a different action -->
<s:submit key="button.save" action="Registration_save"/>

<!-- A cancel button that submits to a different action
and avoids validation -->
<s:submit action="Registration!input" key="button.cancel"
  onclick="form.onsubmit=null"/>

<!-- Using logic to present either a label or a field -->
<s:if test="task == 'Create'">
  <s:textfield key="username"/>
</s:if>
<s:else>
  <s:label key="username"/>
  <s:hidden name="username"/>
</s:else>

<!-- Using logic to change the page title -->
<head>
<s:if test="task=='Create'">
  <title><s:text name="registration.title.create"/></title>
</s:if>
<s:if test="task=='Edit'">
```

```
<title><s:text name="registration.title.edit"/></title>
</s:if>
<link href="<s:url value="/css/mailreader.css"/>"
      rel="stylesheet" type="text/css"/>
</head>

<!-- Changing the locale (using a link with parameters -->
<s:url id="es" action="Hello">
  <s:param name="request_locale">es</s:param>
</s:url>
<s:a href="%{es}">Espanol</s:a>

<!-- Presenting text from the message bundle -->
<s:text value="message"/>

<!-- Presenting a property from a POJO -->
<s:property value="user.fullName"/>

<!-- Iterating over a POJO (property of the Action -->
<s:iterator value="user.subscriptions">
<tr>
  <td align="left">
    <s:property value="host"/>
  </td>
  <td align="left">
    <s:property value="username"/>
  </td>
  <td align="center">
    <s:property value="type"/>
  </td>
  <td align="center">
    <s:property value="autoConnect"/>
  </td>
  <td align="center">
    <a href="<s:url action="Subscription_delete">
      <s:param name="host" value="host"/></s:url">">
      <s:text name="registration.deleteSubscription"/>
    </a>
    &nbsp;
    <a href="<s:url action="Subscription_edit">
      <s:param name="host" value="host"/></s:url">">
      <s:text name="registration.editSubscription"/>
    </a>
  </td>
</tr>
</s:iterator>
```

```
<!-- Link to an action (s:url and HTML a tags) -->  
<a href="<s:url action="Login_input"/>">  
  Try again!  
</a>
```

## A.15. Key Best Practices

What are the most important Struts best practices?

- Link only to actions, never to server pages or templates
- Do not expose the action extension in your tags. Use the action parameter instead.
- Use namespaces to organize your application into logical "modules" or units of work.
- Unit test Action classes before trying them in a web application.
- Do not embed business logic in action classes.
- Centralize application logic into a base support class that other Actions can share.
- Be on the lookout for ways to "normalize" an application. Remember: the two best best-practices are "Separate Concerns" and "Don't Repeat Yourself". Let these two practices be your guide!