



Reference Documentation

Version 1.3.0

Last Updated December 17, 2009 (Latest documentation)

Copyright © 2004-2008 Mark Pollack, Rick Evans, Aleksandar Seovic,
Bruno Baia, Erich Eichinger, Federico Spinazzi, Rob Harrop, Griffin
Caprio, Ruben Bartelink, Choy Rim, Erez Mazor, The Spring Java Team

Copies of this document may be made for your own use and for distribution to others,
provided that you do not charge any fee for such copies and further provided that
each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Preface	1
2. Introduction	2
2.1. Overview	2
2.2. Background	2
2.3. Modules	3
2.4. Usage Scenarios	4
2.5. Quickstart applications	4
2.6. License Information	5
2.7. Support	5
3. Background information	6
3.1. Inversion of Control	6
4. Migrating from 1.1 M2	7
4.1. Introduction	7
4.2. Important Changes	7
4.2.1. Namespaces	7
4.2.2. Core	8
4.2.3. Web	8
4.2.4. Data	8
I. Core Technologies	9
5. The IoC container	10
5.1. Introduction	10
5.2. Container overview	10
5.2.1. Configuration metadata	11
5.2.2. Instantiating a container	12
5.2.3. Using the container	16
5.2.4. Object definition overview	16
5.2.5. Instantiating objects	18
5.2.6. Object creation of generic types	20
5.3. Dependencies	22
5.3.1. Dependency injection	22
5.3.2. Dependencies and configuration in detail	29
5.3.3. Declarative Event Listener Registration	38
5.3.4. Using depends-on	40
5.3.5. Lazily-initialized objects	40
5.3.6. Autowiring collaborators	41
5.3.7. Checking for dependencies	42
5.3.8. Method injection	43
5.3.9. Setting a reference using the members of other objects and classes.	46
5.3.10. Provided IFactoryObject implementations	50
5.4. Object Scopes	50
5.4.1. The singleton scope	51
5.4.2. The prototype scope	51
5.4.3. Singleton objects with prototype-object dependencies	52
5.4.4. Request, session and web application scopes	52
5.5. Type conversion	52
5.5.1. Type Conversion for Enumerations	53
5.5.2. Built-in TypeConverters	53
5.5.3. Custom Type Conversion	54
5.6. Customizing the nature of an object	56

5.6.1. Lifecycle interfaces	56
5.6.2. IApplicationContextAware and IObjectNameAware	57
5.7. Object definition inheritance	58
5.8. Container extension points	59
5.8.1. Obtaining an IFactoryObject, not its product	60
5.9. Container extension points	60
5.9.1. Customizing objects with IObjectPostProcessors	61
5.9.2. Customizing configuration metadata with ObjectFactoryPostProcessors	65
5.9.3. Customizing instantiation logic using IFactoryObjects	70
5.10. The IApplicationContext	71
5.10.1. IObjectFactory or IApplicationContext?	71
5.11. Configuration of IApplicationContext	72
5.11.1. Registering custom parsers	73
5.11.2. Registering custom resource handlers	74
5.11.3. Registering Type Aliases	74
5.11.4. Registering Type Converters	75
5.12. Added functionality of the IApplicationContext	76
5.12.1. Context Hierarchies	76
5.12.2. Using IMessageSource	77
5.12.3. Using resources within Spring.NET	79
5.12.4. Loosely coupled events	79
5.12.5. Event notification from IApplicationContext	80
5.13. Customized behavior in the ApplicationContext	81
5.13.1. The IApplicationContextAware marker interface	81
5.13.2. The IObjectPostProcessor	81
5.13.3. The IObjectFactoryPostProcessor	82
5.13.4. The PropertyPlaceholderConfigurer	82
5.14. Configuration of ApplicationContext without using XML	82
5.15. Service Locator access	82
5.16. Stereotype attributes	83
6. The IObjectWrapper and Type conversion	85
6.1. Introduction	85
6.2. Manipulating objects using the IObjectWrapper	85
6.2.1. Setting and getting basic and nested properties	85
6.2.2. Other features worth mentioning	87
6.3. Type conversion	87
6.3.1. Type Conversion for Enumerations	88
6.4. Built-in TypeConverters	88
6.4.1. Custom type converters	89
7. Resources	90
7.1. Introduction	90
7.2. The IResource interface	90
7.3. Built-in IResource implementations	91
7.3.1. Registering custom IResource implementations	91
7.4. The IResourceLoader	92
7.5. The IResourceLoaderAware interface	92
7.6. Application contexts and IResource paths	93
8. Threading and Concurrency Support	94
8.1. Introduction	94

8.2. Thread Local Storage	94
8.3. Synchronization Primitives	95
8.3.1. ISync	95
8.3.2. SyncHolder	95
8.3.3. Latch	96
8.3.4. Semaphore	96
9. Object Pooling	98
9.1. Introduction	98
9.2. Interfaces and Implementations	98
10. Spring.NET miscellanea	99
10.1. Introduction	99
10.2. PathMatcher	99
10.2.1. General rules	99
10.2.2. Matching filenames	99
10.2.3. Matching subdirectories	100
10.2.4. Case does matter, slashes don't	100
11. Expression Evaluation	102
11.1. Introduction	102
11.2. Evaluating Expressions	102
11.3. Language Reference	103
11.3.1. Literal expressions	103
11.3.2. Properties, Arrays, Lists, Dictionaries, Indexers	104
11.3.3. Methods	105
11.3.4. Operators	105
11.3.5. Assignment	108
11.3.6. Expression lists	108
11.3.7. Types	108
11.3.8. Type Registration	108
11.3.9. Constructors	109
11.3.10. Variables	109
11.3.11. Ternary Operator (If-Then-Else)	110
11.3.12. List Projection and Selection	110
11.3.13. Collection Processors and Aggregators	111
11.3.14. Spring Object References	114
11.3.15. Lambda Expressions	114
11.3.16. Delegate Expressions	115
11.3.17. Null Context	116
11.4. Classes used in the examples	116
12. Validation Framework	118
12.1. Introduction	118
12.2. Example Usage	118
12.3. Validator Groups	119
12.4. Validators	120
12.4.1. Condition Validator	120
12.4.2. Required Validator	121
12.4.3. Regular Expression Validator	121
12.4.4. Generic Validator	122
12.4.5. Conditional Validator Execution	122
12.5. Validator Actions	123

12.5.1. Error Message Action	123
12.5.2. Exception Action	123
12.5.3. Generic Actions	123
12.6. Validator References	124
12.7. Programmatic usage	124
12.8. Usage tips within ASP.NET	125
12.8.1. Rendering Validation Errors	126
12.8.2. How Validate() and Validation Controls play together	127
13. Aspect Oriented Programming with Spring.NET	129
13.1. Introduction	129
13.1.1. AOP concepts	129
13.1.2. Spring.NET AOP capabilities	130
13.1.3. AOP Proxies in Spring.NET	131
13.2. Pointcut API in Spring.NET	131
13.2.1. Concepts	132
13.2.2. Operations on pointcuts	132
13.2.3. Convenience pointcut implementations	133
13.2.4. Custom pointcuts	135
13.3. Advice API in Spring.NET	136
13.3.1. Advice Lifecycle	136
13.3.2. Advice types	136
13.4. Advisor API in Spring.NET	141
13.5. Using the ProxyFactoryObject to create AOP proxies	141
13.5.1. Basics	142
13.5.2. ProxyFactoryObject Properties	142
13.5.3. Proxying Interfaces	143
13.5.4. Proxying Classes	145
13.5.5. Concise proxy definitions	145
13.6. Proxying mechanisms	146
13.6.1. InheritanceBasedAopConfigurer	147
13.7. Creating AOP Proxies Programatically with the ProxyFactory	147
13.8. Manipulating Advised Objects	148
13.9. Using the "autoproxy" facility	149
13.9.1. Autoproxy object definitions	149
13.9.2. Using attribute-driven auto-proxying	154
13.10. Using AOP Namespace	154
13.11. Using TargetSources	155
13.11.1. Hot swappable target sources	156
13.11.2. Pooling target sources	156
13.11.3. Prototype target sources	157
13.11.4. ThreadLocal target sources	158
13.12. Defining new Advice types	158
13.13. Further reading and resources	158
14. Aspect Library	159
14.1. Introduction	159
14.2. Caching	159
14.3. Exception Handling	161
14.3.1. Language Reference	164
14.4. Logging	164

14.5. Retry	166
14.5.1. Language Reference	167
14.6. Transactions	167
14.7. Parameter Validation	167
15. Common Logging	169
15.1. Introduction	169
16. Testing	170
16.1. Introduction	170
16.2. Unit testing	170
16.3. Integration testing	170
16.3.1. Context management and caching	171
16.3.2. Dependency Injection of test fixtures	171
16.3.3. Transaction management	173
16.3.4. Convenience variables	174
II. Middle Tier Data Access	175
17. Transaction management	176
17.1. Introduction	176
17.2. Motivations	176
17.3. Key Abstractions	178
17.4. Resource synchronization with transactions	179
17.4.1. High-level approach	180
17.4.2. Low-level approach	180
17.5. Declarative transaction management	180
17.5.1. Understanding Spring's declarative transaction implementation	181
17.5.2. Example of declarative transaction implementation	182
17.5.3. Declarative transactions using the transaction namespace	184
17.5.4. Transaction attribute settings	189
17.5.5. Declarative Transactions using AutoProxy	190
17.6. Programmatic transaction management	191
17.6.1. Using the TransactionTemplate	191
17.6.2. Using the PlatformTransactionManager	193
17.7. Choosing between programmatic and declarative transaction management	194
17.8. Transaction lifecycle and status information	194
18. DAO support	195
18.1. Introduction	195
18.2. Consistent exception hierarchy	195
18.3. Consistent abstract classes for DAO support	198
19. DbProvider	200
19.1. Introduction	200
19.2. IDbProvider and DbProviderFactory	200
19.3. XML based configuration	203
19.4. Connection String management	204
19.5. Additional IDbProvider implementations	205
19.5.1. UserCredentialsDbProvider	205
19.5.2. MultiDelegatingDbProvider	205
20. Data access using ADO.NET	207
20.1. Introduction	207
20.2. Motivations	208
20.3. Provider Abstraction	209

20.3.1. Creating an instance of IDbProvider	210
20.4. Namespaces	210
20.5. Approaches to Data Access	210
20.6. Introduction to AdoTemplate	211
20.6.1. Execute Callback	211
20.6.2. Execute Callback in .NET 2.0	211
20.6.3. Execute Callback in .NET 1.1	213
20.6.4. Quick Guide to AdoTemplate Methods	214
20.6.5. Quick Guide to AdoTemplate Properties	216
20.7. Transaction Management	217
20.8. Exception Translation	217
20.9. Parameter Management	217
20.9.1. IDbParametersBuilder	218
20.9.2. IDbParameters	218
20.9.3. Parameter names in SQL text	219
20.10. Custom IDataReader implementations	219
20.11. Basic data access operations	219
20.11.1. ExecuteNonQuery	220
20.11.2. ExecuteScalar	220
20.12. Queries and Lightweight Object Mapping	220
20.12.1. ResultSetExtractor	220
20.12.2. RowCallback	221
20.12.3. RowMapper	222
20.12.4. Query for a single object	223
20.12.5. Query using a CommandCreator	223
20.13. DataTable and DataSet	225
20.13.1. DataTables	226
20.13.2. DataSets	226
20.14. TableAdapters and participation in transactional context	228
20.15. Database operations as Objects	229
20.15.1. AdoQuery	229
20.15.2. MappingAdoQuery	230
20.15.3. AdoNonQuery	230
20.15.4. Stored Procedure	231
21. Object Relational Mapping (ORM) data access	233
21.1. Introduction	233
21.2. NHibernate	234
21.2.1. Resource management	234
21.2.2. Transaction Management	235
21.2.3. SessionFactory set up in a Spring container	235
21.2.4. Implementing DAOs based on plain Hibernate 1.2/2.x API	237
21.2.5. Declarative transaction demarcation	240
21.2.6. Programmatic transaction demarcation	241
21.2.7. Transaction management strategies	242
21.2.8. Web Session Management	243
21.2.9. Session Scope	244
21.2.10. Integration Testing	244
III. The Web	245
22. Spring.NET Web Framework	246

22.1. Introduction to Spring.NET Web Framework	246
22.2. Comparing Spring.NET and ASP.NET	247
22.3. Automatic context loading and hierarchical contexts	248
22.3.1. Configuration of a web application	248
22.3.2. Context hierarchy	250
22.4. Dependency injection for ASP.NET pages	251
22.4.1. Injecting dependencies into controls	252
22.4.2. Injecting dependencies into custom HTTP modules	253
22.4.3. Injecting dependencies into HTTP handlers and handler factories	253
22.4.4. Injecting dependencies in custom ASP.NET providers	254
22.4.5. Customizing control dependency injection	255
22.5. Web object scopes	256
22.6. Support for ASP.NET 1.1 master pages in Spring.Web	256
22.6.1. Linking child pages to their master page file	257
22.7. Bidirectional data binding and data model management	258
22.7.1. Data binding under the hood	262
22.7.2. Using DataBindingPanel	267
22.7.3. Customizing model persistence	268
22.8. Localization and message sources	268
22.8.1. Working with localizers	268
22.8.2. Automatic localization with localizers ("push" localization)	269
22.8.3. Global message sources	271
22.8.4. Applying resources manually ("pull" localization)	272
22.8.5. Localizing images within a web application	272
22.8.6. User culture management	273
22.8.7. Changing cultures	274
22.9. Result mapping	274
22.9.1. Registering user defined transfer modes	277
22.10. Client-side scripting	278
22.10.1. Registering scripts within the head HTML section	278
22.10.2. Adding CSS definitions to the head section	278
22.10.3. Well-known directories	278
22.11. Spring user controls	279
22.11.1. Validation controls	279
22.11.2. Databinding controls	279
22.11.3. Calendar control	279
22.11.4. Panel control	279
23. ASP.NET AJAX	280
23.1. Introduction	280
23.2. Web Services	280
23.2.1. Exposing Web Services	280
23.2.2. Calling Web Services by using JavaScript	281
IV. Services	282
24. Introduction to Spring Services	283
24.1. Introduction	283
25. .NET Remoting	285
25.1. Introduction	285
25.2. Publishing SAOs on the Server	285
25.2.1. SAO Singleton	285

25.2.2. SAO SingleCall	286
25.2.3. IIS Application Configuration	287
25.3. Accessing a SAO on the Client	288
25.4. CAO best practices	289
25.5. Registering a CAO object on the Server	289
25.5.1. Applying AOP advice to exported CAO objects	290
25.6. Accessing a CAO on the Client	290
25.6.1. Applying AOP advice to client side CAO objects.	290
25.7. XML Schema for configuration	290
25.8. Additional Resources	291
26. .NET Enterprise Services	292
26.1. Introduction	292
26.2. Serviced Components	292
26.3. Server Side	292
26.4. Client Side	294
27. Web Services	295
27.1. Introduction	295
27.2. Server-side	295
27.2.1. Removing the need for .asmx files	295
27.2.2. Injecting dependencies into web services	296
27.2.3. Exposing PONOs as Web Services	298
27.2.4. Exporting an AOP Proxy as a Web Service	299
27.3. Client-side	299
27.3.1. Using VS.NET generated proxy	300
27.3.2. Generating proxies dynamically	300
27.3.3. Configuring the proxy instance	301
28. Windows Communication Foundation (WCF)	302
28.1. Introduction	302
28.2. Configuring WCF services via Dependency Injection	302
28.2.1. Dependency Injection	302
28.3. Apply AOP advice to WCF services	304
28.4. Creating client side proxies declaratively	304
28.5. Exporting PONOs as WCF Services	305
V. Integration	306
29. Message Oriented Middleware - Apache ActiveMQ and TIBCO EMS	307
29.1. Introduction	307
29.1.1. Multiple Vendor Support	307
29.1.2. Separation of Concerns	308
29.1.3. Interoperability and provider portability	309
29.1.4. The role of Messaging API in a 'WCF world'	309
29.2. Using Spring Messaging	309
29.2.1. Messaging Template overview	309
29.2.2. Connections	310
29.2.3. Caching Messaging Resources	310
29.2.4. Dynamic Destination Management	311
29.2.5. Message Listener Containers	312
29.2.6. Transaction Management	312
29.3. Sending a Message	312
29.3.1. Using MessageConverters	313

29.. Session and Producer Callback	315
29.5. Receiving a message	315
29.5.1. Synchronous Reception	315
29.5.2. Asynchronous Reception	316
29.5.3. The ISessionAwareMessageListener interface	318
29.5.4. MessageListenerAdapater	318
29.5.5. Processing messages within a messaging transaction	320
29.5.6. Messaging Namespace support	320
30. Message Oriented Middleware - TIBCO EMS	323
30.1. Introduction	323
30.2. Interface based APIs	323
30.3. Using Spring's EMS based Messaging	324
30.3.1. Overview	324
30.3.2. Connections	324
30.3.3. Caching Messaging Resources	324
30.3.4. Dynamic Destination Management	325
30.3.5. Accessing Admistrated objects via JNDI	325
30.3.6. MessageListenerContainers	326
30.3.7. Transaction Management	327
30.3.8. Sending a Message	327
30.4. Using MessageConverters	328
30.5. Session and Producer Callback	328
30.6. Receiving a messages	328
30.6.1. Synchronous Reception	328
30.6.2. Asynchronous Reception	328
30.6.3. The ISessionAwareMessageListener interface	329
30.6.4. MessageListenerAdapter	329
30.6.5. Processing messages within a messaging transaction	329
30.6.6. Messaging Namespace support	330
31. Message Oriented Middleware - MSMQ	331
31.1. Introduction	331
31.2. A quick tour for the impatient	332
31.3. Using Spring MSMQ	334
31.3.1. MessageQueueTemplate	334
31.3.2. MessageQueueFactoryObject	336
31.3.3. MessageQueue and IMessageConverter resource management	337
31.3.4. Message Listener Containers	337
31.4. MessageConverters	342
31.4.1. Using MessageConverters	342
31.5. Interface based message processing	343
31.5.1.	343
31.6. Comparison with using WCF	344
32. Scheduling and Thread Pooling	345
32.1. Introduction	345
32.2. Using the Quartz.NET Scheduler	345
32.2.1. Using the JobDetailObject	345
32.2.2. Using the MethodInvokingJobDetailFactoryObject	346
32.2.3. Wiring up jobs using triggers and the SchedulerFactoryObject	347
33. Template Engine Support	348

33.1. Introduction	348
33.2. Dependencies	348
33.3. Configuring a VelocityEngine	348
33.3.1. Simple file based template engine definition	348
33.3.2. Configuration Options	348
33.3.3. Assembly based template loading	349
33.3.4. Using Spring's IResourceLoader to load templates	349
33.3.5. Defining a custom resource loader	350
33.3.6. Resource Loader configuration options	350
33.3.7. Using a custom configuration file	350
33.3.8. Logging	351
33.4. Merging a template	351
33.5. Configuring a VelocityEngine without a custom namespace	351
VI. VS.NET Integration	353
34. Visual Studio.NET Integration	354
34.1. XML Editing and Validation	354
34.2. Solution Templates	356
34.2.1. Class Library	356
34.2.2. ADO.NET based application library	357
34.2.3. NHibernate based application library	358
34.2.4. Spring based web application	359
34.3. Resharper Type Completion	360
34.4. Resharper templates	361
34.5. Versions of XML Schema	362
34.6. API documentation	362
VII. Quickstart applications	363
35. IoC Quickstarts	364
35.1. Introduction	364
35.2. Movie Finder	364
35.2.1. Getting Started - Movie Finder	364
35.2.2. First Object Definition	365
35.2.3. Setter Injection	366
35.2.4. Constructor Injection	366
35.2.5. Summary	367
35.2.6. Logging	368
35.3. ApplicationContext and IMessageSource	369
35.3.1. Introduction	369
35.4. ApplicationContext and IEventRegistry	371
35.4.1. Introduction	371
35.5. Pooling example	372
35.5.1. Implementing Spring.Pool.IPoolableObjectFactory	372
35.5.2. Being smart using pooled objects	374
35.5.3. Using the executor to do a parallel grep	375
35.6. AOP	375
36. AOP QuickStart	376
36.1. Introduction	376
36.2. The basics	376
36.2.1. Applying advice	376
36.2.2. Using Pointcuts - the basics	379

36.3. Going deeper	381
36.3.1. Other types of Advice	381
36.3.2. Using Attributes to define Pointcuts	387
36.4. The Spring.NET AOP Cookbook	388
36.4.1. Caching	388
36.4.2. Performance Monitoring	389
36.4.3. Retry Rules	389
36.5. Spring.NET AOP Best Practices	389
37. Portable Service Abstraction Quick Start	390
37.1. Introduction	390
37.2. .NET Remoting Example	390
37.3. Implementation	392
37.4. Running the application	397
37.5. Remoting Schema	398
37.6. .NET Enterprise Services Example	398
37.7. Web Services Example	400
37.8. Additional Resources	403
38. Web Quickstarts	404
38.1. Introduction	404
39. SpringAir - Reference Application	405
39.1. Introduction	405
39.2. Getting Started	405
39.3. Container configuration	405
39.4. Bi-directional data binding	407
39.5. Declarative Validation	407
39.6. Internationalization	408
39.7. Web Services	408
40. Data Access QuickStart	410
40.1. Introduction	410
40.1.1. Database configuration	410
40.1.2. CommandCallback	411
41. Transactions QuickStart	413
41.1. Introduction	413
41.2. Application Overview	413
41.2.1. Interfaces	413
41.3. Implementation	414
41.4. Configuration	417
41.4.1. Rollback Rules	418
41.5. Adding additional Aspects	419
42. NHibernate QuickStart	421
42.1. Introduction	421
42.2. Getting Started	421
42.3. Implementation	424
42.3.1. The Data Access Layer	424
42.3.2. The domain objects	425
42.3.3. NHibernate based DAO implementation	426
42.3.4. The Service layer	429
42.3.5. Integration testing	430
42.3.6. Web Application	432

43. Quartz QuickStart	434
43.1. Introduction	434
43.2. Application Overview	434
43.3. Standard job scheduling	434
43.4. Scheduling arbitrary methods as jobs	435
44. NMS QuickStart	437
44.1. Introduction	437
44.2. Message Destinations	437
44.3. Gateways	438
44.4. Message Data	438
44.5. Message Handlers	440
44.6. Message Converters	441
44.7. Messaging Infrastructure	441
44.8. Running the application	442
45. TIBCO EMS QuickStart	444
45.1. Introduction	444
45.2. Message Destinations	444
45.3. Messaging Infrastructure	445
45.4. Running the application	446
46. MSMQ QuickStart	447
46.1. Introduction	447
46.2. Message Destinations	447
46.3. Gateways	447
46.4. Message Data	447
46.5. Message Handlers	448
46.6. MessageConverters	448
46.7. Messaging Infrastructure	448
46.8. Running the application	450
47. WCF QuickStart	451
47.1. Introduction	451
47.2. The server side	451
47.2.1. WCF Dependency Injection and AOP in self-hosted application	452
47.2.2. WCF Dependency Injection and AOP in IIS web application	452
47.3. Client access	452
VIII. Spring.NET for Java developers	454
48. Spring.NET for Java Developers	455
48.1. Introduction	455
48.2. Beans to Objects	455
48.3. PropertyEditors to TypeConverters	456
48.4. ResourceBundle-ResourceManager	456
48.5. Exceptions	456
48.6. Application Configuration	456
48.7. AOP Framework	457
48.7.1. Cannot specify target name at the end of interceptorNames for ProxyFactoryObject	457
IX. Appendices	459
A. Classic Spring Usage	460
A.1. Classic Hibernate Usage	460
A.1.1. The HibernateTemplate	460

A.1.2. Implementing Spring-based DAOs without callbacks	461
A.2. Classic Declarative Transaction Configurations	462
A.2.1. Declarative Transaction Configuration using DefaultAdvisorAutoProxyCreator	462
A.2.2. Declarative Transactions using TransactionProxyFactoryObject	463
A.2.3. Concise proxy definitions	464
A.2.4. Declarative Transactions using ProxyFactoryObject	465
B. XML Schema-based configuration	467
B.1. Introduction	467
B.2. XML Schema-based configuration	467
B.2.1. Referencing the schemas	467
B.2.2. The tx (transaction) schema	468
B.2.3. The aop schema	469
B.2.4. The db schema	469
B.2.5. The remoting schema	470
B.2.6. The nms messaging schema	471
B.2.7. The validation schema	471
B.2.8. The objects schema	472
B.3. Setting up your IDE	472
C. Extensible XML authoring	473
C.1. Introduction	473
C.2. Authoring the schema	473
C.3. Coding a INamespaceParser	474
C.4. Coding an IObjectDefinitionParser	475
C.5. Registering the handler and the schema	476
C.5.1. NamespaceParsersSectionHandler	476
C.6. Using a custom extension in your Spring XML configuration	476
C.7. Further Resources	477
D. Spring.NET's spring-objects.xsd	478

Chapter 1. Preface

Developing software applications is hard enough even with good tools and technologies. Spring provides a light-weight solution for building enterprise-ready applications. Spring provides a consistent and transparent means to configure your application and integrate AOP into your software. Highlights of Spring's functionality are providing declarative transaction management for your middle tier as well as a full-featured ASP.NET framework.

Spring could potentially be a one-stop-shop for many areas of enterprise application development; however, Spring is modular, allowing you to use just those parts of it that you need, without having to bring in the rest. You can use just the IoC container to configure your application and use traditional ADO.NET based data access code, but you could also choose to use just the Hibernate integration code or the ADO.NET abstraction layer. Spring has been (and continues to be) designed to be non-intrusive, meaning dependencies on the framework itself are generally none (or absolutely minimal, depending on the area of use).

This document provides a reference guide to Spring's features. Since this document is still to be considered very much work-in-progress, if you have any requests or comments, please post them on the user mailing list or on the support forums at forum.springframework.net.

Before we go on, a few words of gratitude are due to Christian Bauer (of the [Hibernate](#) team), who prepared and adapted the DocBook-XSL software in order to be able to create Hibernate's reference guide, thus also allowing us to create this one. Also thanks to Russell Healy for doing an extensive and valuable review of some of the material.

Chapter 2. Introduction

2.1. Overview

Spring.NET is an application framework that provides comprehensive infrastructural support for developing enterprise .NET applications. It allows you to remove incidental complexity when using the base class libraries makes best practices, such as test driven development, easy practices. Spring.NET is created, supported and sustained by [SpringSource](#).

The design of Spring.NET is based on the Java version of the Spring Framework, which has shown real-world benefits and is used in thousands of enterprise applications world wide. Spring .NET is not a quick port from the Java version, but rather a 'spiritual port' based on following proven architectural and design patterns in that are not tied to a particular platform. The breadth of functionality in Spring .NET spans application tiers which allows you to treat it as a 'one stop shop' but that is not required. Spring .NET is not an all-or-nothing solution. You can use the functionality in its modules independently. These modules are described below.

Enterprise applications typically are composed of a number of a variety of physical tiers and within each tier functionality is often split into functional layers. The business service layer for example typically uses a objects in the data access layer to fulfill a use-case. No matter how your application is architected, at the end of the day there are a variety of objects that collaborate with one another to form the application proper. The objects in an application can thus be said to have dependencies between themselves.

The .NET platform provides a wealth of functionality for architecting and building applications, ranging all the way from the very basic building blocks of primitive types and classes (and the means to define new classes), to rich full-featured application servers and web frameworks. One area that is decidedly conspicuous by its absence is any means of taking the basic building blocks and composing them into a coherent whole; this area has typically been left to the purvey of the architects and developers tasked with building an application (or applications). Now to be fair, there are a number of design patterns devoted to the business of composing the various classes and object instances that makeup an all-singing, all-dancing application. Design patterns such as Factory, Abstract Factory, Builder, Decorator, and Service Locator (to name but a few) have widespread recognition and acceptance within the software development industry (presumably that is why these patterns have been formalized as patterns in the first place). This is all very well, but these patterns are just that: best practices given a name, typically together with a description of what the pattern does, where the pattern is typically best applied, the problems that the application of the pattern addresses, and so forth. Notice that the last paragraph used the phrase "... a description of what the pattern does..."; pattern books and wikis are typically listings of such formalized best practice that you can certainly take away, mull over, and then implement yourself in your application.

The Spring Framework takes best practices that have been proven over the years in numerous applications and formalized as design patterns, and actually codifies these patterns as first class objects that you as an architect and developer can take away and integrate into your own application(s). This is a Very Good Thing Indeed as attested to by the numerous organizations and institutions that have used the Spring Framework to engineer robust, maintainable applications. For example, the IoC component of the Spring Framework addresses the enterprise concern of taking the classes, objects, and services that are to compose an application, by providing a formalized means of composing these various disparate components into a fully working application ready for use

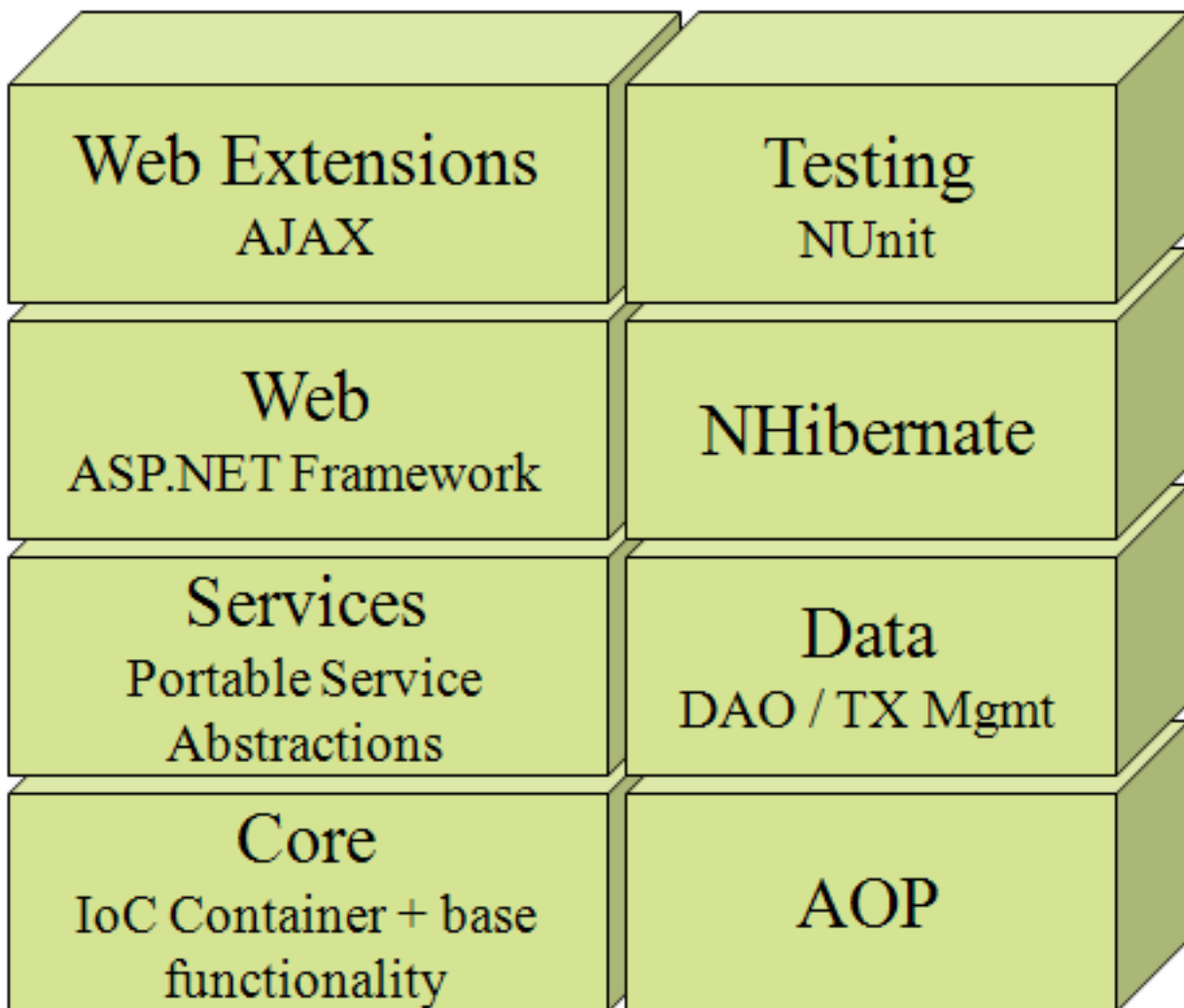
2.2. Background

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: "the question is, what aspect of control are [they] inverting?". Fowler then suggested renaming the principle (or at least giving

it a more self-explanatory name), and started to use the term Dependency Injection. His article then continued to explain the ideas underpinning the Inversion of Control (IoC) and Dependency Injection (DI) principle. If you need a decent insight into IoC and DI, please do refer to the article : <http://martinfowler.com/articles/injection.html>.

2.3. Modules

The Spring Framework contains a lot of features, which are well-organized into modules shown in the diagram below. The diagram below shows the various core modules of Spring.NET.



Click on the module name for more information.

Spring.Core is the most fundamental part of the framework allowing you to configure your application using Dependency Injection. Other supporting functionality, listed below, is located in Spring.Core

Spring.Aop - Use this module to perform Aspect-Oriented Programming (AOP). AOP centralizes common functionality that can then be declaratively applied across your application in a targeted manner. Spring's aspect library provides predefined easy to use aspects for transactions, logging, performance monitoring, caching, method retry, and exception handling.

Spring.Data - Use this module to achieve greater efficiency and consistency in writing data access functionality in ADO.NET and to perform declarative transaction management.

Spring.Data.NHibernate - Use this module to integrate NHibernate with Spring's declarative transaction management functionality allowing easy mixing of ADO.NET and NHibernate operations within the same transaction. NHibernate 1.0 users will benefit from ease of use APIs to perform data access operations.

Spring.Web - Use this module to raise the level of abstraction when writing ASP.NET web applications allowing you to effectively address common pain-points in ASP.NET such as data binding, validation, and ASP.NET page/control/module/provider configuration.

Spring.Web.Extensions - Use this module to raise the level of abstraction when writing ASP.NET web applications allowing you to effectively address common pain-points in ASP.NET such as data binding, validation, and ASP.NET page/control/module/provider configuration.

Spring.Services - Use this module to adapt plain .NET objects so they can be used with a specific distributed communication technology, such as .NET Remoting, Enterprise Services, and ASMX Web Services. These services can be configured via dependency injection and 'decorated' by applying AOP.

Spring.Testing.NUnit - Use this module to perform integration testing with NUnit.

The **Spring.Core** module also includes the following additional features

- **Expression Language** - provides efficient querying and manipulation of an object graphs at runtime.
- **Validation Framework** - a robust UI agnostic framework for creating complex validation rules for business objects either programatically or declaratively.
- **Data binding Framework** - a UI agnostic framework for performing data binding.
- **Dynamic Reflection** - provides a high performance reflection API
- **Threading** - provides additional concurrency abstractions such as Latch, Semaphore and Thread Local Storage.
- **Resource abstraction** - provides a common interface to treat the `InputStream` from a file and from a URL in a polymorphic and protocol-independent manner.

2.4. Usage Scenarios

With the building blocks described above you can use Spring in all sorts of scenarios, from simple stand alone console applications to fully-fledged enterprise applications using Spring's transaction management functionality and web framework integration.

It is important to note that the Spring Framework *does not* force you to use everything within it; it is not an *all-or-nothing* solution. Existing front-ends built using standard ASP.NET can be integrated perfectly well with a Spring-based middle-tier, allowing you to use the transaction and/or data access features that Spring offers. The only things you need to do is wire up your business logic using Spring's IoC container and integrate it into your web layer using `WebApplicationContext` to locate middle tier services and/or configure your standard ASP.NET pages with dependency injection.

While the Spring framework does not force any particular application architecture it encourages the use of a well layered application architecture with distinct tiers for the presentation, service, data access, and database.

2.5. Quickstart applications

There are several sample applications that showcase individual features. If you are already familiar with the concepts of dependency injection, AOP, or have experience using the Java version of the Spring framework you

may find jumping into the examples a better way to bootstrap the learning processing process. The following quickstart applications are available and can be found in the examples directory in the distribution. Click on the links for additional information.

- **Movie Finder** - A simple demonstration of Dependency Injection (DI) techniques using Spring's Inversion of Control (IoC) container.
- **Application Context** - Demonstrates IoC container features such as localization, accessing of ResourceSet objects, and applying resources to object properties.
- **Aspect Oriented Programming** - Demonstrates use of the AOP framework to add additional behavior to your existing objects. Examples of programmatic and declarative AOP configuration are shown.
- **Distributed Computing** - A calculator demonstrating remote service abstractions that let you 'export' a plain .NET object (PONO) via .NET Remoting, Web Services, or an EnterpriseService ServiceComponent. Corresponding client side proxies are also demonstrated.
- **WCF** - Shows a WCF based calculator example that configures your WCF service via dependency injection and apply AOP advice.
- **Web Application - SpringAir** -A ticket booking application that demonstrates the ASP.NET framework showing features such as DI for ASP.NET pages, data binding, validation, and localization.
- **Web Development** - Introductory examples showing use of dependency injection and Spring's bi-directional data binding in ASP.NET.
- **Data Access** - Demonstrates the ADO.NET framework showing how to simplify developing ADO.NET based data access layers.
- **Transaction Management** : Demonstrates the use of declarative transaction management for both local and distributed transaction in both .NET 1.1 and 2.0.
- **AJAX** : Demonstrates how to access a plain .NET object as a webservice in client side JavaScript
- **NHibernate Northwind**: Demonstrates use of Spring's NHibernate integration to simplify the use of NHibernate. Web tier is also included showing how to use the Open-Session In View approach to session management in the web tier.
- **Quartz Quickstart** - Application that shows the use of Quartz.NET integration for scheduling.
- **NMS** - Applicatoin demonstrating NMS helper classes.

2.6. License Information

Spring.NET is licensed according to the terms of the Apache License, Version 2.0. The full text of this license are available online at <http://www.apache.org/licenses/LICENSE-2.0> . You can also view the full text of the license in the license.txt file located in the root installation directory.

2.7. Support

Training and support are available through [SpringSource](#) in addition to the mailing lists and forums you can find on the main [Spring.NET](#) website.

Chapter 3. Background information

3.1. Inversion of Control

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: *"the question, is what aspect of control are they inverting?"*. After talking about the term Inversion of Control Martin suggests renaming the pattern, or at least giving it a more self-explanatory name, and starts to use the term *Dependency Injection*. His [article](#) continues to explain some of the ideas behind this important software engineering principle.

Other references you may find useful are

- Wikipedia Article - [Dependency Injection](#)
- CodeProject article - [Dependency Injection for Loose Coupling](#)

Chapter 4. Migrating from 1.1 M2

4.1. Introduction

Several API changes were made after 1.1 M2 (before 1.1 RC1) due primarily by the need to refactor the code base to remove circular dependency cycles, which are now all removed. Class and schema name changes were also made to provide a more consistent naming convention across the codebase. As a result of these changes, you can not simply drop in the new .dlls as you may have done in previous release. This document serves as a high level guide to the most likely areas where you will need to make changes to either your configuration or your code.

The file, BreakingChanges-1.1.txt, in the root directory of the distribution contains the full listing of breaking changes made for RC1 and higher

4.2. Important Changes

This section covers the common areas where you will need to make changes in code/configuration when migration from M2 to RC1 or higher.

4.2.1. Namespaces

Note: If you previously installed Spring .xsd files to your VS.NET installation directory, remove them manually, and copy over the new ones, which have the -1.1.xsd suffix.

The names of the section handlers to register custom schemas has changed, from ConfigParsersSectionHandler to NamespaceParsersSectionHandler.

The target namespaces have changed, the 'directory' named /schema/ has been removed. For example, the target schema changed from <http://www.springframework.net/schema/tx> to <http://www.springframework.net/tx>.

A typical declaration to use custom schemas within your configuration file looks like this

```
<objects xmlns='http://www.springframework.net'
  xmlns:db="http://www.springframework.net/database"
  xmlns:tx="http://www.springframework.net/tx"
  xmlns:aop="http://www.springframework.net/aop">
```

The class XmlParserRegistry was renamed to NamespaceParserRegistry.

Renamed `Spring.Validation.ValidationConfigParser` to `Spring.Validation.Config.ValidationNamespaceParser`

Renamed from `DatabaseConfigParser` to `DatabaseNamespaceParser`

Renamed/Moved `Remoting.RemotingConfigParser` to `Remoting.Config.RemotingNamespaceParser`

A typical registration of custom parsers within your configuration file looks like this

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>
```

```
<spring>
  <parsers>
    <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
    <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
    <parser type="Spring.Transaction.Config.TxNamespaceParser, Spring.Data" />
  </parsers>
</spring>
```

A manual registration would look like this

```
NamespaceParserRegistry.RegisterParser(typeof(AopNamespaceParser));
NamespaceParserRegistry.RegisterParser(typeof(DatabaseNamespaceParser));
NamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParser));
```

4.2.2. Core

Moved Spring.Util.DynamicReflection to Spring.Reflection.Dynamic

Moved TypeRegistry and related classes from Spring.Context.Support to Spring.Core.TypeResolution

Moved Spring.Objects.TypeConverters to Spring.Core.TypeConvesion

4.2.3. Web

Moved Spring.Web.Validation to Spring.Web.UI.Validation

4.2.4. Data

Changed schema to use 'provider' instead of 'dbProvider' element, usage is now <db:provider ... /> and not <db:dbProvider .../>

Moved TransactionTemplate, TransactionDelegate and ITransactionCallback from Spring.Data to Spring.Data.Support

Moved AdoTemplate, AdoAccessor, AdoDaoSupport, RowMapperResultSetExtractor from Spring.Data to Spring.Data.Core

Moved AdoPlatformTransactionManager, ServiceDomainPlatformTransactionManager, and TxScopeTransactionManager from Spring.Data to Spring.Data.Core

Part I. Core Technologies

This initial part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container. A thorough treatment of the Spring Framework's IoC container is closely followed by comprehensive coverage of Spring's Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in enterprise programming.

The core functionality also includes an expression language for lightweight scripting and a ui-agnostic validation framework.

Finally, the adoption of the test-driven-development (TDD) approach to software development is certainly advocated by the Spring team, and so coverage of Spring's support for integration testing is covered (alongside best practices for unit testing). The Spring team have found that the correct use of IoC certainly does make both unit and integration testing easier (in that the presence of properties and appropriate constructors on classes makes them easier to wire together on a test without having to set up service locator registries and suchlike)... the chapter dedicated solely to testing will hopefully convince you of this as well.

- Chapter 5, *The IoC container*
- Chapter 6, *The IObjectWrapper and Type conversion*
- Chapter 7, *Resources*
- Chapter 8, *Threading and Concurrency Support*
- Chapter 9, *Object Pooling*
- Chapter 11, *Expression Evaluation*
- Chapter 10, *Spring.NET miscellanea*
- Chapter 12, *Validation Framework*
- Chapter 13, *Aspect Oriented Programming with Spring.NET*
- Chapter 14, *Aspect Library*
- Chapter 15, *Common Logging*
- Chapter 16, *Testing*

Chapter 5. The IoC container

5.1. Introduction

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC)¹ principle

The `Spring.Core` assembly is the basis for Spring.NET's IoC container. The `IObjectFactory` [<http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Objects.Factory.IObjectFactory.html>] interface provides an advanced configuration mechanism capable of managing any type of object. `IApplicationContext` [<http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Context.IApplicationContext.html>] is a sub-interface of `IObjectFactory`. It adds easier integration with Spring.NET's Aspect Oriented Programming (AOP) features, message resource handling (for use in internationalization), event propagation and application layer-specific context such as `WebApplicationContext` for use in web applications.

In short, the `IObjectFactory` provides the configuration framework and basic functionality, and the `IApplicationContext` adds more enterprise-specific functionality. The `IApplicationContext` is a complete superset of the `IObjectFactory` and is used exclusively in this chapter in descriptions of Spring's IoC container.

If you are new to Spring.NET or IoC containers in general, you may want to consider starting with Chapter 35, *IoC Quickstarts*, which contains a number of introductory level examples that actually demonstrate a lot of what is described in detail below. Don't worry if you don't absorb everything at once... those examples serve only to paint a picture of how Spring.NET hangs together in really broad brushstrokes. Once you have finished with those examples, you can come back to this section which will fill in all the fine detail.

5.2. Container overview

The interface `IApplicationContext` represents the Spring IoC container and is responsible for instantiating, configuring, and assembling many of the objects in your application. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML. The configuration metadata allows you to express the objects that compose your application and the rich interdependencies between such objects.



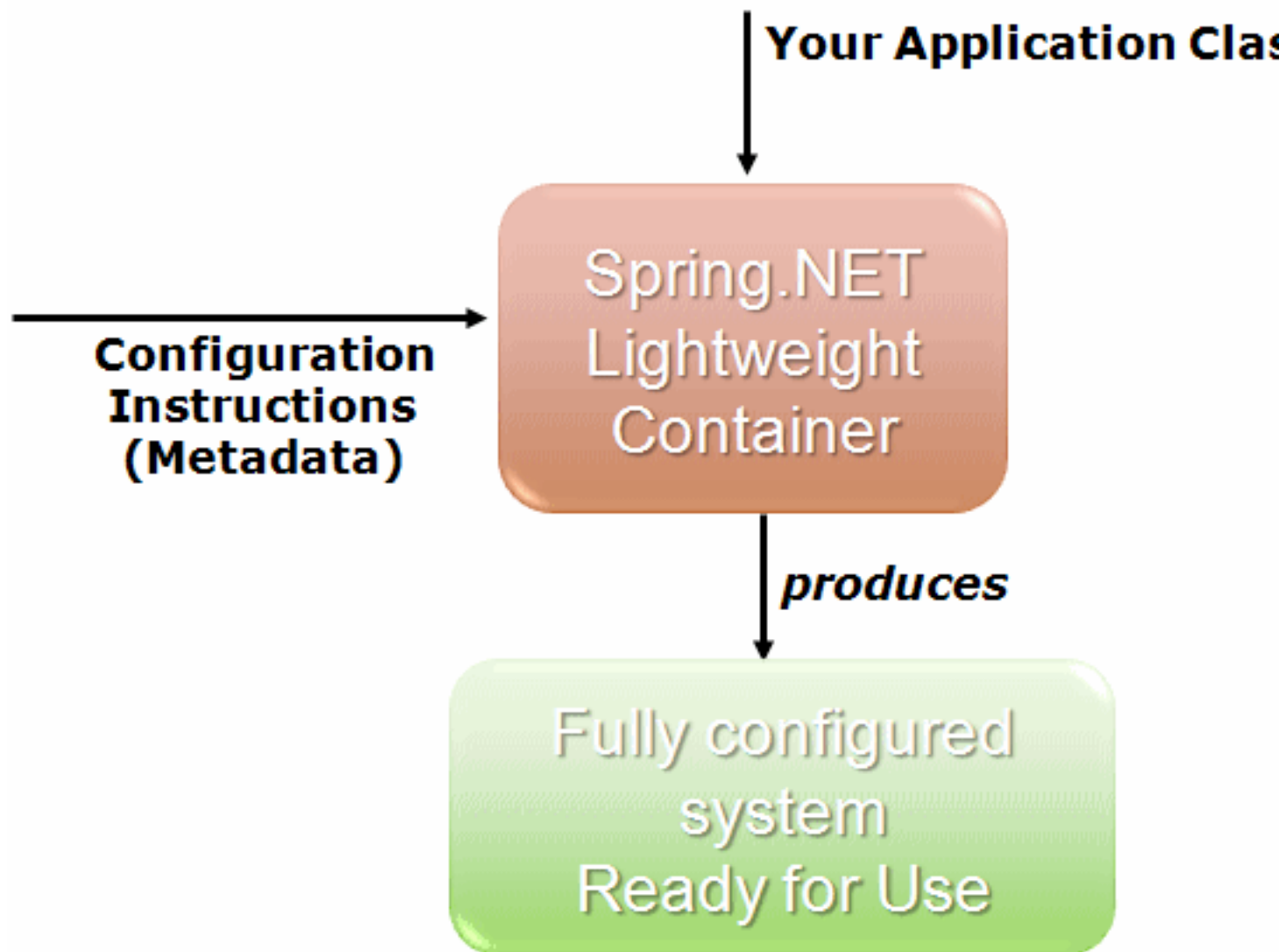
Note

Note that other ways to specify the metadata, such as attributes and .NET code, are planned for future releases, the core IoC container does not assume any specific metadata format. The Java version of Spring already supports such functionality.

Several implementations of the `IApplicationContext` interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of an `XmlApplicationContext` either programmatically or declaratively in your applications `App.config` file. In web applications Spring provides a `WebApplicationContext` implementation which is configured by adding a custom HTTP module and HTTP handler to your `Web.config` file. See the section on Web Configuration for more details.

¹See the section entitled Section 3.1, "Inversion of Control"

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the `ApplicationContext` is created and initialized, you have a fully configured and executable system or application.



5.2.1. Configuration metadata

As the preceding diagram shows, the Spring IoC container consumes a form of *configuration metadata*; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application. Configuration metadata is supplied in a simple and intuitive XML format



Note

XML-based metadata is by far the most commonly used form of configuration metadata. It is not however the only form of configuration metadata that is allowed. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. Attribute based and code based metadata will be part of an upcoming release and it is already part of the Spring Java framework.

Spring configuration consists of at least one and typically more than one object definition that the container must manage. XML-based configuration shows these objects as `<object/>` elements inside a top-level `<objects/>` element.

These object definitions correspond to the actual objects that make up your application. Typically you define service layer objects, data access objects (DAOs), presentation objects such as ASP.NET page instances, infrastructure objects such as NHibernate `SessionFactory`s, and so forth. Typically one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create/load domain objects.

The following example shows the basic structure of XML-based configuration metadata:

```
<objects xmlns="http://www.springframework.net">

  <object id="..." type="...">
    <!-- collaborators and configuration for this object go here -->
  </object>

  <object id="..." type="...">
    <!-- collaborators and configuration for this object go here -->
  </object>

  <!-- more object definitions go here -->

</objects>
```

The `id` attribute is a string that you use to identify the individual object definition. The `type` attribute defines the type of the object and uses the fully qualified type name, including the assembly name. The value of the `id` attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example; Dependencies for more information.

Spring.NET comes with an XSD schema to make the validation of the XML object definitions a whole lot easier. The XSD document is thoroughly documented so feel free to take a peek inside (see Appendix D, *Spring.NET's spring-objects.xsd*). The XSD is currently used in the implementation code to validate the XML document. The XSD schema serves a dual purpose in that it also facilitates the editing of XML object definitions inside an XSD aware editor (typically Visual Studio) by providing validation (and Intellisense support in the case of Visual Studio). You may wish to refer to Chapter 34, *Visual Studio.NET Integration* for more information regarding such integration.

5.2.2. Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an `IApplicationContext` constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, embedded assembly resources, and so on.

```
IApplicationContext context = new XmlApplicationContext("services.xml", "data-access.xml");
```

The following example shows the service layer objects (`services.xml`) configuration file.

```
<objects xmlns="http://www.springframework.net">

  <object id="PetStore" type="PetStore.Services.PetStoreService, PetStore">
    <property name="AccountDao" ref="AccountDao"/>
    <property name="ItemDao" ref="ItemDao"/>
    <!-- additional collaborators and configuration for this object go here -->
  </object>

  <!-- more object definitions for services go here -->

</objects>
```

The following example shows the data access objects (`daos.xml`) configuration file:

```
<objects xmlns="http://www.springframework.net">

  <object id="AccountDao" type="Petstore.Dao.HibernateAccountDao, PetStore">
    <!-- additional collaborators and configuration for this object go here -->
  </object>

  <object id="ItemDao" type="Petstore.Dao.HibernateItemDao, PetStore">
    <!-- additional collaborators and configuration for this object go here -->
  </object>

  <!-- more object definitions for data access objects go here -->
</objects>
```

In the preceding example, the service layer consists of the class `PetStoreService`, and two data access objects of the type `HibernateAccountDao` and `HibernateItemDao` are based on the NHibernate Object/Relational mapping framework. The property name element refers to the name of the class's property, and the ref element refers to the name of another object definition. This linkage between id and ref elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see [Dependencies](#).

5.2.2.1. Loading configuration metadata from non-default resource locations

In the previous example the configuration resources are assumed to be located in the bin\Debug directory. You can use Spring's `IResource` [\[http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Core.IO.IResource.html\]](http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Core.IO.IResource.html) abstraction to load resources from other locations.

The following example shows how to create an IoC container referring to resources located in the root directory of the filesystem as an embedded assembly resource.

```
IApplicationContext context = new XmlApplicationContext(
    "file:///services.xml",
    "assembly:///MyAssembly/MyDataAccess/data-access.xml");
```

The above example uses Spring.NET's `IResource` [\[http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Core.IO.IResource.html\]](http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Core.IO.IResource.html) abstraction. The `IResource` interface provides a simple and uniform interface to a wide array of IO resources that can represent themselves as `System.IO.Stream`.



Note

After you learn about Spring's IoC container, you may want to know more about Spring's `IResource` [\[http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Core.IO.IResource.html\]](http://www.springframework.net/doc-latest/api/net-2.0/html/Spring.Core~Spring.Core.IO.IResource.html) abstraction to load metadata from other locations as described below and also in the chapter *Chapter 7, Resources*.

These resources are most frequently files or URLs but can also be resources that have been embedded inside a .NET assembly. A simple URI syntax is used to describe the location of the resource, which follows the standard conventions for files, i.e. `file:///services.xml` and other well known protocols such as `http`.

The following snippet shows the use of the URI syntax for referring to a resource that has been embedded inside a .NET assembly, `assembly://<AssemblyName>/<NameSpace>/<ResourceName>`. The `IResource` abstraction is explained further in Section 7.1, "Introduction".



Note

To create an embedded resource using Visual Studio you must set the Build Action of the .xml configuration file to Embedded Resource in the file property editor. Also, you will need to explicitly rebuild the project containing the configuration file if it is the only change you make between

successive builds. If using NAnt to build, add a `<resources>` section to the csc task. For example usage, look at the `Spring.Core.Tests.build` file included the distribution.

5.2.2.2. Declarative configuration of the container in `App.config/Web.config`

You can also create a container by using a custom configuration section in the standard .NET application configuration file (one of `App.config` or `Web.config`). A custom configuration section that creates the same `IApplicationContext` as the previous example is

```
<spring>
  <context>
    <resource uri="file://services.xml"/>
    <resource uri="assembly://MyAssembly/MyDataAccess/data-access.xml"/>
  </context>
</spring>
```

The context type (specified as the value of the `type` attribute of the `context` element) is optional. In a standalone application the context type defaults to the `Spring.Context.Support.XmlApplicationContext` class and in a Web application defaults to `WebApplicationContext`. An example of explicitly configuring the context type The following example shows explicit use of the context type attribute:

```
<spring>
  <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
    <resource uri="file://services.xml"/>
    <resource uri="assembly://MyAssembly/MyDataAccess/data-access.xml"/>
  </context>
</spring>
```

To acquire a reference to an `IApplicationContext` using a custom configuration section, one simply uses the following code;

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

The `ContextRegistry` is used to both instantiate the application context and to perform service locator style access to other objects. (See Section 5.15, “Service Locator access” for more information). The glue that makes this possible is an implementation of the Base Class Library (BCL) provided `IConfigurationSectionHandler` interface, namely the `Spring.Context.Support.ContextHandler` class. The handler class needs to be registered in the `configSections` section of the .NET configuration file as shown below.

```
<configSections>
  <sectionGroup name="spring">
    <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
  </sectionGroup>
</configSections>
```

This declaration now enables the use of a custom context section starting at the `spring` root element.

In some usage scenarios, user code will not have to explicitly instantiate an appropriate implementation `IApplicationContext` interface, since Spring.NET code will do it for you. For example, the ASP.NET web layer provides support code to load a Spring.NET `WebApplicationContext` automatically as part of the normal startup process of an ASP.NET web application. As such, once the container has been created for you, it is often the case that you will never need to explicitly interact with it again in your code, for example when configuring ASP.NET pages.

Spring.NET comes with an XSD schema to make the validation of the XML object definitions a whole lot easier. The XSD document is thoroughly documented so feel free to take a peek inside (see Appendix D, *Spring.NET's spring-objects.xsd*). The XSD is currently used in the implementation code to validate the XML document. The XSD schema serves a dual purpose in that it also facilitates the editing of XML object definitions inside an XSD aware editor (typically Visual Studio) by providing validation (and Intellisense support in the case of Visual

Studio). You may wish to refer to Chapter 34, *Visual Studio.NET Integration* for more information regarding such integration.

Your XML object definitions can also be defined within the standard .NET application configuration file by registering the `Spring.Context.Support.DefaultSectionHandler` class as the configuration section handler for inline object definitions. This allows you to completely configure one or more `IApplicationContext` instances within a single standard .NET application configuration file as shown in the following example.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>

  <spring>

    <context>
      <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
      ...
    </objects>

  </spring>

</configuration>
```

Other options available to structure the configuration files are described in Section 5.12.1, “Context Hierarchies” and Section 5.2.2.3, “Composing XML-based configuration metadata”.

The `IApplicationContext` can be configured to register other resource handlers, custom parsers to integrate user-contributed XML schema into the object definitions section, type converters, and define type aliases. These features are discussed in section Section 5.11, “Configuration of `IApplicationContext`”

5.2.2.3. Composing XML-based configuration metadata

It can be useful to have object definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the `IApplicationContext` constructor to load object definitions from all these XML fragments. This constructor takes multiple `IResource` locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<import/>` element to load object definitions from another file (or files). For example:

```
<objects xmlns="http://www.springframework.net">

  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>

  <object id="object1" type="..." />
  <object id="object2" type="..." />

</objects>
```

In the preceeding example, external object definitions are being loaded from three files, `services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are relative to the definition file doing the importing, so `services.xml` must be in the same directory as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is ignored, but given that these paths are relative, it is better form not to use

the slash at all. The contents of the files being imported, including the top level `<objects/>` element, must be valid XML object definitions according to the Spring Schema.

5.2.3. Using the container

An `IApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different objects and their dependencies. Using the method `GetObject(string)` or the indexer `[string]` you can retrieve instances of your objects.

The `IApplicationContext` enables you to read object definitions and access them as follows:

```
// create and configure objects
IApplicationContext context = new XmlApplicationContext("services.xml", "daos.xml");

// retrieve configured instance
PetStoreService service = (PetStoreService) context.GetObject("PetStoreService");

// use configured instance
IList userList = service.GetUserNames();
```

You use the method `GetObject` to retrieve instances of your objects. The `IApplicationContext` interface has a few other methods for retrieving objects, but ideally your application code should never use them. Indeed, your application code should have no calls to the `GetObject` method at all, and thus no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides for dependency injection for various web framework classes such as ASP.NET pages and user controls.



Note

The syntactical inconvenience of the cast will be addressed in a future release of Spring.NET that is based on a generic API. Note, that even when using a generic API, looking up an object by name in no way guarantees that the return type will be that of the generic type.

5.2.4. Object definition overview

A Spring IoC container manages one or more objects. These objects are created with the configuration metadata that you supply to the container.

Within the container itself, these object definitions are represented as `IObjectDefinition` objects, which contain (among other information) the following metadata:

- *A type name*: typically the actual implementation class of the object being defined..
- Object behavioral configuration elements, which state how the object should behave in the container (i.e. prototype or singleton, lifecycle callbacks, and so forth)
- References to other objects which are needed for the object to do its work: these references are also called *collaborators* or *dependencies*.
- Other configuration settings to set in the newly created object. An example would be the number of threads to use in an object that manages a worker thread pool, or the size limit of the pool.

This metadata translates to a set of properties that make up each object definition. The following table lists some of these properties, with links to documentation

Table 5.1. Object definition explanation

Property	More info
type	Section 5.2.5, “Instantiating objects”

Property	More info
id and name	Section 5.2.4.1, “Naming objects”
singleton or prototype	Section 5.4, “Object Scopes”
object properties	Section 5.3.1, “Dependency injection”
constructor arguments	Section 5.3.1, “Dependency injection”
autowiring mode	Section 5.3.6, “Autowiring collaborators”
dependency checking mode	Section 5.3.7, “Checking for dependencies”
initialization method	Section 5.6.1, “Lifecycle interfaces”
destruction method	Section 5.6.1, “Lifecycle interfaces”

In addition to object definitions which contain information on how to create a specific object, the [IApplicationContext](#) implementations also permit the registration of existing objects that are created outside the container, by users. This is done by accessing the [ApplicationContext's IObjectFactory](#) via the property `ObjectFactory` which returns the [IObjectFactory](#) implementation [DefaultListableObjectFactory](#). [DefaultListableObjectFactory](#) supports registration through the methods `registerSingleton(...)` and `registerObjectDefinition(...)`. However, typical applications work solely with objects defined through metadata object definitions.

5.2.4.1. Naming objects

Every object has one or more identifiers. These identifiers must be unique within the container that hosts the objects. An object usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

A convention that has evolved is to use the standard C# convention for Property names when naming objects. That is, object names start with an uppercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) 'AccountManager', 'AccountService', 'UserDao', 'LoginController', and so forth.

Naming objects consistently makes your configuration easier to read and understand, and if you are using Spring AOP it helps a lot when applying advice to a set of objects related by name.

When using XML-based configuration metadata, you use the `'id'` and/or `'name'` attributes to specify the object identifier(s). The `'id'` attribute allows you to specify exactly one id, and because it is a real XML element ID attribute, the XML parser is able to do some extra validation when other elements reference the id. As such, it is the preferred way to specify an object id. However, the XML specification does limit the characters which are legal in XML IDs. This is usually not a constraint, but if you have a need to use one of these special XML characters, or want to introduce other aliases to the object, you can specify them in the `'name'` attribute, separated by a comma (,), semicolon (;), or whitespace.



Note

You are not required to supply a name or id for an object. If no name or id is supplied explicitly, the container will generate a unique name for that object. However, if you want to refer to that object

by name, through the use of the `ref` element or Service Location style lookup, you must provide a name. The motivations for not supplying a name for an object are to use autowiring and inline-objects which will be discussed later.

5.2.4.1.1. Aliasing an object outside the object definition

In an object definition itself, you may supply more than one name for the object, by using a combination of up to one name specified by the `id` attribute, and any number of other names in the `name` attribute. These names are equivalent aliases to the same object, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using an object name that is specific to that component itself.

Specifying all aliases where the object is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for an object that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the `<alias/>` element to accomplish this.

```
<alias name="fromName" alias="toName"/>
```

In this case, an object in the same container which is named `fromName`, may also after the use of this alias definition, be referred to as `toName`.

For example, the configuration metadata for subsystem A may refer to a `DbProvider` via the name 'SubsystemA-DbProvider'. The configuration metadata for subsystem B may refer to a `DbProvider` via the name 'SubsystemB-DbProvider'. When composing the main application that uses both these subsystems the main application refers to the `DbProvider` via the name 'MyApp-DbProvider'. To have all three names refer to the same object you add to the `MyApp` configuration metadata the following aliases definitions:

```
<alias name="SubsystemA-DbProvider" alias="SubsystemB-DbProvider"/>
<alias name="SubsystemA-DbProvider" alias="MyApp-DbProvider"/>
```

Now each component and the main app can refer to the connection through a name that is unique and guaranteed not to clash with any other definition (effectively there is a namespace), yet they refer to the same object.

5.2.5. Instantiating objects

An object definition essentially is a recipe for creating one or more objects. The container looks at the recipe for a named object when asked, and uses the configuration metadata encapsulated by that object definition to create (or acquire) an actual object.

If you are using XML-based configuration metadata, you can specify the type of object that is to be instantiated in the `'type'` attribute of the `<object/>` element. This `'type'` attribute (which internally is a `Type` property on a `IOBJECTDefinition` instance) is usually mandatory. (For exceptions see the section called *Instantiation using an instance factory method and Object definition inheritance*.) You use the `Type` property in one of two ways:

- Typically, to specify the type of the object to be constructed in the case where the container itself directly creates the object by calling its constructor reflectively, somewhat equivalent to C# code using the `'new'` operator.
- To specify the actual class containing the `static` factory method that will be invoked to create the object, in the less common case where the container invokes a `static factory` method on a class to create the object. The object type returned from the invocation of the `static` factory method may be the same type or another type entirely.

5.2.5.1. Instantiation with a constructor

When you create an object using the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the object type should be sufficient. However, depending on what type of IoC you are going to use for that specific object, you may need to create a default constructor.

With XML-based configuration metadata you can specify your object class as follows:

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary"/>
```

For details about the mechanism for supplying arguments to the constructor (if required), and setting object instance properties after the object is constructed, see Section 5.3.1, “Dependency injection”.

This XML fragment describes an object definition that will be identified by the *exampleObject* name, instances of which will be of the `Examples.ExampleObject` type that has been compiled into the `ExamplesLibrary` assembly. Take special note of the structure of the `type` attribute's value... the namespace-qualified name of the class is specified, followed by a comma, followed by (at a bare minimum) the name of the assembly that contains the class. In the preceding example, the `ExampleObject` class is defined in the `Examples` namespace, and it has been compiled into the `ExamplesLibrary` assembly.

The name of the assembly that contains the type *must* be specified in the `type` attribute. Furthermore, it is recommended that you specify the fully qualified assembly name² in order to guarantee that the type that Spring.NET uses to instantiate your object (s) is indeed the one that you expect. Usually this is only an issue if you are using classes from (strongly named) assemblies that have been installed into the Global Assembly Cache (GAC).

If you have defined nested classes use the addition symbol, `+`, to reference the nested class. For example, if the class `Examples.ExampleObject` had a nested class `Person` the XML declaration would be

```
<object id="exampleObject" type="Examples.ExampleObject+Person, ExamplesLibrary"/>
```

If you are defining classes that have been compiled into assemblies that are available to your application (such as the `bin` directory in the case of ASP.NET applications) via the standard assembly probing mechanisms, then you can specify simply the name of the assembly (e.g. `ExamplesLibrary.Data`)... this way, when (or if) the assemblies used by your application are updated, you won't have to change the value of every `<object/>` definition's `type` attribute to reflect the new version number (if the version number has changed)... Spring.NET will automatically locate and use the newer versions of your assemblies (and their attendant classes) from that point forward.

5.2.5.2. Instantiation with a static factory method

When defining an object which is to be created using a static factory method, you use the `type` attribute to specify the type containing the static factory method and an attribute named `factory-method` to specify the name of the factory method itself. You should be able to call this method (with an optional list of arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such an object definition is to call static factories in legacy code.

The following object definition specifies that the object will be created by calling a factory-method. The definition does not specify the type of the returned object, only the type containing the factory method. In this example, `CreateInstance` must be a static method.

²More information about assembly names can be found in the *Assembly Names* section of the .NET Framework Developer's Guide (installed as part of the .NET SDK), or online at Microsoft's MSDN website, by searching for *Assembly Names*.

```
<object id="exampleObject"
      type="Examples.ExampleObjectFactory, ExamplesLibrary"
      factory-method="CreateInstance"/>
```

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after it has been returned from the factory, see Section 5.3.2, “Dependencies and configuration in detail”

5.2.5.3. Object creation via an instance factory method

Similar to instantiation through a static factory method, instantiation with an instance factory method invokes a non-static method on an existing object from the container to create a new object. To use this mechanism, leave the `type` attribute empty, and in the `factory-object` attribute specify the name of an object in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the `factory-method` attribute.

```
<!-- the factory object, which contains an instance method called 'CreateInstance' -->
<object id="exampleFactory" type="...">
  <!-- inject any dependencies required by this object -->
</object>

<!-- the object that is to be created by the factory object -->
<object id="exampleObject"
      factory-method="CreateInstance"
      factory-object="exampleFactory"/>
```

This approach shows that the factory object itself can be managed and configured through dependency injection (DI). See Dependencies and configuration in detail.



Note

In Spring documentation, '*factory object*', refers to an object that is configured in the Spring container that will create objects via an instance or static factory method. By contrast, `IFactoryObject` (notice the capitalization) refers to a Spring-specific `IFactoryObject`.

5.2.6. Object creation of generic types

Generic types can also be created in much the same manner as non-generic types.

5.2.6.1. Object creation of generic types via constructor invocation

The following examples show the definition of simple generic types and how they can be created in Spring's XML based configuration file.

```
namespace GenericsPlay
{
    public class FilterableList<T>
    {
        private List<T> list;

        private String name;

        public List<T> Contents
        {
            get { return list; }
            set { list = value; }
        }

        public String Name
        {
```

```

        get { return name; }
        set { name = value; }
    }

    public List<T> ApplyFilter(string filterExpression)
    {
        /// should really apply filter to list ;)
        return new List<T>();
    }
}

```

The XML configuration to create and configure this object is shown below

```

<object id="myFilteredIntList" type="GenericsPlay.FilterableList<int>, GenericsPlay">
  <property name="Name" value="My Integer List"/>
</object>

```

There are a few items to note in terms how to specify a generic type. First, the left bracket that specifies the generic type, i.e. <, is replaced with the string < due to XML escape syntax for the less than symbol. Yes, we all realize this is less than ideal from the readability point of view. Second, the generic type arguments can not be fully assembly qualified as the comma is used to separate generic type arguments. Alternative characters used to overcome the two quirks can be implemented in the future but so far, all proposals don't seem to help clarify the text. The suggested solution to improve readability is to use type aliases as shown below

```

<typeAliases>
  <alias name="GenericDictionary" type=" System.Collections.Generic.Dictionary<,>" />
  <alias name="myDictionary" type="System.Collections.Generic.Dictionary<int,string>" />
</typeAliases>

```

So that instead of something like this

```

<object id="myGenericObject"
  type="GenericsPlay.ExampleGenericObject<System.Collections.Generic.Dictionary<int , string>>, GenericsPlay" />

```

It can be shortened to

```

<object id="myOtherGenericObject"
  type="GenericsPlay.ExampleGenericObject<GenericDictionary<int , string>>, GenericsPlay" />

```

or even shorter

```

<object id="myOtherOtherGenericObject"
  type="GenericsPlay.ExampleGenericObject<MyIntStringDictionary>, GenericsPlay" />

```

Refer to Section 5.11, “Configuration of IApplicationContext” for additional information on using type aliases.

5.2.6.2. Object creation of generic types via static factory method

The following classes are used to demonstrate the ability to create instances of generic types that themselves are created via a static generic factory method.

```

public class TestGenericObject<T, U>
{
    public TestGenericObject()
    {
    }

    private IList<T> someGenericList = new List<T>();

    private IDictionary<string, U> someStringKeyedDictionary =
        new Dictionary<string, U>();

    public IList<T> SomeGenericList
    {

```

```

        get { return someGenericList; }
        set { someGenericList = value; }
    }

    public IDictionary<string, U> SomeStringKeyedDictionary
    {
        get { return someStringKeyedDictionary; }
        set { someStringKeyedDictionary = value; }
    }
}

```

The accompanying factory class is

```

public class TestGenericObjectFactory
{
    public static TestGenericObject<V, W> StaticCreateInstance<V, W>()
    {
        return new TestGenericObject<V, W>();
    }

    public TestGenericObject<V, W> CreateInstance<V, W>()
    {
        return new TestGenericObject<V, W>();
    }
}

```

The XML snippet to create an instance of TestGenericObject where v is a List of integers and w is an integer is shown below

```

<object id="myTestGenericObject"
        type="GenericsPlay.TestGenericObjectFactory, GenericsPlay"
        factory-method="StaticCreateInstance<System.Collections.Generic.List<int>,int>"
/>

```

The StaticCreateInstance method is responsible for instantiating the object that will be associated with the id 'myTestGenericObject'.

5.2.6.3. Object creation of generic types via instance factory method

Using the class from the previous example the XML snippet to create an instance of a generic type via an instance factory method is shown below

```

<object id="exampleFactory" type="GenericsPlay.TestGenericObject<int,string>, GenericsPlay"/>

<object id="anotherTestGenericObject"
        factory-object="exampleFactory"
        factory-method="CreateInstance<System.Collections.Generic.List<int>,int>"/>

```

This creates an instance of TestGenericObject<List<int>,int>

5.3. Dependencies

A typical enterprise application does not consist of a single object. Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of object definitions that stand-alone to a fully realized application where objects collaborate to achieve a goal.

5.3.1. Dependency injection

Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments and properties that are set on the object instance after it

is constructed. (Factory methods may be considered a special case of providing constructor arguments for the purposes of this description). The container injects these dependencies when it creates the object. This process is fundamentally the inverse to the case when the object itself is controlling the instantiation or location of its dependencies by using direct construction of classes, or the Service Locator pattern. The inverting of this responsibility is why the name Inversion of Control (IoC) is used to describe the container's actions.

Code is cleaner when using DI and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies. Long sections of initialization code that you used to hide in a `#region` tag simply go away, and are placed by container configuration metadata. One can also consider this clean up an application of the principal of Separation of Concerns. Before using DI, your class was responsible for business logic AND its configuration, it was concerned with doing more than one thing. DI removes the responsibility of configuration from the class, leaving it only with a single purpose, as the location of business logic. Furthermore, since your class does not know the location of its dependencies these classes also become easier to test, in particular when the dependencies are interfaces or abstract base classes allowing for stub or mock implementation to be used in unit tests.

Dependency injection exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

5.3.1.1. Constructor-based dependency injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a static factory method with specific arguments to construct the object is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection. Notice that there is nothing special about this class (no container specific interfaces, base classes or attributes)

```
public class SimpleMovieLister
{
    // the SimpleMovieLister has a dependency on a MovieFinder
    private IMovieFinder movieFinder;

    // a constructor so that the Spring container can 'inject' a MovieFinder
    public MovieLister(IMovieFinder movieFinder)
    {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected IMovieFinder is omitted...
}
```

5.3.1.1.1. Constructor argument resolution

Constructor argument resolution matching occurs using the argument's type. If ambiguity exists in the constructor arguments of a object definition, then the order in which the constructor arguments are defined in a object definition is the order in which those arguments are supplied to the appropriate constructor when the object being instantiated. Consider the following class:

```
namespace X.Y
{
    public class Foo
    {
        public Foo(Bar bar, Baz baz)
        {
            // ...
        }
    }
}
```

No potential ambiguity exists, assuming of course that Bar and Baz classes are not related by inheritance. Thus the following configuration will work just fine, and you do not need to specify the constructor argument indexes and / or types explicitly in the `<constructor-arg>` element.

```
<object id="foo" type="X.Y.Foo, Example">
  <constructor-arg ref="bar"/>
  <constructor-arg ref="baz"/>
</object>

<object id="bar" type="X.Y.Bar, Example"/>
<object id="baz" type="X.Y.Baz, Example"/>
```

When another object is referenced, the type is known, and matching can occur (as was the case with the preceding example).

When a simple type is used, such as `<value>true</value>`, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

```
using System;

namespace SimpleApp
{
    public class ExampleObject
    {
        private int years;           //No. of years to the calculate the Ultimate Answer

        private string ultimateAnswer; //The Answer to Life, the Universe, and Everything

        public ExampleObject(int years, string ultimateAnswer)
        {
            this.years = years;
            this.ultimateAnswer = ultimateAnswer;
        }
    }
}
```

5.3.1.1.1. Constructor argument type matching

In the preceding scenario, the container *can* use type matching with simple types by explicitly specifying the type of the constructor argument using the `'type'` attribute. For example:

```
<object name="exampleObject" type="SimpleApp.ExampleObject, SimpleApp">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="string" value="42"/>
</object>
```

The type attribute specifies the `System.Type` of the constructor argument, such as `System.Int32`. Alias' are available to for common simple types (and their array equivalents). These alias' are...

Table 5.2. Type aliases

Type	Alias'	Array Alias'
System.Char	char, Char	char[], Char()
System.Int16	short, Short	short[], Short()
System.Int32	int, Integer	int[], Integer()
System.Int64	long, Long	long[], Long()
System.UInt16	ushort	ushort[]
System.UInt32	uint	uint[]

Type	Alias'	Array Alias'
System.UInt64	ulong	ulong[]
System.Float	float, Single	float[], Single()
System.Double	double, Double	double[], Double()
System.Date	date, Date	date[], Date()
System.Decimal	decimal, Decimal	decimal[], Decimal()
System.Boolean	bool, Boolean	bool[], Boolean()
System.String	string, String	string[], String()

5.3.1.1.1.2. Constructor argument Index

Use the `index` attribute to specify explicitly the index of constructor arguments. For example:

```
<object name="exampleObject" type="SimpleApp.ExampleObject, SimpleApp">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</object>
```

In addition to resolving the ambiguity of multiple simple values, specifying an index also resolves ambiguity where a constructor has two arguments of the same type. Note that the *index is 0 based*.

5.3.1.1.1.3. Constructor arguments by name

You can specify constructor argument by name using `name` attribute of the `<constructor-arg>` element.

```
<object name="exampleObject" type="SimpleApp.ExampleObject, SimpleApp">
  <constructor-arg name="years" value="7500000"/>
  <constructor-arg name="ultimateAnswer" value="42"/>
</object>
```

5.3.1.2. Setter-based dependency injection

Setter-based DI is accomplished by the container invoking setter properties on your objects after invoking a no-argument constructor or no-argument static factory method to instantiate your object.

The following example shows a class that can only be dependency injected using pure setter injection.

```
public class MovieLister
{
    private IMovieFinder movieFinder;

    public IMovieFinder MovieFinder
    {
        set
        {
            movieFinder = value;
        }
    }

    // business logic that actually 'uses' the injected IMovieFinder is omitted...
}
```

Constructor-based or setter-based DI?

The Spring team generally advocates the usage of setter injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional. The presence of setter properties also makes objects of that class amenable to reconfigured or reinjection later. Management through WMI is a compelling use case.

Some purists favor constructor-based injection. Supplying all object dependencies means that the object is always returned to client (calling) code in a totally initialized state. The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

Use the DI that makes the most sense for a particular class. Sometimes, when dealing with third-party classes to which you do not have the source, the choice is made for you. A legacy class may not expose any setter methods, and so constructor injection is the only available DI.

Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

The `IApplicationContext` supports constructor- and setter-based DI for the objects it manages. It also supports setter-based DI after some dependencies have already been supplied via the constructor approach..

The configuration for the dependencies comes in the form of the `IObjectDefinition` class, which is used together with `TypeConverters` to know how to convert properties from one format to another. However, most users of Spring.NET will not be dealing with these classes directly (that is programatically), but rather with an XML definition file which will be converted internally into instances of these classes, and used to load an entire Spring IoC container instance. Refer to Section 6.3, “Type conversion” for more information regarding type conversion, and how you can design your classes to be convertible by Spring.NET.

The container resolves object dependences as:

1. The `IApplicationContext` is created and initialized with a configuration that describes all the objects. Most Spring.NET users use an `IObjectFactory` Or `IApplicationContext` variant that supports XML format configuration files.
2. Each object has dependencies expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the object, *when the object is actually created*.
3. Each property or constructor argument is either an actual definition of the value to set, or a reference to another object in the container.
4. Each property or constructor argument which is a value must be able to be converted from whatever format it was specified in, to the actual `System.Type` of that property or constructor argument. By default Spring.NET can convert a value supplied in string format to all built-in types, such as `int`, `long`, `string`, `bool`, etc.

The Spring container validates the configuration of each object as the container is created, including the validation of whether object reference properties refer to valid object. However, the object properties themselves are not set until the object is actually created. Objects that are defined as singletons and set to be pre-instantiated, are created when the container is created. Otherwise, the object is created only when it is requested. Creation of an object potentially causes a graph of objects to be created as the objects dependencies and its dependencies' dependencies (and so on) are created and assigned.

Circular Dependencies

If you are using predominantly constructor injection it is possible to create unresolvable circular dependency scenario.

For example: Class A, which requires an instance of class B to be provided via constructor injection, and class B, which requires an instance of class A to be provided via constructor injection. If you configure objects for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throw a `ObjectCurrentlyInCreationException`.

One possible solution to this issue is to edit the source code of some of your classes to be configured via setters instead of via constructors. Alternatively, avoid constructor injection and stick to setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between object A and object B will force one of the objects to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

You can generally trust Spring.NET to do the right thing. It detects configuration problems, such as references to non-existent object definitions and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, which is when the object is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies. For example, the object throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why `IApplicationContext` by default pre-instantiate singleton objects. At the cost of some upfront time and memory to create these objects before they are actually needed, you discover configuration issues when the `IApplicationContext` is created, not later. If you wish, you can still override this default behavior and set any of these singleton objects will lazy-initialize, rather than be pre-instantiated.

If no circular dependencies exist, when one or more collaborating objects are being injected into a dependent object, each collaborating object is *totally* configured prior to being passed into the dependent object. This means that if object A has a dependency on object B, the Spring IoC container completely configures object B prior to invoking the setter method on object A. In other words, the object is instantiated (if not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a configured init method or the `IInitializingObject` callback method) will all be invoked.

5.3.1.3. Examples of dependency injection

First, an example of using XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifying some object definitions:

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary">

  <!-- setter injection using the ref attribute -->
  <property name="objectOne" ref="anotherExampleObject"/>
  <property name="objectTwo" ref="yetAnotherObject"/>
  <property name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>
```

```
[C#]
public class ExampleObject
{
```

```

private AnotherObject objectOne;
private YetAnotherObject objectTwo;
private int i;

public AnotherObject ObjectOne
{
    set { this.objectOne = value; }
}

public YetAnotherObject ObjectTwo
{
    set { this.objectTwo = value; }
}

public int IntegerProperty
{
    set { this.i = value; }
}
}

```

In the preceding example, setters have been declared to match against the properties specified in the XML file. Find below an example of using constructor-based DI.

```

<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <constructor-arg name="objectTwo" ref="yetAnotherObject"/>
    <constructor-arg name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>

```

```

[Visual Basic.NET]
Public Class ExampleObject

    Private myObjectOne As AnotherObject
    Private myObjectTwo As YetAnotherObject
    Private i As Integer

    Public Sub New (
        anotherObject As AnotherObject,
        yetAnotherObject As YetAnotherObject,
        i As Integer)

        myObjectOne = anotherObject
        myObjectTwo = yetAnotherObject
        Me.i = i
    End Sub
End Class

```

The constructor arguments specified in the object definition will be used to pass in as arguments to the constructor of the `ExampleObject`.

Now consider a variant of this where instead of using a constructor, Spring is told to call a static factory method to return an instance of the object

```

<object id="exampleObject" type="Examples.ExampleFactoryMethodObject, ExamplesLibrary"
    factory-method="CreateInstance">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <constructor-arg name="objectTwo" ref="yetAnotherObject"/>
    <constructor-arg name="intProp" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>

```

```

[C#]
public class ExampleFactoryMethodObject
{
    private AnotherObject objectOne;

```

```

private YetAnotherObject objectTwo;
private int i;

// a private constructor
private ExampleFactoryMethodObject()
{
}

public static ExampleFactoryMethodObject CreateInstance(AnotherObject objectOne,
                                                         YetAnotherObject objectTwo,
                                                         int intProp)
{
    ExampleFactoryMethodObject fmo = new ExampleFactoryMethodObject();
    fmo.AnotherObject = objectOne;
    fmo.YetAnotherObject = objectTwo;
    fmo.IntegerProperty = intProp;
    return fmo;
}

// Property definitions
}

```

Arguments to the static factory method are supplied via `<constructor-arg/>` elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class which contains the static factory method, although in this example it is. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the `factory-object` attribute instead of the `type` attribute), so will not be detailed here.

Note that Setter Injection and Constructor Injection are not mutually exclusive. It is perfectly reasonable to use both for a single object definition, as can be seen in the following example:

```

<object id="exampleObject" type="Examples.MixedIoCObject, ExamplesLibrary">
  <constructor-arg name="objectOne" ref="anotherExampleObject"/>
  <property name="objectTwo" ref="yetAnotherObject"/>
  <property name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>

```

```

[C#]
public class MixedIoCObject
{
    private AnotherObject objectOne;
    private YetAnotherObject objectTwo;
    private int i;

    public MixedIoCObject (AnotherObject obj)
    {
        this.objectOne = obj;
    }

    public YetAnotherObject ObjectTwo
    {
        set { this.objectTwo = value; }
    }

    public int IntegerProperty
    {
        set { this.i = value; }
    }
}

```

5.3.2. Dependencies and configuration in detail

As mentioned in the previous section, you can define object properties and constructor arguments as either references to other managed objects (collaborators), or as values defined inline. Spring's XML-based

configuration metadata supports sub-element types within its `<property/>` and `<constructor-arg/>` elements for this purpose.

5.3.2.1. Straight values (primitives, strings, and so on)

The `value` attribute of the `<property/>` element specifies a property or constructor argument as a human-readable string representation. As mentioned previously, `TypeConverter` instances are used to convert these string values from a `System.String` to the actual property or argument type.

In the following example, we use a `SqlConnection` from the `System.Data.SqlClient` namespace. This class (like many other existing classes) can easily configured by Spring as it offers a convenient public property for configuration of its `ConnectionString` property.

```
<objects xmlns="http://www.springframework.net">
  <object id="myConnection" type="System.Data.SqlClient.SqlConnection">
    <!-- results in a call to the setter of the ConnectionString property -->
    <property
      name="ConnectionString"
      value="Integrated Security=SSPI;database=northwind;server=mySQLServer"/>
  </object>
</objects>
```

5.3.2.1.1. The `idref` element

An `idref` element is simply an error-proof way to pass the `id` (string value - not a reference) of another object in the container to a `<constructor-arg/>` or `<property/>` element.

```
<object id="theTargetObject" type="...">
  . . .
</object>

<object id="theClientObject" type="...">
  <property name="targetName">
    <idref object="theTargetObject"/>
  </property>
</object>
```

This above object definition snippet is *exactly* equivalent (at runtime) to the following snippet:

```
<object id="theTargetObject" type="...">
  . . .
</object>

<object id="theClientObject" type="...">
  <property name="targetName" value="theTargetObject"/>
</object>
```

The first form is preferable to the second is that using the `idref` tag allows the container to validate *at deployment time* that the referenced, named object actually exists. In the second variation, no validation is performed on the value that is passed to the `targetName` property of the client object. Typos are only discovered (with oft mikely fatal results) when the 'client' object is actually instantiated. If the 'client' object is a prototype object, this typo and the resulting exception may only be discovered long after the container is deployed.

Additionally, if the reference object is in the same XML unit, and the object name is the object `id`, you can use the `local` attribute which allows the XML parser itself to validate the object name earlier, at XML document parse time.

```
<property name="targetName">
  <idref local="theTargetObject"/>
</property>
```

5.3.2.1.2. Whitespace Handling

Usually all leading and trailing whitespaces are trimmed from a `<value />` element's text. In some cases it is necessary to maintain whitespaces exactly as they are written into the xml element. The parser does understand the `xml:space` attribute in this case:

```
<property name="myProp">
  <value xml:space="preserve"> &#x000a;&#x000d;&#x0009;</value>
</property>
```

The above configuration will result in the string `"\n\r\t"`. Note, that you don't have to explicitly specify the 'xml' namespace on top of your configuration.

5.3.2.2. References to other objects (collaborators)

The `ref` element is the final element allowed inside a `<constructor-arg/>` or `<property/>` definition element. Here you set the value of the specified property to be a reference to another object (a collaborator) managed by the container. The referenced object is a dependency of the object whose property will be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton object it may be initialized already by the container.) All references are ultimately just a reference to another object. Scoping and validation depend on whether you specify the `id`/`name` of the object through the `object`, `local`, or `parent` attributes.

Specifying the target object through the `object` attribute of the `ref` tag is the most general form, and allows creation of a reference to any object in the same container or parent container, regardless of whether it is in the same XML file. The value of the `object` attribute may be the same as the `id` attribute of the target object, or as one of the values in the `name` attribute of the target object.

```
<ref object="someObject"/>
```

Specifying the target object by using the `local` attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the `local` attribute must be the same as the `id` attribute of the target object. The XML parser will issue an error if no matching element is found in the same file. As such, using the `local` variant is the best choice (in order to know about errors as early as possible) if the target object is in the same XML file.

```
<ref local="someObject"/>
```

Specifying the target object through the `parent` attribute creates a reference to an object that is in a parent container of the current container. The value of the 'parent' attribute may be the same as either the 'id' attribute of the target object, or one of the values in the 'name' attribute of the target object, and the target object must be in a parent container to the current one. You use this object reference variant mainly when you have a hierarchy of containers and you want to wrap an existing object in a parent container with some sort of proxy which will have the same name as the parent object.

```
<!-- in the parent context -->
<object id="AccountService" type="MyApp.SimpleAccountService, MyApp">
  <!-- insert dependencies as required as here -->
</object>
```

```
<!-- in the child (descendant) context -->
<object id="AccountService" <-- notice that the name of this object is the same as the name of the 'parent' object
  type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="target">
    <ref parent="AccountService"/> <-- notice how we refer to the parent object -->
  </property>
  <!-- insert other configuration and dependencies as required as here -->
</object>
```

5.3.2.3. Inner objects

An `<object/>` element inside the `<constructor-arg/>` or `<property/>` element defines so called inner object.

```
<object id="outer" type="...">

  <!-- Instead of using a reference to target, just use an inner object -->

  <property name="target">
    <object type="ExampleApp.Person, ExampleApp">
      <property name="name" value="Tony"/>
      <property name="age" value="51"/>
    </object>
  </property>
</object>
```

An inner object definition does not require a defined id or name; the container ignores these values. It also ignores the scope flag. Inner object are *always* anonymous and they are *always* scoped as prototypes. It is not possible to inject inner objects into collaborating objects other than into the enclosing object.

5.3.2.4. Setting collection values

The list, set, name-values and dictionary elements allow properties and arguments of the type `ICollection`, `ISet`, `NameValueCollection` and `IDictionary`, respectively, to be defined and set.

```
<objects xmlns="http://www.springframework.net">
  <object id="moreComplexObject" type="Example.ComplexObject">
    <!--
    results in a call to the setter of the SomeList (System.Collections.ICollection) property
    -->
    <property name="SomeList">
      <list>
        <value>a list element followed by a reference</value>
        <ref object="myConnection"/>
      </list>
    </property>
    <!--
    results in a call to the setter of the SomeDictionary (System.Collections.IDictionary) property
    -->
    <property name="SomeDictionary">
      <dictionary>
        <entry key="a string => string entry" value="just some string"/>
        <entry key-ref="myKeyObject" value-ref="myConnection"/>
      </dictionary>
    </property>
    <!--
    results in a call to the setter of the SomeNameValue (System.Collections.NameValueCollection) property
    -->
    <property name="SomeNameValue">
      <name-values>
        <add key="HarryPotter" value="The magic property"/>
        <add key="JerrySeinfeld" value="The funny (to Americans) property"/>
      </name-values>
    </property>
    <!--
    results in a call to the setter of the SomeSet (Spring.Collections.ISet) property
    -->
    <property name="someSet">
      <set>
        <value>just some string</value>
        <ref object="myConnection"/>
      </set>
    </property>
  </object>
</objects>
```

Many classes in the BCL expose only read-only properties for collection classes. When Spring.NET encounters a read-only collection, it will configure the collection by using the getter property to obtain a reference to the

collection class and then proceed to add the additional elements to the existing collection. This results in an additive behavior for collection properties that are exposed in this manner.

The value of a Dictionary entry, or a set value, can also again be any of the following elements:

```
(object | ref | idref | expression | list | set | dictionary |
  name-values | value | null)
```

The shortcut forms for value and references are useful to reduce XML verbosity when setting collection properties. See Section 5.3.2.9, “Value and ref shortcut forms” for more information.

5.3.2.5. Setting generic collection values

Spring supports setting values for classes that expose properties based on the generic collection interfaces `ICollection<T>` and `IDictionary<TKey, TValue>`. The type parameter for these collections is specified by using the XML attribute `element-type` for `ICollection<T>` and the XML attributes `key-type` and `value-type` for `IDictionary<TKey, TValue>`. The values of the collection are automatically converted from a string to the appropriate type. If you are using your own user-defined type as a generic type parameter you will likely need to register a custom type converter. Refer to Section 5.5, “Type conversion” for more information. The implementations of `ICollection<T>` and `IDictionary<TKey, TValue>` that is created are `System.Collections.Generic.List` and `System.Collections.Generic.Dictionary`.

The following class represents a lottery ticket and demonstrates how to set the values of a generic `ICollection`.

```
public class LotteryTicket {

    List<int> list;

    DateTime date;

    public List<int> Numbers {
        set { list = value; }
        get { return list; }
    }

    public DateTime Date {
        get { return date; }
        set { date = value; }
    }
}
```

The XML fragment that can be used to configure this class is shown below

```
<object id="MyLotteryTicket" type="GenericsPlay.Lottery.LotteryTicket, GenericsPlay">
  <property name="Numbers">
    <list element-type="int">
      <value>11</value>
      <value>21</value>
      <value>23</value>
      <value>34</value>
      <value>36</value>
      <value>38</value>
    </list>
  </property>
  <property name="Date" value="4/16/2006"/>
</object>
```

The following shows the definition of a more complex class that demonstrates the use of generics using the `Spring.Expressions.IExpression` interface as the generic type parameter for the `ICollection` element-type and the value-type for `IDictionary`. `Spring.Expressions.IExpression` has an associated type converter, `Spring.Objects.TypeConverters.ExpressionConverter` that is already pre-registered with Spring.

```
public class GenericExpressionHolder
```

```

{
    private System.Collections.Generic.IList<IExpression> expressionsList;

    private System.Collections.Generic.IDictionary<string, IExpression> expressionsDictionary;

    public System.Collections.Generic.IList<IExpression> ExpressionsList
    {
        set { this.expressionsList = value; }
    }

    public System.Collections.Generic.IDictionary<string, IExpression> ExpressionsDictionary
    {
        set { this.expressionsDictionary = value; }
    }

    public IExpression this[int index]
    {
        get
        {
            return this.expressionsList[index];
        }
    }

    public IExpression this[string key]
    {
        get { return this.expressionsDictionary[key]; }
    }
}

```

An example XML configuration of this class is shown below

```

<object id="genericExpressionHolder"
    type="Spring.Objects.Factory.Xml.GenericExpressionHolder,
    Spring.Core.Tests">
    <property name="ExpressionsList">
        <list element-type="Spring.Expressions.IExpression, Spring.Core">
            <value>1 + 1</value>
            <value>date('1856-7-9').Month</value>
            <value>'Nikola Tesla'.ToUpper()</value>
            <value>DateTime.Today > date('1856-7-9')</value>
        </list>
    </property>
    <property name="ExpressionsDictionary">
        <dictionary key-type="string" value-type="Spring.Expressions.IExpression, Spring.Core">
            <entry key="zero">
                <value>1 + 1</value>
            </entry>
            <entry key="one">
                <value>date('1856-7-9').Month</value>
            </entry>
            <entry key="two">
                <value>'Nikola Tesla'.ToUpper()</value>
            </entry>
            <entry key="three">
                <value>DateTime.Today > date('1856-7-9')</value>
            </entry>
        </dictionary>
    </property>
</object>

```

5.3.2.6. Collection Merging

As of Spring 1.3, the container supports the merging of collections. An application developer can define a parent-style `<list/>`, `<dictionary/>`, `<set/>` or `<name-value/>` element, and have child-style `<list/>`, `<dictionary/>`, `<set/>` or `<name-value/>` elements inherit and override values from the parent collection. That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child object mechanism. Readers unfamiliar with parent and child object definitions may wish to read the relevant section before continuing.

The following example demonstrates collection merging:

```
<object id="parent" abstract="true" type="Example.ComplexObject, Examples">
  <property name="AdminEmails">
    <name-values>
      <add key="administrator" value="administrator@example.com"/>
      <add key="support" value="support@example.com"/>
    </name-values>
  </property>
</object>

<object id="child" parent="parent" >
  <property name="AdminEmails">
    <!-- the merge is specified on the *child* collection definition -->
    <name-values merge="true">
      <add key="sales" value="sales@example.com"/>
      <add key="support" value="support@example.co.uk"/>
    </name-values>
  </property>
</object>
```

Notice the use of the `merge=true` attribute on the `<name-values/>` element of the `AdminEmails` property of the child object definition. When the child object is resolved and instantiated by the container, the resulting instance has an `AdminEmails Properties` collection that contains the result of the merging of the child's `AdminEmails` collection with the parent's `AdminEmails` collection.

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

The child `Properties` collection's value set inherits all property elements from the parent `<name-values/>`, and the child's value for the support value overrides the value in the parent collection. This merging behavior applies similarly to the `<list/>`, `<dictionary/>`, and `<set/>` collection types. In the specific case of the `<list/>` element, the semantics associated with the `ICollection` collection type, that is, the notion of an ordered collection of values, is maintained; the parent's values precede all of the child list's values. In the case of the `IDictionary`, `ISet`, and `NameValueCollection` collection types, no ordering exists. Hence no ordering semantics are in effect for the collection types that underlie the associated `IDictionary`, `ISet`, and `NameValueCollection` implementation types that the container uses internally.

5.3.2.7. Null and empty values

Spring treats empty arguments for properties and the like as empty Strings. The following XML-based configuration metadata snippet sets the email property to the empty String value (`""`)

```
<object type="Examples.ExampleObject, ExamplesLibrary">
  <property name="email" value=""/>
</object>
```

This results in the email property being set to the empty string value (`""`), in much the same way as can be seen in the following snippet of C# code

```
exampleObject.Email = "";
```

The `<null>` element is used to handle null values. For example:

```
<object type="Examples.ExampleObject, ExamplesLibrary">
  <property name="email"><null/></property>
</object>
```

This results in the email property being set to `null`, again in much the same way as can be seen in the following snippet of C# code:

```
exampleObject.Email = null;
```

5.3.2.8. Setting indexer properties

An indexer lets you set and get values from a collection using a familiar bracket `[]` notation. Spring's XML configuration supports the setting of indexer properties. Overloaded indexers as well as multiparameter indexers are also supported. The property expression parser described in Chapter 11, *Expression Evaluation* is used to perform the type conversion of the indexer name argument from a string in the XML file to a matching target type. As an example consider the following class

```
public class Person
{
    private IList favoriteNames = new ArrayList();

    private IDictionary properties = new Hashtable();

    public Person()
    {
        favoriteNames.Add("p1");
        favoriteNames.Add("p2");
    }

    public string this[int index]
    {
        get { return (string) favoriteNames[index]; }
        set { favoriteNames[index] = value; }
    }

    public string this[string keyName]
    {
        get { return (string) properties[keyName]; }
        set { properties.Add(keyName, value); }
    }
}
```

The XML configuration snippet to populate this object with data is shown below

```
<object id="person" type="Test.Objects.Person, Test.Objects">
  <property name="[0]" value="Master Shake"/>
  <property name="['one']" value="uno"/>
</object>
```



Note

The use of the property expression parser in Release 1.0.2 changed how you configure indexer properties. The following section describes this usage.

The older style configuration uses the following syntax

```
<object id="objectWithIndexer" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="Item[0]" value="my string value"/>
</object>
```

You can also change the name used to identify the indexer by adorning your indexer method declaration with the attribute `[IndexerName("MyItemName")]`. You would then use the string `MyItemName[0]` to configure the first element of that indexer.

There are some limitations to be aware in the older indexer configuration. The indexer can only be of a single parameter that is convertible from a string to the indexer parameter type. Also, multiple indexers are not supported. You can get around that last limitation currently if you use the `IndexerName` attribute.

5.3.2.9. Value and ref shortcut forms

Spring XML used to be even more verbose. What is now popular usage is actually the shortcut form of the original way to specify values and references.

There are also some shortcut forms that are less verbose than using the full `value` and `ref` elements. The `property`, `constructor-arg`, and `entry` elements all support a `value` attribute which may be used instead of embedding a full `value` element. Therefore, the following:

```
<property name="myProperty">
  <value>hello</value>
</property>

<constructor-arg>
  <value>hello</value>
</constructor-arg>

<entry key="myKey">
  <value>hello</value>
</entry>
```

are equivalent to:

```
<property name="myProperty" value="hello"/>

<constructor-arg value="hello"/>

<entry key="myKey" value="hello"/>
```

In general, when typing definitions by hand, you will probably prefer to use the less verbose shortcut form.

The `property` and `constructor-arg` elements support a similar shortcut `ref` attribute which may be used instead of a full nested `ref` element. Therefore, the following...

```
<property name="myProperty">
  <ref object="anotherObject"/>
</property>

<constructor-arg index="0">
  <ref object="anotherObject"/>
</constructor-arg>
```

is equivalent to...

```
<property name="myProperty" ref="anotherObject"/>

<constructor-arg index="0" ref="anotherObject"/>
```



Note

The shortcut form is equivalent to a `<ref object="xxx">` element; there is no shortcut for either the `<ref local="xxx">` or `<ref parent="xxx">` elements. For a local or parent `ref`, you must still use the long form.

Finally, the `entry` element allows a shortcut form to specify the key and/or value of a dictionary, in the form of `key/key-ref` and `value/value-ref` attributes. Therefore, the following

```
<entry>
  <key>
    <ref object="MyKeyObject"/>
  </key>
  <ref object="MyValueObject"/>
</entry>
```

Is equivalent to:

```
<entry key-ref="MyKeyObject" value-ref="MyValueObject"/>
```

As mentioned previously, the equivalence is to `<ref object="xxx">` and not the local or parent forms of object references.

5.3.2.10. Compound property names and Spring expression references

You can use compound or nested property names when you set object properties. Property names are interpreted using the Spring Expression Language (SpEL) and therefore can leverage its many features to set property names. For example, in this object definition a simple nested property name is configured

```
<object id="foo" type="Spring.Foo, Spring.Foo">
  <property name="bar.baz.name" value="Bingo"/>
</object>
```

As an example of some alternative ways to declare the property name, you can use SpEL's support for indexers to configure a Dictionary key value pair as an alternative to the nested `<dictionary>` element. More importantly, you can use the 'expression' element to refer to a Spring expression as the value of the property. Simple examples of this are shown below

```
<property name="minValue" expression="int.MinValue" />
<property name="weekFromToday" expression="DateTime.Today + 7"/>
```

Using SpEL's support for method evaluation, you can easily call static method on various helper classes in your XML configuration.

5.3.3. Declarative Event Listener Registration

In C# events are built right into the language thanks to the `event` keyword. Under the scenes, events are essentially a shorthand notation for delegates with some additional guidelines as to what the parameters to an event handler method should be (i.e. a sender `System.Object` and an `System.EventArgs` object).

```
public class EventSource
public event EventHandler Click;
```

In use, .NET events are combined with one or more event handler methods. Each handler method is programmatically added, or removed, from the event and corresponds to an object's method that should be invoked when a particular event occurs. When more than one handler method is added to an event, then each of the registered methods will be invoked in turn when an event occurs.

```
TestObject source = new TestObject();
TestEventHandler eventListener1 = new TestEventHandler();
TestEventHandler eventListener2 = new TestEventHandler();

source.Click += eventListener1.HandleEvent; // Adding the first event handler method to the event
source.Click += eventListener2.HandleEvent; // Adding a second event handler method to the event

source.OnClick(); // First eventListener1.HandleEvent is invoked, then eventListener2.HandleEvent
```

When `OnClick()` is invoked, the event is fired.

```
public void OnClick()
{
    if (Click != null)
    {
        Click(this, EventArgs.Empty); // Fire the event off to the registered handler methods
    }
}
```

}

One of the not so nice things about using events is that, without employing late binding, you declare the objects that are registered with a particular event programmatically. Spring .NET offers a way to declaratively register your handler methods with particular events using the `<listener>` element inside your `<object>` elements.

5.3.3.1. Declarative event handlers

Rather than having to specifically declare in your code that you are adding a method to be invoked on an event, using the `<listener>` element you can register a plain object's methods with the corresponding event declaratively in your application configuration.

Using the `listener` element you can:

- Configure a method to be invoked when an event is fired.
- Register a collection of handler methods based on a regular expression.
- Register a handler method against an event name that contains a regular expression.

5.3.3.2. Configuring a method to be invoked when an event is fired

The same event registration in the example above can be achieved using configuration using the `<listener>` element.

```
<object id="eventListener1" type="SpringdotNETEventsExample.TestEventHandler, SpringdotNETEventsExample">
  <!-- wired up to an event exposed on an instance -->
  <listener event="Click" method="HandleEvent">
    <ref object="source"/>
  </listener>
</object>

<object id="eventListener2" type="SpringdotNETEventsExample.TestEventHandler, SpringdotNETEventsExample">
  <!-- wired up to an event exposed on an instance -->
  <listener event="Click" method="HandleEvent">
    <ref object="source"/>
  </listener>
</object>
```

In this case the two different objects will have their `HandleEvent` method invoked, as indicated explicitly using the `method` attribute, when a `Click` event, as specified by the `event` attribute, is triggered on the object referred to by the `ref` element.

5.3.3.3. Registering a collection of handler methods based on a regular expression

Regular expressions can be employed to wire up more than one handler method to an object that contains one or more events.

```
<object id="eventListener" type="SpringdotNETEventsExample.TestEventHandler, SpringdotNETEventsExample">
  <listener method="Handle.+">
    <ref object="source"/>
  </listener>
</object>
```

Here all the `eventListener`'s handler methods that begin with 'Handle', and that have the corresponding two parameters of a `System.Object` and a `System.EventArgs`, will be registered against all events exposed by the `source` object.

You can also use the name of the event in regular expression to filter your handler methods based on the type of event triggered.

```
<object id="eventListener" type="SpringdotNETEventsExample.TestEventHandler, SpringdotNETEventsExample">
  <!-- For the Click event, the HandleClick handler method will be invoked. -->
  <listener method="Handle${event}">
    <ref object="source"/>
  </listener>
</object>
```

5.3.3.4. Registering a handler method against an event name that contains a regular expression

Finally, you can register an object's handler methods against a selection of events, filtering based on their name using a regular expression.

```
<object id="eventListener" type="SpringdotNETEventsExample.TestEventHandler, SpringdotNETEventsExample">
  <listener method="HandleEvent" event="Cl.+">
    <ref object="source"/>
  </listener>
</object>
```

In this example the `eventListener`'s `HandleEvent` handler method will be invoked for any event that begins with 'Cl'.

5.3.4. Using `depends-on`

If an object is a dependency of another that usually means that one object is set as a property of another. Typically you accomplish this with the `<ref/>` element in XML-based configuration metadata. However, sometimes dependencies between objects are less direct; for example, a static initializer in a class needs to be triggered, such as device driver registration. The `depends-on` attribute can explicitly force one or more objects to be initialized before the object using this element is initialized. The following example uses the `depends-on` attribute to express a dependency on a single object:

```
<object id="objectOne" type="Examples.ExampleObject, ExamplesLibrary" depends-on="manager">
  <property name="manager" ref="manager"/>
</object>

<object id="manager" type="Examples.ManagerObject, ExamplesLibrary"/>
```

To express a dependency on multiple objects, supply a list of object names as the value of the `'depends-on'` attribute, with commas, whitespace and semicolons used as valid delimiters:

```
<object id="objectOne" type="Examples.ExampleObject, ExamplesLibrary" depends-on="manager,accountDao">
  <property name="manager" ref="manager" />
</object>

<object id="manager" type="Examples.ManagerObject, ExamplesLibrary" />
<object id="accountDao" type="Examples.AdoAccountDao, ExamplesLibrary" />
```



Note

The `depends-on` attribute in the object definition can specify both an initialization time dependency and, in the case of a singleton object only, a corresponding destroy time dependency. Dependent objects that define a `depends-on` relationship with a given object are destroyed first, prior to the given object itself being destroyed. Thus `depends-on` can also control shutdown order.

5.3.5. Lazily-initialized objects

By default, `IApplicationContext` implementations eagerly pre-instantiate all singleton objects as part of the initialization process. Generally this pre-instantiation is desirable, because errors in configuration or the

surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton object by marking the object definition as lazy-initialized. A lazy-initialized object tells the IoC container to create an object instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the `'lazy-init'` attribute on the `<object/>` element; for example:

```
<object id="lazy" type="MyCompany.ExpensiveToObject, MyApp" lazy-init="true"/>

<object name="not.lazy" type="MyCompany.AnotherObject, MyApp"/>
```

When the preceding configuration is consumed by an `IApplcationContext`, the object named `lazy` is not eagerly pre-instantiated when the `IApplcationContext` is starting up, whereas the `not.lazy` object is eagerly pre-instantiated.

However, when a lazy-initialized object is a dependency of a singleton object that is *not* lazy-initialized, the `IApplcationContext` creates the lazy-initialized object at startup, because it must satisfy the singleton's dependencies. The lazy-initialized object is injected into a singleton object elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the `default-lazy-init` attribute on the `<objects/>` element; for example:

```
<objects default-lazy-init="true">
  <!-- no objects will be pre-instantiated... -->
</objects>
```

5.3.6. Autowiring collaborators

The Spring container is able to autowire relationships between collaborating objects. This means that it is possible to automatically let Spring resolve collaborators (other objects) for your object by inspecting the contents of the IoC container.. The autowiring functionality has five modes. Autowiring is specified per object and can thus be enabled for some object, while other objects will not be autowired. Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor arguments, thus saving a significant amount of typing. When using XML-based configuration metadata, the autowire mode for an object definition is specified by using the `autowire` attribute of the `<object/>` element. The following values are allowed:

Table 5.3. Autowiring modes

Mode	Explanation
no	No autowiring at all. This is the default value and you are encouraged not to change this for large applications, since specifying your collaborators explicitly gives you a feeling for what you're actually doing (always a bonus) and is a great way of somewhat documenting the structure of your system.
byName	This option will inspect the objects within the container, and look for an object named exactly the same as the property which needs to be autowired. For example, if you have an object definition that is set to autowire by name, and it contains a <code>Master</code> property, Spring.NET will look for an object definition named <code>Master</code> , and use it as the value of the <code>Master</code> property on your object definition.
byType	This option gives you the ability to resolve collaborators by type instead of by name. Supposing you have an <code>IOBJECTDefinition</code> with a collaborator typed <code>SqlConnection</code> , Spring.NET will search the entire object factory for an object definition of type <code>SqlConnection</code> and use it as the collaborator. <i>If 0 (zero) or more than 1 (one) object</i>

Mode	Explanation
	<i>definitions of the desired type exist in the container, a failure will be reported and you won't be able to use autowiring for that specific object.</i>
constructor	This is analogous to <i>byType</i> , but applies to constructor arguments. If there isn't exactly one object of the constructor argument type in the object factory, a fatal error is raised.
autodetect	Chooses <i>constructor</i> or <i>byType</i> through introspection of the object class. If a default constructor is found, <i>byType</i> gets applied.

Note that explicit dependencies in property and constructor-arg settings always override autowiring. Please also note that it is not currently possible to autowire so-called simple properties such as primitives, Strings, and Types (and arrays of such simple properties). (This is by-design and should be considered a feature.) When using either the *byType* or *constructor* autowiring mode, it is possible to wire arrays and typed-collections. In such cases all autowire candidates within the container that match the expected type will be provided to satisfy the dependency. Strongly-typed IDictionaries can even be autowired if the expected key type is string. An autowired IDictionary values will consist of all object instances that match the expected type, and the IDictionary's keys will contain the corresponding object names.

Autowire behavior can be combined with dependency checking, which will be performed after all autowiring has been completed. It is important to understand the various advantages and disadvantages of autowiring. Some advantages of autowiring include:

- Autowiring can significantly reduce the volume of configuration required. However, mechanisms such as the use of a object template (discussed elsewhere in this chapter) are also valuable in this regard.
- Autowiring can cause configuration to keep itself up to date as your objects evolve. For example, if you need to add an additional dependency to a class, that dependency can be satisfied automatically without the need to modify configuration. Thus there may be a strong case for autowiring during development, without ruling out the option of switching to explicit wiring when the code base becomes more stable.

Some disadvantages of autowiring:

- Autowiring is more magical than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity which might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.

Another issue to consider when autowiring by type is that multiple object definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or IDictionary, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity will not be arbitrarily resolved. Instead, if no unique object definition is available, an Exception will be thrown.

When deciding whether to use autowiring, there is no wrong or right answer in all cases. A degree of consistency across a project is best though; for example, if autowiring is not used in general, it might be confusing to developers to use it just to wire one or two object definitions.

5.3.7. Checking for dependencies

The Spring IoC container can check for unresolved dependencies of an object deployed into the container. When enabling checking for unresolved dependencies all properties of the object must have an explicit values set for them in the object definition or have their values set via autowiring.

This feature useful when you want to ensure that all properties (or all properties of a certain type) are set on an object. An object often has default values for many properties, or some properties do not apply to all usage scenarios, so this feature is of limited use. You can enable dependency checking per object, just as with the autowiring functionality. The default is not *not* check dependencies. In XML-based configuration metadata, you specify dependency checking via the `dependency-check` attribute in an object definition, which may have the following values.

Table 5.4. Dependency checking modes

Mode	Explanation
none	(Default) No dependency checking. Properties of the object which have no value specified for them are simply not set.
simple	Dependency checking for primitive types and collections (everything except collaborators).
object	Dependency checking for collaborators only.
all	Dependency checking for collaborators, primitive types and collections.

5.3.8. Method injection

In most application scenarios, most object in the container are singletons. When a singleton object needs to collaborate with (use) another singleton object, or a non-singleton object needs to collaborate with another non-singleton object, you typically handle the dependency by defining one object as a property of the other. A problem arises when the object lifecycles are different. Suppose singleton object A needs to use a non-singleton (prototype) object B, perhaps on each method invocation on A. The container only creates the singleton object A once, and thus only get the opportunity to set the properties. The container cannot provide object A with a new instance of object B every time one is needed.

A solution is to forego some inversion of control. You can make object A aware of the container by implementing the `IApplicationContextAware` interface, and by making a `GetObject("B")` call to the container ask for (a typically new) object B every time it needs it. Find below an example of this approach

```
using System.Collections;
using Spring.Objects.Factory;

namespace Fiona.Apple
{
    public class CommandManager : IObjectFactoryAware
    {
        private IObjectFactory objectFactory;

        public object Process(IDictionary commandState)
        {
            // grab a new instance of the appropriate Command
            Command command = CreateCommand();
            // set the state on the (hopefully brand new) Command instance
            command.State = commandState;
            return command.Execute();
        }

        // the Command returned here could be an implementation that executes asynchronously, or whatever
        protected Command CreateCommand()
        {
            return (Command) objectFactory.GetObject("command"); // notice the Spring API dependency
        }

        public IObjectFactory ObjectFactory
        {
            set { objectFactory = value; }
        }
    }
}
```

```

    }
}

```

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

5.3.8.1. Lookup Method Injection

Lookup method injection is the ability of the container to override methods on *container managed objects*, to return the result of looking up another named object in the container. The lookup typically involves a prototype object as in the scenario described in the preceding section. The Spring framework implements this method injection by dynamically generating a subclass overriding the method using the classes in the `System.Reflection.Emit` namespace.



Note

You can read more about the motivation for Method Injection in this blog entry [<http://blog.springframework.com/rod/?p=1>].

Looking at the `CommandManager` class in the previous code snippet, you see that the Spring container will dynamically override the implementation of the `CreateCommand()` method. Your `CommandManager` class will not have any Spring dependencies, as can be seen in this reworked example below:

```

using System.Collections;

namespace Fiona.Apple
{
    public abstract class CommandManager
    {
        public object Process(IDictionary commandState)
        {
            Command command = CreateCommand();
            command.State = commandState;
            return command.Execute();
        }

        // okay... but where is the implementation of this method?
        protected abstract Command CreateCommand();
    }
}

```

In the client class containing the method to be injected (the `CommandManager` in this case) the method to be injected requires a signature of the following form:

```
<public|protected> [abstract] <return-type> TheMethodName(no-arguments);
```

If the method is abstract, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. Let's look at an example:

```

<!-- a stateful object deployed as a prototype (non-singleton) -->
<object id="command" class="Fiona.Apple.AsyncCommand, Fiona" singleton="false">
  <!-- inject dependencies here as required -->
</object>

<!-- commandProcessor uses a statefulCommandHelper -->
<object id="commandManager" type="Fiona.Apple.CommandManager, Fiona">
  <lookup-method name="CreateCommand" object="command"/>

```

```
</object>
```

The object identified as `commandManager` will call its own method `CreateCommand` whenever it needs a new instance of the `command` object. You must be careful to deploy the `command` object as prototype, if that is actually what is needed. If it is deployed as a singleton the same instance of `singleShotHelper` will be returned each time.

Note that lookup method injection can be combined with Constructor Injection (supplying optional constructor arguments to the object being constructed), and also with Setter Injection (settings properties on the object being constructed).

5.3.8.2. Arbitrary method replacement

A less commonly useful form of method injection than Lookup Method Injection is the ability to replace arbitrary methods in a managed object with another method implementation.

With XML-based configuration metadata, you can use the `replaced-method` element to replace an existing method implementation with another, for a deployed object. Consider the following class, with a method `ComputeValue`, which we want to override:

```
public class MyValueCalculator {

    public virtual string ComputeValue(string input) {
        // ... some real code
    }

    // ... some other methods
}
```

A class implementing the `Spring.Objects.Factory.Support.IMethodReplacer` interface is needed to provide the new (injected) method definition.

```
/// <summary>
/// Meant to be used to override the existing ComputeValue(string)
/// implementation in MyValueCalculator.
/// </summary>
public class ReplacementComputeValue : IMethodReplacer
{
    public object Implement(object target, MethodInfo method, object[] arguments)
    {
        // get the input value, work with it, and return a computed result...
        string value = (string) arguments[0];
        // compute...
        return result;
    }
}
```

The object definition to deploy the original class and specify the method override would look like this:

```
<object id="myValueCalculator" type="Examples.MyValueCalculator, ExampleAssembly">
  <!-- arbitrary method replacement -->
  <replaced-method name="ComputeValue" replacer="replacementComputeValue">
    <arg-type match="String"/>
  </replaced-method>
</object>

<object id="replacementComputeValue" type="Examples.ReplacementComputeValue, ExampleAssembly"/>
```

You can use one or more contained `arg-type` elements within the `replaced-method` element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match `System.String`.

```
System.String
```

```
String
Str
```

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by allowing you to typ just the shortest string which will match an argument type.

5.3.9. Setting a reference using the members of other objects and classes.

This section details those configuration scenarios that involve the setting of properties and constructor arguments using the members of other objects and classes. This kind of scenario is quite common, especially when dealing with legacy classes that you cannot (or won't) change to accommodate some of Spring.NET's conventions... consider the case of a class that has a constructor argument that can only be calculated by going to say, a database. The `MethodInvokingFactoryObject` handles exactly this scenario ... it will allow you to inject the result of an arbitrary method invocation into a constructor (as an argument) or as the value of a property setter. Similarly, `PropertyRetrievingFactoryObject` and `FieldRetrievingFactoryObject` allow you to retrieve values from another object's property or field value. These classes implement the `IFactoryObject` interface which indicates to Spring.NET that this object is itself a factory and the factories product, not the factory itself, is what will be associated with the object id. Factory objects are discussed further in Section 5.9.3, "Customizing instantiation logic using `IFactoryObjects`"

5.3.9.1. Setting a reference to the value of property.

The `PropertyRetrievingFactoryObject` is an `IFactoryObject` that addresses the scenario of setting one of the properties and / or constructor arguments of an object to the value of a property exposed on another object or class. One can use it to get the value of any **public** property exposed on either an instance or a class (in the case of a property exposed on a class, the property must obviously be static).

In the case of a property exposed on an instance, the target object that a `PropertyRetrievingFactoryObject` will evaluate can be either an object instance specified directly inline or a reference to another arbitrary object. In the case of a static property exposed on a class, the target object will be the class (the .NET `System.Type`) exposing the property.

The result of evaluating the property lookup may then be used in another object definition as a property value or constructor argument. Note that nested properties are supported for both instance and class property lookups. The `IFactoryObject` is discussed more generally in Section 5.9.3, "Customizing instantiation logic using `IFactoryObjects`".

Here's an example where a property path is used against another object instance. In this case, an inner object definition is used and the property path is nested, i.e. `spouse.age`.

```
<object name="person" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="age" value="20"/>
  <property name="spouse">
    <object type="Spring.Objects.TestObject, Spring.Core.Tests">
      <property name="age" value="21"/>
    </object>
  </property>
</object>

// will result in 21, which is the value of property 'spouse.age' of object 'person'
<object name="theAge" type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="person"/>
  <property name="TargetProperty" value="spouse.age"/>
</object>
```

An example of using a `PropertyRetrievingFactoryObject` to evaluate a static property is shown below.

```

<object id="cultureAware"
    type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
    <property name="culture" ref="cultureFactory"/>
</object>

<object id="cultureFactory"
    type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject, Spring.Core">
    <property name="StaticProperty">
        <value>System.Globalization.CultureInfo.CurrentCulture, Mscorlib</value>
    </property>
</object>

```

Similarly, an example showing the use of an instance property is shown below.

```

<object id="instancePropertyCultureAware"
    type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
    <property name="Culture" ref="instancePropertyCultureFactory"/>
</object>

<object id="instancePropertyCultureFactory"
    type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject, Spring.Core">
    <property name="TargetObject" ref="instancePropertyCultureAwareSource"/>
    <property name="TargetProperty" value="MyDefaultCulture"/>
</object>

<object id="instancePropertyCultureAwareSource"
    type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests"/>

```

5.3.9.2. Setting a reference to the value of field.

The `FieldRetrievingFactoryObject` class addresses much the same area of concern as the `PropertyRetrievingFactoryObject` described in the previous section. However, as its name might suggest, the `FieldRetrievingFactoryObject` class is concerned with looking up the value of a **public** field exposed on either an instance or a class (and similarly, in the case of a field exposed on a class, the field must obviously be static).

The following example demonstrates using a `FieldRetrievingFactoryObject` to look up the value of a (public, static) field exposed on a class

```

<object id="withTypesField"
    type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
    <property name="Types" ref="emptyTypesFactory"/>
</object>

<object id="emptyTypesFactory"
    type="Spring.Objects.Factory.Config.FieldRetrievingFactoryObject, Spring.Core">
    <property name="TargetType" value="System.Type, Mscorlib"/>
    <property name="TargetField" value="EmPTytypeS"/>
</object>

```

The example in the next section demonstrates the look up of a (public) field exposed on an object instance.

```

<object id="instanceCultureAware"
    type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
    <property name="Culture" ref="instanceCultureFactory"/>
</object>

<object id="instanceCultureFactory"
    type="Spring.Objects.Factory.Config.FieldRetrievingFactoryObject, Spring.Core">
    <property name="TargetObject" ref="instanceCultureAwareSource"/>
    <property name="TargetField" value="Default"/>
</object>

<object id="instanceCultureAwareSource"
    type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests"/>

```

5.3.9.3. Setting a property or constructor argument to the return value of a method invocation.

The `MethodInvokingFactoryObject` rounds out the trio of classes that permit the setting of properties and constructor arguments using the members of other objects and classes. Whereas the `PropertyRetrievingFactoryObject` and `FieldRetrievingFactoryObject` classes dealt with simply looking up and returning the value of property or field on an object or class, the `MethodInvokingFactoryObject` allows one to set a constructor or property to the return value of an arbitrary method invocation,

The `MethodInvokingFactoryObject` class handles both the case of invoking an (instance) method on another object in the container, and the case of a static method call on an arbitrary class. Additionally, it is sometimes necessary to invoke a method just to perform some sort of initialization.... while the mechanisms for handling object initialization have yet to be introduced (see Section 5.6.1.1, “`InitializingObject` / `init-method`”), these mechanisms do not permit any arguments to be passed to any initialization method, and are confined to invoking an initialization method on the object that has just been instantiated by the container. The `MethodInvokingFactoryObject` allows one to invoke pretty much **any** method on any object (or class in the case of a static method).

The following example (in an XML based `IOBJECTFactory` definition) uses the `MethodInvokingFactoryObject` class to force a call to a static factory method prior to the instantiation of the object...

```
<object id="force-init"
  type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="StaticMethod">
    <value>ExampleNamespace.ExampleInitializerClass.Initialize</value>
  </property>
</object>
<object id="myService" depends-on="force-init"/>
```

Note that the definition for the `myService` object has used the `depends-on` attribute to refer to the `force-init` object, which will force the initialization of the `force-init` object first (and thus the calling of its configured `StaticMethod` static initializer method, when `myService` is first initialized. Please note that in order to effect this initialization, the `MethodInvokingFactoryObject` object **must** be operating in `singleton` mode (the default.. see the next paragraph).

Note that since this class is expected to be used primarily for accessing factory methods, this factory defaults to operating in `singleton` mode. As such, as soon as all of the properties for a `MethodInvokingFactoryObject` object have been set, and if the `MethodInvokingFactoryObject` object is still in `singleton` mode, the method will be invoked immediately and the return value cached for later access. The first request by the container for the factory to produce an object will cause the factory to return the cached return value for the current request (and all subsequent requests). The `IsSingleton` property may be set to `false`, to cause this factory to invoke the target method each time it is asked for an object (in this case there is obviously no caching of the return value).

A static target method may be specified by setting the `targetMethod` property to a string representing the static method name, with `TargetType` specifying the `Type` that the static method is defined on. Alternatively, a target instance method may be specified, by setting the `TargetObject` property to the name of another Spring.NET managed object definition (the target object), and the `TargetMethod` property to the name of the method to call on that target object.

Arguments for the method invocation may be specified in two ways (or even a mixture of both)... the first involves setting the `Arguments` property to the list of arguments for the method that is to be invoked. Note that the ordering of these arguments is significant... the order of the values passed to the `Arguments` property must be the same as the order of the arguments defined on the method signature, including the argument `Type`. This is shown in the example below

```
<object id="myObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetType" value="Whatever.MyClassFactory, MyAssembly"/>
  <property name="TargetMethod" value="GetInstance"/>

  <!-- the ordering of arguments is significant -->
  <property name="Arguments">
    <list>
      <value>1st</value>
      <value>2nd</value>
      <value>and 3rd arguments</value>
      <!-- automatic Type-conversion will be performed prior to invoking the method -->
    </list>
  </property>
</object>
```

The second way involves passing an arguments dictionary to the `NamedArguments` property... this dictionary maps argument names (Strings) to argument values (any object). The argument names are not case-sensitive, and order is (obviously) not significant (since dictionaries by definition do not have an order). This is shown in the example below

```
<object id="myObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetObject">
    <object type="Whatever.MyClassFactory, MyAssembly"/>
  </property>
  <property name="TargetMethod" value="Execute"/>

  <!-- the ordering of named arguments is not significant -->
  <property name="NamedArguments">
    <dictionary>
      <entry key="argumentName"><value>1st</value></entry>
      <entry key="finalArgumentName"><value>and 3rd arguments</value></entry>
      <entry key="anotherArgumentName"><value>2nd</value></entry>
    </dictionary>
  </property>
</object>
```

The following example shows how use `MethodInvokingFactoryObject` to call an instance method.

```
<object id="myMethodObject" type="Whatever.MyClassFactory, MyAssembly" />

<object id="myObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="myMethodObject"/>
  <property name="TargetMethod" value="Execute"/>
</object>
```

The above example could also have been written using an anonymous inner object definition... if the object on which the method is to be invoked is not going to be used outside of the factory object definition, then this is the preferred idiom because it limits the scope of the object on which the method is to be invoked to the surrounding factory object.

Finally, if you want to use `MethodInvokingFactoryObject` in conjunction with a method that has a variable length argument list, then please note that the variable arguments need to be passed (and configured) as a *list*. Let us consider the following method definition that uses the `params` keyword (in C#), and its attendant (XML) configuration...

```
[C#]
public class MyClassFactory
{
    public object CreateObject(Type objectType, params string[] arguments)
    {
        return ... // implementation elided for clarity...
    }
}
```

```
<object id="myMethodObject" type="Whatever.MyClassFactory, MyAssembly" />
```



```
<object id="paramsMethodObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="myMethodObject"/>
  <property name="TargetMethod" value="CreateObject"/>
  <property name="Arguments">
    <list>
      <value>System.String</value>
      <!-- here is the 'params string[] arguments' -->
      <list>
        <value>1st</value>
        <value>2nd</value>
      </list>
    </list>
  </property>
</object>
```

5.3.10. Provided IFactoryObject implementations

In addition to `PropertyRetrievingFactoryObject`, `MethodInvokingFactoryObject`, and `FieldRetrievingFactoryObject` Spring.NET comes with other useful implementations of the `IFactoryObject` interface. These are discussed below.

5.3.10.1. Common logging

The `LogFactoryObject` is useful when you would like to share a `Common.Logging` log object across a number of classes instead of creating a logging instance per class or class hierarchy. Information on the `Common.Logging` project can be found here [<http://netcommon.sourceforge.net/>]. In the example shown below the same logging instance, with a logging category name of "DAOLogger", is used in both the `SimpleAccountDao` and `SimpleProductDao` data access objects.

```
<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.net
    http://www.springframework.net/xsd/spring-objects.xsd" >

  <object name="daoLogger" type="Spring.Objects.Factory.Config.LogFactoryObject, Spring.Core">
    <property name="logName" value="DAOLogger"/>
  </object>

  <object name="productDao" type="PropPlayApp.SimpleProductDao, PropPlayApp ">
    <property name="maxResults" value="100"/>
    <property name="dbConnection" ref="myConnection"/>
    <property name="log" ref="daoLogger"/>
  </object>

  <object name="accountDao" type="PropPlayApp.SimpleAccountDao, PropPlayApp ">
    <property name="maxResults" value="100"/>
    <property name="dbConnection" ref="myConnection"/>
    <property name="log" ref="daoLogger"/>
  </object>

  <object name="myConnection" type="System.Data.Odbc.OdbcConnection, System.Data">
    <property name="connectionstring" value="dsn=MyDSN;uid=sa;pwd=myPassword;"/>
  </object>

</objects>
```

5.4. Object Scopes

When you create an object definition, you create a *recipe* for creating actual instances of the class defined by that object definition. The idea that an object definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular object definition, but also the *scope* of the objects created from a particular object

definition. This approach powerful and flexible in that you can *choose* the scope of the objects you create through configuration instead of having to bake in the scope of an object at the .NET class level. Objects can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports five scopes, three of which are available only if you use a web-aware [ApplicationContext](#).

The following scopes supported. Support for user defined custom scopes is planned for Spring .NET 2.0.

Table 5.5. Object Scopes

Scope	Description
singleton	Scopes a single object definition to a single object instance per Spring IoC container.
prototype	Scopes a single object definition to any number of object instances.
request	Scopes a single object definition to the lifecycle of a single HTTP request; that is, each and every HTTP request has its own instance of an object created off the back of a single object definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single object definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single object definition to the lifecycle of a web application. Only valid in the context of a web-aware Spring ApplicationContext.

5.4.1. The singleton scope

Singleton scoped objects have only one shared instance of an object managed by the container. All request for objects with an id or ids matching that object definition result in that one specific object instance being returned by the Spring container.

To put it another way, when you define an object definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that object definition. This single instance is stored in a cache of such singleton object, and all subsequent requests and references for that named object return the cached object.

Spring's concept of a singleton differs from the Singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one *and only one* instance of a particular class is created per ApplicationDomain. The scope of the Spring singleton is best described as *per container and per object*. This means that if you define one object for a particular class in a single Spring container, then the Spring container creates one *and only one* instance of the class defined by that object definition. The singleton scope is the default scope in Spring. To define an object as a singleton in XML, you would write, for example:

```
<object id="accountService" type="MyApp.DefaultAccountService, MyApp"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<object id="accountService" type="MyApp.DefaultAccountService, MyApp" singleton="true"/>
```

5.4.2. The prototype scope

The non-singleton, prototype scope of object deployment results in the creation of a *new object instance* every time a request for that specific object is made. That is, the object is injected into another object or you request

through a `GetObject()` method call on the container. As a rule use the prototype scope for all objects that are stateful and the singleton scope for stateless objects.

The following examples defines an object as a prototype in XML:

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary" scope="prototype"/>
```



Note

The `<singleton/>` attribute was introduced Spring 1.0 as there were only two types of scopes, singleton and prototype. The element `singleton=true` refers to singleton scope and `singleton=false` refers to prototype scope. In Spring 1.1 the additional web scopes were introduced along with the new element 'scope'. The scope element is the preferred element to use.

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype object: the container instantiates, configures, decorates and otherwise assembles a prototype object, hands it to the client, with no further record of that prototype instance. Thus, although *initialization* lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured *destruction* lifecycle callbacks are *not* called. The client code must clean up prototype-scoped objects and release any expensive resources that the prototype object(s) are holding. To get the Spring container to release resources held by prototype-scoped objects, try using a custom object post processor which would hold a reference to the objects that need to be cleaned up.

In some respects, the Spring container's role in regard to a prototype-scoped object is a replacement for the C# 'new' operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of an object in the Spring container, see Section 5.6.1, "Lifecycle interfaces".)

5.4.3. Singleton objects with prototype-object dependencies

When you use singleton-scoped objects with dependencies on prototype objects, be aware that *dependencies are resolved at instantiation time*. Thus if you dependency-inject a prototype-scoped objects into a singleton-scoped object, a new prototype object is instantiated and then dependency-injected into the singleton object. The prototype instance is the sole instance that is ever supplied to the singleton-scoped object.

However, suppose you want the singleton-scoped object to acquire a new instance of the prototype-scoped object repeatedly at runtime. You cannot dependency-inject a prototype-scoped object into your singleton object, because that injection occurs only once, when the Spring container is instantiating the singleton object and resolving and injecting its dependencies. If you need a new instance of a prototype object at runtime more than once, see Section 5.3.8, "Method injection".

5.4.4. Request, session and web application scopes

The request, session and application scopes are *only* available if you use a web-aware Spring [IApplicationContext](#) implementation, such as [WebApplicationContext](#). If you use these scopes with regular Spring IoC containers such as the [XmlApplicationContext](#), you will get an exception complaining about an unknown object scope.

Please refer to the web documentation on object scopes for more information.

5.5. Type conversion

Type converters are responsible for converting objects from one type to another. When using the XML based file to configure the IoC container, string based property values are converted to the target property type. Spring will

rely on the standard .NET support for type conversion unless an alternative `TypeConverter` is registered for a given type. How to register custom `TypeConverters` will be described shortly. As a reminder, the standard .NET type converter support works by associating a `TypeConverter` attribute with the class definition by passing the type of the converter as an attribute argument.³ For example, an abbreviated class definition for the BCL type `Font` is shown below.

```
[Serializable, TypeConverter(typeof(FontConverter)), ...]
public sealed class Font : MarshalByRefObject, ICloneable, ISerializable, IDisposable
{
    // Methods

    ... etc ..
}
```

5.5.1. Type Conversion for Enumerations

The default type converter for enumerations is the `System.ComponentModel.EnumConverter` class. To specify the value for an enumerated property, simply use the name of the property. For example the `TestObject` class has a property of the enumerated type `FileMode`. One of the values for this enumeration is named `Create`. The following XML fragment shows how to configure this property

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="name" value="Rod"/>
  <property name="FileMode" value="Create"/>
</object>
```

5.5.2. Built-in TypeConverters

Spring.NET pre-registers a number of custom `TypeConverter` instances (for example, to convert a type expressed as a string into a real `System.Type` object). Each of those is listed below and they are all located in the `Spring.Objects.TypeConverters` namespace of the `Spring.Core` library.

Table 5.6. Built-in `TypeConverters`

Type	Explanation
<code>RuntimeTypeConverter</code>	Parses strings representing <code>System.Types</code> to actual <code>System.Types</code> and the other way around.
<code>FileInfoConverter</code>	Capable of resolving strings to a <code>System.IO.FileInfo</code> object.
<code>StringArrayConverter</code>	Capable of resolving a comma-delimited list of strings to a string-array and vice versa.
<code>UriConverter</code>	Capable of resolving a string representation of a <code>Uri</code> to an actual <code>Uri</code> -object.
<code>CredentialConverter</code>	Capable of resolving a string representation of a credential for Web client authentication into an instance of <code>System.Net.ICredentials</code>
<code>StreamConverter</code>	Capable of resolving <code>Spring IResource Uri (string)</code> to its corresponding <code>InputStream</code> -object.
<code>ResourceConverter</code>	Capable of resolving <code>Spring IResource Uri (string)</code> to an <code>IResource</code> object.

³More information about creating custom `TypeConverter` implementations can be found online at Microsoft's MSDN website, by searching for *Implementing a Type Converter*.

Type	Explanation
ResourceManagerConverter	Capable of resolving a two part string (resource name, assembly name) to a <code>System.Resources.ResourceManager</code> object.
RgbColorConverter	Capable of resolving a comma separated list of Red, Green, Blue integer values to a <code>System.Drawing.Color</code> structure.
ExpressionConverter	Capable of resolving a string into an instance of an object that implements the <code>IExpression</code> interface.
NameValueCollectionConverter	Capable of resolving an XML formatted string to a <code>Specialized.NameValueCollection</code>
RegexConverter	Capable of resolving a string into an instance of <code>Regex</code>
RegistryKeyConverter	Capable of resolving a string into a <code>Microsoft.Win32.RegistryKey</code> object.

Spring.NET uses the standard .NET mechanisms for the resolution of `System.Types`, including, but not limited to checking any configuration files associated with your application, checking the Global Assembly Cache (GAC), and assembly probing.

5.5.3. Custom Type Conversion

There are a few ways to register custom type converters. The fundamental storage area in Spring for custom type converters is the `TypeConverterRegistry` class. The *most convenient* way if using an XML based implementation of `IObjectFactory` or `IApplicationContext` is to use the custom configuration section handler `TypeConverterSectionHandler`. This is demonstrated in section Section 5.11, “Configuration of `IApplicationContext`”

An alternate approach, present for legacy reasons in the port of Spring.NET from the Java code base, is to use the object factory post-processor `Spring.Objects.Factory.Config.CustomConverterConfigurer`. This is described in the next section.

If you are constructing your IoC container Programmatically then you should use the `RegisterCustomConverter(Type requiredType, TypeConverter converter)` method of the `ConfigurableObjectFactory` interface.

5.5.3.1. Using CustomConverterConfigurer

This section shows in detail how to define a custom type converter that does not use the .NET `TypeConverter` attribute. The type converter class is standalone and inherits from the `TypeConverter` class. It uses the legacy factory post-processor approach.

Consider a user class *ExoticType*, and another class *DependsOnExoticType* which needs *ExoticType* set as a property:

```
public class ExoticType
{
    private string name;

    public ExoticType(string name)
    {
        this.name = name;
    }
}
```

```

    public string Name
    {
        get { return this.name; }
    }
}

```

and

```

public class DependsOnExoticType
{
    public DependsOnExoticType() {}

    private ExoticType exoticType;

    public ExoticType ExoticType
    {
        get { return this.exoticType; }
        set { this.exoticType = value; }
    }

    public override string ToString()
    {
        return exoticType.Name;
    }
}

```

When things are properly set up, we want to be able to assign the type property as a string, which a `TypeConverter` will convert into a real `ExoticType` object behind the scenes:

```

<object name="sample" type="SimpleApp.DependsOnExoticType, SimpleApp">
  <property name="exoticType" value="aNameForExoticType"/>
</object>

```

The `TypeConverter` looks like this:

```

public class ExoticTypeConverter : TypeConverter
{
    public ExoticTypeConverter()
    {
    }

    public override bool CanConvertFrom (
        ITypeDescriptorContext context,
        Type sourceType)
    {
        if (sourceType == typeof (string))
        {
            return true;
        }
        return base.CanConvertFrom (context, sourceType);
    }

    public override object ConvertFrom (
        ITypeDescriptorContext context,
        CultureInfo culture, object value)
    {
        string s = value as string;
        if (s != null)
        {
            return new ExoticType(s.ToUpper());
        }
        return base.ConvertFrom (context, culture, value);
    }
}

```

Finally, we use the `CustomConverterConfigurer` to register the new `TypeConverter` with the `IApplicationContext`, which will then be able to use it as needed:

```

<object id="customConverterConfigurer"
  type="Spring.Objects.Factory.Config.CustomConverterConfigurer, Spring.Core">

```

```
<property name="CustomConverters">
  <dictionary>
    <entry key="SimpleApp.ExoticType">
      <object type="SimpleApp.ExoticTypeConverter"/>
    </entry>
  </dictionary>
</property>
</object>
```

5.6. Customizing the nature of an object

5.6.1. Lifecycle interfaces

To interact with the container's management of the object lifecycle, you can implement the Spring `InitializingObject` and standard `System.IDisposable` interfaces. The container calls `AfterPropertiesSet()` method for the former and the `Dispose()` method for the latter, thus allowing you to do things upon the initialization and destruction of your objects. You can also achieve the same integration with the container without coupling your objects to Spring interfaces though the use of `init-method` and `destroy-method` object definition metadata.

Internally, Spring.NET uses implementations of the `IObjectPostProcessor` interface to process any call interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring.NET does not offer out-of-the-box, you can implement an `IObjectPostProcessor` yourself. For more information see Section 5.9.1, “Customizing objects with `IObjectPostProcessors`”.

5.6.1.1. `IInitializingObject` / `init-method`

The `Spring.Objects.Factory.IInitializingObject` interface allows an object to perform initialization work after all the necessary properties on an object are set by the container. The `IInitializingObject` interface specifies a single method:

- `void AfterPropertiesSet()`: called after all properties have been set by the container. This method enables you to do checking to see if all necessary properties have been set correctly, or to perform further initialization work. You can throw *any* `Exception` to indicate misconfiguration, initialization failures, etc.

It is recommended that you do not use the `IInitializingObject` interface because it unnecessarily couples the code to Spring. Alternatively, specify an POJO initialization method. In the case of XML-based configuration metadata, you use the `init-method` attribute to specify the name of the method that has a void no-argument signature. For example, the following definition:

```
<object id="exampleInitObject" type="Examples.ExampleObject" init-method="init"/>
[C#]
public class ExampleObject
{
    public void Init()
    {
        // do some initialization work
    }
}
```

...is exactly the same as...

```
<object id="exampleInitObject" type="Examples.AnotherExampleObject"/>
[C#]
public class AnotherExampleObject : IInitializingObject
{
    public void AfterPropertiesSet()
    {
```

```
// do some initialization work
}
}
```

... but does not couple the code to Spring.NET.

5.6.1.2. IDisposable / destroy-method

Implementing the `System.IDisposable` interface allows an object to get a callback when the container containing it is destroyed. The `IDisposable` interface specifies a single method:

- `void Dispose()`: and is called on destruction of the container. This allows you to release any resources you are keeping in this object (such as database connections). You can throw *any* `Exception` here... however, any such `Exception` will not stop the destruction of the container - it will only get logged.

Since the `IDisposable` interface resides in the core .NET library, it does not couple your class to Spring as in the case with the `IInitializingObject` interface. However, you may also specify a destruction method that is not tied to the `IDisposable` interface. In the case of XML-based configuration metadata, you use the `destroy-method` attribute to specify the name of the method that has a void no-argument signature. For example, the following definition:

```
<object id="exampleInitObject" type="Examples.ExampleObject" destroy-method="cleanup"/>
[C#]
public class ExampleObject
{
    public void cleanup()
    {
        // do some destruction work (such as closing any open connection (s))
    }
}
```

is exactly the same as:

```
<object id="exampleInitObject" type="Examples.AnotherExampleObject"/>
[C#]
public class AnotherExampleObject : IDisposable
{
    public void Dispose()
    {
        // do some destruction work
    }
}
```

5.6.2. IApplicationContextAware and IObjectNameAware

When an `IApplicationContext` creates a class that implements the `IApplicationContextAware` interface, the class is provided with a reference to that `IApplicationContext`.

```
public interface IApplicationContextAware {
    IApplicationContext ApplicationContext {
        set;
    }
}
```

Thus objects can manipulate programmatically the `IApplicationContext` that created them, through the `IApplicationContext` interface, or by casting the reference to a known subclass of this interface, such as `IConfigurableApplicationContext`, which exposes additional functionality. One use would be the programmatic retrieval of other objects. Sometimes this capability is useful; however, in general you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to objects as properties. Other methods of the `IApplicationContext` provide access to file resources, publishing

application events, and accessing a [IMessageSource](#). These additional features are described in Section 5.10, “The `IApplicationContext`”

5.6.2.1. `IObjNameAware`

When an [IApplicationContext](#) creates a class that implements the `Spring.Objects.Factory.IObjectNameAware` interface, the class is provided with a reference to the name defined in its associated object definition.

```
public interface IObjectNameAware {
    String ObjectName {
        set;
    }
}
```

The callback is invoked after population of normal object properties but before an initialization callback such as [IInitializingObject](#)'s `AfterPropertiesSet` method or a custom initialization method is invoked.

5.7. Object definition inheritance

An object definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on. A child object definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child object definitions can save a lot of typing. Effectively, this is a form of templating.

If you work with an `IApplicationContext` interface programmatically, child object definitions are represented by the `ChildObjectDefinition` class. Most users do not work with them on this level, instead configuring object definitions declaratively in something like the `XmlApplicationContext`. When you use XML-based configuration metadata, you indicate a child object using the `parent` attribute, specifying the parent object definition as the value of this attribute.

```
<object id="inheritedTestObject" type="Spring.Objects.TestObject, Spring.Core.Tests" abstract="true">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsWithDifferentClass" type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObject" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</object>
```

A child object definition uses the object class from the parent definition if none is specified, but can also override it. In the latter case, the child object class must be compatible with the parent, that is, it must accept the parent's property values.

A child object definition inherits constructor argument values, property values and method overrides from the parent, with the option to add new values. Any initialization method, destroy method and/or static factory methods that you specify will override the corresponding parent settings.

The remaining settings are always be taken from the child definition: `depends on`, `autowire mode`, `dependency check`, `singleton`, `lazy init`.

The preceding example explicitly marks the parent object definition as abstract using the `abstract` attribute. If the parent definition does not specify a class, explicitly marking the parent object definition as abstract is required, as follows:

```
<object id="inheritedTestObjectWithoutClass" abstract="true">
```



```

<property name="name" value="parent"/>
<property name="age" value="1"/>
</object>
<object id="inheritsWithClass" type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObjectWithoutClass" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</object>

```

The parent object cannot be instantiated on its own since it is incomplete, and it is also explicitly marked as abstract. When a definition is abstract like this, it is usable only as a pure template object definition that serves as a parent definition for child definitions. Trying to use such an abstract parent object on its own, by referring to it as a ref property of another object, or doing an explicit `GetObject()` with the parent object id, returns an error. The container's internal `PreInstantiateSingletons` method will completely ignore object definitions that are considered abstract.



Note

Application contexts pre-instantiate all singletons by default. Therefore it is important (at least for singleton objects) that if you have a (parent) object definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually (attempt to) pre-instantiate the abstract object.

5.8. Container extension points

The Spring container is essentially nothing more than an advanced factory capable of maintaining a registry of different

objects and their dependencies. The `IObjFactory` enables you to read object definitions and access them using the object factory. When using just the `IObjFactory` you would create an instance of one and then read in some object definitions in the XML format as follows:

```

[C#]
IResource input = new FileSystemResource ("objects.xml");
XmlObjectFactory factory = new XmlObjectFactory(input);

```

That is pretty much it. Using `GetObject(string)` (or the more concise indexer method `factory ["string"]`) you can retrieve instances of your objects...

```

[C#]
object foo = factory.GetObject ("foo"); // gets the object defined as 'foo'
object bar = factory ["bar"];           // same thing, just using the indexer

```

You'll get a reference to the same object if you defined it as a singleton (the default) or you'll get a new instance each time if you set the `singleton` property of your object definition to *false*.

```

<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary"/>
<object id="anotherObject" type="Examples.ExampleObject, ExamplesLibrary" singleton="false"/>

```

```

[C#]
object one = factory ["exampleObject"]; // gets the object defined as 'exampleObject'
object two = factory ["exampleObject"];
Console.WriteLine (one == two)           // prints 'true'
object three = factory ["anotherObject"]; // gets the object defined as 'anotherObject'
object four = factory ["anotherObject"];
Console.WriteLine (three == four);       // prints 'false'

```

The client-side view of the `IObjFactory` is surprisingly simple. The `IObjFactory` interface has only seven methods (and the aforementioned indexer) for clients to call:

- `bool ContainsObject(string)`: returns true if the `IObjectFactory` contains an object definition that matches the given name.
- `object GetObject(string)`: returns an instance of the object registered under the given name. Depending on how the object was configured by the `IObjectFactory` configuration, either a singleton (and thus shared) instance or a newly created object will be returned. An `ObjectsException` will be thrown when either the object could not be found (in which case it'll be a `NoSuchObjectDefinitionException`), or an exception occurred while instantiated and preparing the object.
- `Object this [string]`: this is the indexer for the `IObjectFactory` interface. It functions in all other respects in exactly the same way as the `GetObject(string)` method. The rest of this documentation will always refer to the `GetObject(string)` method, but be aware that you can use the indexer anywhere that you can use the `GetObject(string)` method.
- `Object GetObject(string, Type)`: returns an object, registered under the given name. The object returned will be cast to the given `Type`. If the object could not be cast, corresponding exceptions will be thrown (`ObjectNotOfRequiredTypeException`). Furthermore, all rules of the `GetObject(string)` method apply (see above).
- `bool IsSingleton(string)`: determines whether or not the object definition registered under the given name is a singleton or a prototype. If the object definition corresponding to the given name could not be found, an exception will be thrown (`NoSuchObjectDefinitionException`)
- `string[] GetAliases(string)`: returns the aliases for the given object name, if any were defined in the `IObjectDefinition`.
- `void ConfigureObject(object target)`: Injects dependencies into the supplied target instance. The name of the abstract object definition is the `System.Type.FullName` of the target instance. This method is typically used when objects are instantiated outside the control of a developer, for example when ASP.NET instantiates web controls and when a WinForms application creates `UserControls`.
- `void ConfigureObject(object target, string name)`: Offers the same functionality as the previously listed `Configure` method but uses a named object definition instead of using the type's full name.

A sub-interface of `IObjectFactory`, `IConfigurableObjectFactory` adds some convenient methods such as

- `void RegisterSingleton(string name, object objectInstance)`: Register the given existing object as singleton in the object factory under the given object name.
- `void RegisterAlias(string name, string theAlias)`: Given an object name, create an alias.

Check the SDK docs for additional details on `IConfigurableObjectFactory` methods and properties and the full `IObjectFactory` class hierarchy.

5.8.1. Obtaining an `IFactoryObject`, not its product

Sometimes there is a need to ask an `IObjectFactory` for an actual `IFactoryObject` instance itself, not the object it produces. This may be done by prepending the object id with `&` when calling the `GetObject` method of the `IObjectFactory` and `IApplicationContext` interfaces. So for a given `IFactoryObject` with an id `myObject`, invoking `GetObject("myObject")` on the `IObjectFactory` will return the product of the `IFactoryObject`, but invoking `GetObject("&myObject")` will return the `IFactoryObject` instance itself.

5.9. Container extension points

The IoC component of the Spring Framework has been designed for extension. There is typically no need for an application developer to subclass any of the various `IObjectFactory` or `IApplicationContext` implementation classes. The Spring IoC container can be infinitely extended by plugging in implementations of special integration interfaces. The next few sections are devoted to detailing all of these various integration interfaces.

5.9.1. Customizing objects with `IOjectPostProcessors`

The first extension point that we will look at is the `Spring.Objects.Factory.Config.IObjectPostProcessor` interface. This interface defines a number of callback methods that you as an application developer can implement in order to provide your own (or override the containers default) instantiation logic, dependency-resolution logic, and so forth. If you want to do some custom logic after the Spring container has finished instantiating, configuring and otherwise initializing an object, you can plug in one or more `IObjectPostProcessor` implementations.

You can configure multiple `IObjectPostProcessors` if you wish. You can control the order in which these `IObjectPostProcessor` execute by setting the 'Order' property (you can only set this property if the `IObjectPostProcessor` implements the `IOrdered` interface; if you write your own `IObjectPostProcessor` you should consider implementing the `IOrdered` interface too); consult the SDK docs for the `IObjectPostProcessor` and `IOrdered` interfaces for more details.



Note

`IObjectPostProcessor` operate on object instances; that is to say, the Spring IoC container will have instantiated a object instance for you, and then `IObjectPostProcessors` get a chance to do their stuff. If you want to change the actual object definition (that is the recipe that defines the object), then you rather need to use a `IObjectFactoryPostProcessor` (described below in the section entitled Customizing configuration metadata with `IObjectFactoryPostProcessors`).

Also, `IObjectPostProcessors` are scoped per-container. This is only relevant if you are using container hierarchies. If you define a `IObjectPostProcessor` in one container, it will only do its stuff on the objects in that container. Objects that are defined in another container will not be post-processed by `IObjectPostProcessors` in another container, even if both containers are part of the same hierarchy.

The `Spring.Objects.Factory.Config.IObjectPostProcessor` interface, which consists of two callback methods shown below.

```
object PostProcessBeforeInitialization(object instance, string name);

object PostProcessAfterInitialization(object instance, string name);
```

When such a class is registered as a post-processor with the container, for each object instance that is created by the container,(see below for how this registration is effected), for each object instance that is created by the container, the post-processor will get a callback from the container both *before* any initialization methods (such as the `AfterPropertiesSet` method of the `IInitializingObject` interface and any declared `init` method) are called, and also afterwards. The post-processor is free to do what it wishes with the object, including ignoring the callback completely. An object post-processor will typically check for marker interfaces, or do something such as wrap an object with a proxy. Some Spring.NET AOP infrastructure classes are implemented as object post-processors as they do this proxy-wrapping logic.

Other extensions to the `IObjectPostProcessors` interface are `IInstantiationAwareObjectPostProcessor` and `IDestructionAwareObjectPostProcessor` defined below

```
public interface IInstantiationAwareObjectPostProcessor : IObjectPostProcessor
{
    object PostProcessBeforeInstantiation(Type objectType, string objectName);

    bool PostProcessAfterInstantiation(object objectInstance, string objectName);

    IPropertyValues PostProcessPropertyValues(IPropertyValues pvs, PropertyInfo[] pis, object objectInstance, string objectName);
}
```

```
public interface IDestructionAwareObjectPostProcessor : IObjectPostProcessor
{
    void PostProcessBeforeDestruction (object instance, string name);
}
```

The `PostProcessBeforeInstantiation` callback method is called right before the container creates the object. If the object returned by this method is not null then the default instantiation behavior of the container is short circuited. The returned object is the one registered with the container and no other `IObjectPostProcessor` callbacks will be invoked on it. This mechanism is useful if you would like to expose a proxy to the object instead of the actual target object. The `PostProcessAfterInstantiation` callback method is called after the object has been instantiated but before Spring performs property population based on explicit properties or autowiring. A return value of false would short circuit the standard Spring based property population. The callback method `PostProcessPropertyValues` is called after Spring collects all the property values to apply to the object, but before they are applied. This gives you the opportunity to perform additional processing such as making sure that a property is set to a value if it contains a `[Required]` attribute or to perform attribute based wiring, i.e. adding the attribute `[Inject("objectName")]` on a property. Both of these features are scheduled to be included in Spring .12.

The `IDestructionAwareObjectPostProcessor` callback contains a single method, `PostProcessBeforeDestruction`, which is called before a singleton's destroy method is invoked.

It is important to know that the `IObjectFactory` treats object post-processors slightly differently than the `IApplicationContext`. An `IApplicationContext` will automatically detect any objects which are deployed into it that implement the `IObjectPostProcessor` interface, and register them as post-processors, to be then called appropriately by the factory on object creation. Nothing else needs to be done other than deploying the post-processor in a similar fashion to any other object. On the other hand, when using plain `IObjectFactories`, object post-processors have to manually be explicitly registered, with a code sequence such as...

```
ConfigurableObjectFactory factory = new .....; // create an IObjectFactory
... // now register some objects
// now register any needed IObjectPostProcessors
MyObjectPostProcessor pp = new MyObjectPostProcessor();
factory.AddObjectPostProcessor(pp);
// now start using the factory
...
```

This explicit registration step is not convenient, and this is one of the reasons why the various `IApplicationContext` implementations are preferred above plain `IObjectFactory` implementations in the vast majority of Spring-backed applications, especially when using `IObjectPostProcessors`.



Note

`IObjectPostProcessors` and AOP auto-proxying

Classes that implement the `IObjectPostProcessor` interface are special, and so they are treated differently by the container. All `IObjectPostProcessors` and their directly referenced object will be instantiated on startup, as part of the special startup phase of the `IApplicationContext`, then all those `IObjectPostProcessors` will be registered in a sorted fashion - and applied to all further objects. Since AOP auto-proxying is implemented as a `IObjectPostProcessor` itself, no `IObjectPostProcessors` or directly referenced objects are eligible for auto-proxying (and thus will not have aspects 'woven' into them). For any such object, you should see an info log message: "Object 'foo' is not eligible for getting processed by all `IObjectPostProcessors` (for example: not eligible for auto-proxying)".

5.9.1.1. Example: Hello World, IObjectPostProcessor-style

This first example is hardly compelling, but serves to illustrate basic usage. All we are going to do is code a custom `IObjectPostProcessor` implementation that simply invokes the `ToString()` method of each object as it is created by the container and prints the resulting string to the system console. Yes, it is not hugely useful, but serves to get the basic concepts across before we move into the second example which is actually useful. The basis of the example is the `MovieFinder` quickstart that is included with the Spring.NET distribution.

Find below the custom `IObjectPostProcessor` implementation class definition

```
using System;
using Spring.Objects.Factory.Config;

namespace Spring.IocQuickStart.MovieFinder
{
    public class TracingObjectPostProcessor : IObjectPostProcessor
    {
        public object PostProcessBeforeInitialization(object instance, string name)
        {
            return instance;
        }

        public object PostProcessAfterInitialization(object instance, string name)
        {
            Console.WriteLine("Object '" + name + "' created : " + instance.ToString());
            return instance;
        }
    }
}
```

And the following configuration

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net" >
    <description>An example that demonstrates simple IoC features.</description>

    <object id="MyMovieLister"
            type="Spring.IocQuickStart.MovieFinder.MovieLister, Spring.IocQuickStart.MovieFinder">
        <property name="movieFinder" ref="MyMovieFinder"/>
    </object>

    <object id="MyMovieFinder"
            type="Spring.IocQuickStart.MovieFinder.SimpleMovieFinder, Spring.IocQuickStart.MovieFinder"/>

    <!-- when the above objects are instantiated, this custom IObjectPostProcessor implementation
         will output the fact to the system console -->
    <object type="Spring.IocQuickStart.MovieFinder.TracingObjectPostProcessor, Spring.IocQuickStart.MovieFinder"/>
</objects>
```

Notice how the `TracingObjectPostProcessor` is simply defined; it doesn't even have a name, and because it is a object it can be dependency injected just like any other object.

Find below a small driver script to exercise the above code and configuration;

```
IApplicationContext ctx =
    new XmlApplicationContext(
        "assembly://Spring.IocQuickStart.MovieFinder/Spring.IocQuickStart.MovieFinder/AppContext.xml");

MovieLister lister = (MovieLister) ctx.GetObject("MyMovieLister");
Movie[] movies = lister.MoviesDirectedBy("Roberto Benigni");
LOG.Debug("Searching for movie...");
foreach (Movie movie in movies)
{
    LOG.Debug(string.Format("Movie Title = '{0}', Director = '{1}'.", movie.Title, movie.Director));
}
LOG.Debug("MovieApp Done.");
```

The output of executing the above program will be:

```
INFO - Object 'Spring.IocQuickStart.MovieFinder.TracingObjectPostProcessor' is not eligible for being processed by all IObj
      (for example: not eligible for auto-proxying).
Object 'MyMovieFinder' created : Spring.IocQuickStart.MovieFinder.SimpleMovieFinder
Object 'MyMovieLister' created : Spring.IocQuickStart.MovieFinder.MovieLister
DEBUG - Searching for movie...
DEBUG - Movie Title = 'La vita e bella', Director = 'Roberto Benigni'.
DEBUG - MovieApp Done.
```

5.9.1.2. Example: the RequiredAttributeObjectPostProcessor

Using callback interfaces or annotations in conjunction with a custom `IObjPostProcessor` implementation is a common means of extending the Spring IoC container. The `[Required]` attribute in the `Spring.Objects.Factory.Attributes` namespace can be used to *mark* a property as being *'required-to-be-set'* (i.e. an setter property with this attribute applied must be configured to be dependency injected with a value), else an `ObjectInitializationException` will be thrown by the container at runtime.

The best way to illustrate the usage of this attribute is with an example.

```
public class MovieLister
{
    // the MovieLister has a dependency on the MovieFinder
    private IMovieFinder _movieFinder;

    // a setter property so that the Spring container can 'inject' a MovieFinder
    [Required]
    public IMovieFinder MovieFinder
    {
        {
            set { _movieFinder = value; }
        }
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

Hopefully the above class definition reads easy on the eye. Any and all `IObjDefinitions` for the `MovieLister` class must be provided with a value.

Let's look at an example of some XML configuraiton that will not pass validation.

```
<object id="MyMovieLister"
        type="Spring.IocQuickStart.MovieFinder.MovieLister, Spring.IocQuickStart.MovieFinder">
    <!-- whoops, no MovieFinder is set (and this property is [Required]) -->
</object>
```

At runtime the following message will be generated by the Spring container

```
Error creating context 'spring.root': Property 'MovieFinder' required for object 'MyMovieLister'
```

There is one last little piece of Spring configuration that is required to actually 'switch on' this behavior. Simply annotating the 'setter' properties of your classes is not enough to get this behavior. You need to enable a component that is aware of the `[Required]` attribute and that can process it appropriately.

This component is the `RequiredAttributeObjectPostProcessor` class. This is a special `IObjPostProcessor` implementation that is `[Required]`-aware and actually provides the 'blow up if this required property has not been set' logic. It is very easy to configure; simply drop the following object definition into your Spring XML configuration.

```
<object type="Spring.Objects.Factory.Attributes.RequiredAttributeObjectPostProcessor, Spring.Core"/>
```

Finally, one can configure an instance of the `RequiredAttributeObjectPostProcessor` class to look for another Attribute type. This is great if you already have your own `[Required]`-style attribute. Simply plug it into the definition of a `RequiredAttributeObjectPostProcessor` and you are good to go. By way of an example,

let's suppose you (or your organization / team) have defined an attribute called [Mandatory]. You can make a `RequiredAttributeObjectPostProcessor` instance [Mandatory]-aware like so:

```
<object type="Spring.Objects.Factory.Attributes.RequiredAttributeObjectPostProcessor, Spring.Core">
  <property name="RequiredAttributeType" value="MyApp.Attributes.MandatoryAttribute, MyApp"/>
</object>
```

5.9.2. Customizing configuration metadata with ObjectFactoryPostProcessors

The next extension point that we will look at is the `Spring.Objects.Factory.Config.IObjectFactoryPostProcessor`. The semantics of this interface are similar to the `IObjectPostProcessor`, with one major difference. `IObjectFactoryPostProcessors` operate on; that is to say, the Spring IoC container will allow `IObjectFactoryPostProcessors` to read the configuration metadata and potentially change it before the container has actually instantiated any other objects. By implementing this interface, you will receive a callback after the all the object definitions have been loaded into the IoC container but before they have been instantiated. The signature of the interface is shown below

```
public interface IObjectFactoryPostProcessor
{
    void PostProcessObjectFactory (IConfigurableListableObjectFactory factory);
}
```

You can configure multiple `IObjectFactoryPostProcessors` if you wish. You can control the order in which these `IObjectFactoryPostProcessors` execute by setting the 'Order' property (you can only set this property if the `IObjectFactoryPostProcessors` implements the `IOrdered` interface; if you write your own `IObjectFactoryPostProcessors` you should consider implementing the `IOrdered` interface too); consult the SDK docs for the `IObjectFactoryPostProcessors` and `IOrdered` interfaces for more details.



Note

If you want to change the actual object instances (the objects that are created from the configuration metadata), then you rather need to use a `IObjectObjectPostProcessor` (described above in the section entitled Customizing objects with `IObjectPostProcessors`).

Also, `IObjectFactoryPostProcessors` are scoped per-container. This is only relevant if you are using container hierarchies. If you define a `IObjectFactoryPostProcessors` in one container, it will only do its stuff on the object definitions in that container. Object definitions in another container will not be post-processed by `IObjectFactoryPostProcessors` in another container, even if both containers are part of the same hierarchy.

An object factory post-processor is executed manually (in the case of a `IObjectFactory`) or automatically (in the case of an `IApplicationContext`) to apply changes of some sort to the configuration metadata that defines a container. Spring.NET includes a number of pre-existing object factory post-processors, such as `PropertyResourceConfigurer` and `PropertyPlaceholderConfigurer`, both described below and `ObjectNameAutoProxyCreator`, which is very useful for wrapping other objects transactionally or with any other kind of proxy, as described later in this manual.

In an `IObjectFactory`, the process of applying an `IObjectFactoryPostProcessor` is manual, and will be similar to this:

```
XmlObjectFactory factory = new XmlObjectFactory(new FileSystemResource("objects.xml"));
// create placeholderconfigurer to bring in some property
// values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
```



```
cfg.setLocation(new FileSystemResource("ado.properties"));
// now actually do the replacement
cfg.PostProcessObjectFactory(factory);
```

This explicit registration step is not convenient, and this is one of the reasons why the various `IApplicationContext` implementations are preferred above plain `IObjFactory` implementations in the vast majority of Spring-backed applications, especially when using `IObjFactoryPostProcessors`.

An `IApplicationContext` will detect any objects which are deployed into it that implement the `IObjFactoryPostProcessor` interface, and automatically use them as object factory post-processors, at the appropriate time. Nothing else needs to be done other than deploying these post-processor in a similar fashion to any other object.



Note

Just as in the case of `IObjPostProcessors`, you typically don't want to have `IObjFactoryPostProcessors` marked as being lazily-initialized. If they are marked as such, then the Spring container will never instantiate them, and thus they won't get a chance to apply their custom logic. If you are using the 'default-lazy-init' attribute on the declaration of your `<objects/>` element, be sure to mark your various `IObjFactoryPostProcessor` object definitions with 'lazy-init="false"'.

5.9.2.1. Example: The `PropertyPlaceholderConfigurer`

The `PropertyPlaceholderConfigurer` is an excellent solution when you want to externalize a few properties from a file containing object definitions. This is useful to allow the person deploying an application to customize environment specific properties (for example database configuration strings, usernames, and passwords), without the complexity or risk of modifying the main XML definition file or files for the container.

Variable substitution is performed on simple property values, lists, dictionaries, sets, constructor values, object type name, and object names in runtime object references. Furthermore, placeholder values can also cross-reference other placeholders.

Note that `IApplicationContext`s are able to automatically recognize and apply objects deployed in them that implement the `IObjFactoryPostProcessor` interface. This means that as described here, applying a `PropertyPlaceholderConfigurer` is much more convenient when using an `IApplicationContext`. For this reason, it is recommended that users wishing to use this or other object factory postprocessors use an `IApplicationContext` instead of an `IObjFactory`.

In the example below a data access object needs to be configured with a database connection and also a value for the maximum number of results to return in a query. Instead of hard coding the values into the main Spring.NET configuration file we use place holders, in the NAnt style of `${variableName}`, and obtain their values from `NameValueSections` in the standard .NET application configuration file. The Spring.NET configuration file looks like:

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    </sectionGroup>
    <section name="DaoConfiguration" type="System.Configuration.NameValueSectionHandler"/>
    <section name="DatabaseConfiguration" type="System.Configuration.NameValueSectionHandler"/>
  </configSections>

  <DaoConfiguration>
    <add key="maxResults" value="1000"/>
  </DaoConfiguration>
```



```

<DatabaseConfiguration>
  <add key="connection.string" value="dsn=MyDSN;uid=sa;pwd=myPassword;" />
</DatabaseConfiguration>

<spring>
  <context>
    <resource uri="assembly://DaoApp/DaoApp/objects.xml" />
  </context>
</spring>

</configuration>

```

Notice the presence of two NameValueSections in the configuration file. These name value pairs will be referred to in the Spring.NET configuration file. In this example we are using an embedded assembly resource for the location of the Spring.NET configuration file so as to reduce the chance of accidental tampering in deployment. This Spring.NET configuration file is shown below.

```

<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.net
    http://www.springframework.net/xsd/spring-objects.xsd" >

  <object name="productDao" type="DaoApp.SimpleProductDao, DaoApp ">
    <property name="maxResults" value="{maxResults}" />
    <property name="dbConnection" ref="myConnection" />
  </object>

  <object name="myConnection" type="System.Data.Odbc.OdbcConnection, System.Data">
    <property name="connectionstring" value="{connection.string}" />
  </object>

  <object name="appConfigPropertyHolder"
    type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">

    <property name="configSections">
      <value>DaoConfiguration,DatabaseConfiguration</value>
    </property>

  </object>
</objects>

```

The values of `{maxResults}` and `{connection.string}` match the key names used in the two NameValueSectionHandlers `DaoConfiguration` and `DatabaseConfiguration`. The `PropertyPlaceholderConfigurer` refers to these two sections via a comma delimited list of section names in the `configSections` property. If you are using section groups, prefix the section group name, for example `myConfigSection/DaoConfiguraiton`.

The `PropertyPlaceholderConfigurer` class also supports retrieving name value pairs from other `IResource` locations. These can be specified using the `Location` and `Locations` properties of the `PropertyPlaceHolderConfigurer` class.

If there are properties with the same name in different resource locations the default behavior is that the last property processed overrides the previous values. This behavior is controlled by the `LastLocationOverrides` property. `True` enables overriding while `false` will append the values as one would normally expect using `NameValueCollection.Add`.



Note

In an ASP.NET environment you must specify the full, four-part name of the assembly when using a `NameValueFileSectionHandler`

```
<section name="hibernateConfiguration"
```

```
type="System.Configuration.NameValueFileSectionHandler, System,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
```

5.9.2.1.1. Type, Ref, and Expression substitution

The `PropertyPlaceholderConfigurer` can be used to substitute type names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:

```
<object id="MyMovieFinder" type="${custom.moviefinder.type}"/>
```

If the class is unable to be resolved at runtime to a valid type, resolution of the object will fail once it is about to be created (which is during the `PreInstantiateSingletons()` phase of an `ApplicationContext` for a non-lazy-init object.)

Similarly you can replace 'ref' and 'expression' metadata, as shown below

```
<object id="TestObject" type="Simple.TestObject, MyAssembly">
  <property name="age" expression="${ageExpression}"/>
  <property name="spouse" ref="${spouse-ref}"/>
</object>
```

5.9.2.1.2. Replacement with Environment Variables

You may also use the value environment variables to replace property placeholders. The use of environment variables is controlled via the property `EnvironmentVariableMode`. This property is an enumeration of the type `EnvironmentVariablesMode` and has three values, `Never`, `Fallback`, and `Override`. `Fallback` is the default value and will resolve a property placeholder if it was not already done so via a value from a resource location. `Override` will apply environment variables before applying values defined from a resource location. `Never` will, quite appropriately, disable environment variable substitution. An example of how the `PropertyPlaceholderConfigurer` XML is modified to enable override usage is shown below

```
<object name="appConfigPropertyHolder"
  type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">
  <property name="configSections" value="DaoConfiguration,DatabaseConfiguration"/>
  <property name="EnvironmentVariableMode" value="Override"/>
</object>
</objects>
```

5.9.2.2. Example: The `PropertyOverrideConfigurer`

The `PropertyOverrideConfigurer`, another object factory post-processor, is similar to the `PropertyPlaceholderConfigurer`, but in contrast to the latter, the original definitions can have default values or no values at all for object properties. If an overriding configuration file does not have an entry for a certain object property, the default context definition is used.

Note that the object factory definition is *not* aware of being overridden, so it is not immediately obvious when looking at the XML definition file that the override configurer is being used. In case that there are multiple `PropertyOverrideConfigurer` instances that define different values for the same object property, the last one will win (due to the overriding mechanism).

The example usage is similar to when using `PropertyPlaceholderConfigurer` except that the key name refers to the name given to the object in the Spring.NET configuration file and is suffixed via 'dot' notation with the name of the property. For example, if the application configuration file is

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    </sectionGroup>
```

```

<section name="DaoConfigurationOverride" type="System.Configuration.NameValueSectionHandler"/>
</configSections>

<DaoConfigurationOverride>
  <add key="productDao.maxResults" value="1000"/>
</DaoConfigurationOverride>

<spring>
  <context>
    <resource uri="assembly://DaoApp/DaoApp/objects.xml"/>
  </context>
</spring>

</configuration>

```

Then the value of 1000 will be used to overlay the value of 2000 set in the Spring.NET configuration file shown below

```

<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects.xsd" >

  <object name="productDao" type="PropPlayApp.SimpleProductDao, PropPlayApp " >
    <property name="maxResults" value="2000"/>
    <property name="dbConnection" ref="myConnection"/>
    <property name="log" ref="daoLog"/>
  </object>

  <object name="daoLog" type="Spring.Objects.Factory.Config.LogFactoryObject, Spring.Core">
    <property name="logName" value="DAOLogger"/>
  </object>

  <object name="myConnection" type="System.Data.Odbc.OdbcConnection, System.Data">
    <property name="connectionstring">
      <value>dsn=MyDSN;uid=sa;pwd=myPassword;</value>
    </property>
  </object>

  <object name="appConfigPropertyOverride" type="Spring.Objects.Factory.Config.PropertyOverrideConfigurer, Spring.Core">
    <property name="configSections">
      <value>DaoConfigurationOverride</value>
    </property>
  </object>

</objects>

```

5.9.2.3. IVariableSource

The IVariableSource is the base interface for providing the ability to get the value of property placeholders (name-value) pairs from a variety of sources. Out of the box, Spring.NET supports a number of variable sources that allow users to obtain variable values from .NET config files, java-style property files, environment variables, command line arguments and the registry and the new connection strings configuration section in .NET 2.0. The list of implementing classes is listed below. Please refer to the SDK documentation for more information.

- ConfigSectionVariableSource
- PropertyFileVariableSource
- EnvironmentVariableSource
- CommandLineArgsVariableSource
- RegistryVariableSource
- SpecialFolderVariableSource
- ConnectionStringsVariableSource

You use this by defining an instance of `Spring.Objects.Factory.Config.VariablePlaceholderConfigurer` in your configuration and set the property `VariableSource` to a single `IVariableSource` instance or the list property `VariableSources` to a list of `IVariableSource` instances. In the case of the same property defined in multiple `IVariableSource` implementations, the first one in the list that contains the property value will be used.

```
<object type="Spring.Objects.Factory.Config.VariablePlaceholderConfigurer, Spring.Core">
  <property name="VariableSources">
    <list>
      <object type="Spring.Objects.Factory.Config.ConfigSectionVariableSource, Spring.Core">
        <property name="SectionNames" value="CryptedConfiguration" />
      </object>
    </list>
  </property>
</object>
```

The `IVariableSource` interface is shown below

```
public interface IVariableSource
{
    string ResolveVariable(string name);
}
```

This is a simple contract to implement if you should decide to create your own custom implementation. Look at the source code of the current implementations for some inspiration if you go that route. To register your own custom implementation, simply configure `VariablePlaceholderConfigurer` to refer to your class.

5.9.3. Customizing instantiation logic using `IFactoryObjects`

The `Spring.Objects.Factory.IFactoryObject` interface is to be implemented by objects that *are themselves factories*.

The `IFactoryObject` interface is a point of pluggability into the Spring IoC containers instantiation logic. If you have some complex initialization code that is better expressed in C# as opposed to a (potentially) verbose amount of XML, you can create your own `IFactoryObject`, write the complex initialization inside that class, and then plug your custom `IFactoryObject` into the container.

The `IFactoryObject` interface provides one method and two (read-only) properties:

- `object GetObject()`: has to return an instance of the object this factory creates. The instance can possibly be shared (depending on whether this factory provides singletons or prototypes).
- `bool IsSingleton`: has to return `true` if this `IFactoryObject` returns singletons, `false` otherwise.
- `Type ObjectType`: has to return either the object type returned by the `GetObject()` method or `null` if the type isn't known in advance.

`IFactoryObject`

The `IFactoryObject` concept and interface is used in a number of places within the Spring Framework. Some examples of its use is described in Section 5.3.9, “Setting a reference using the members of other objects and classes.” for the `PropertyRetrievingFactoryObject` and `FieldRetrievingFactoryObject`. An additional use of creating a custom `IFactoryObject` implementation is to retrieve an object from an embedded resource file and use it to set another objects dependency. An example of this is provided here [http://jira.springframework.org/browse/SPRNET-133#action_19743].

Finally, there is sometimes a need to ask a container for an actual `IFactoryObject` instance itself, not the object it produces. This may be achieved by prepending the object id with `'&'` (sans quotes) when calling the `GetObject` method of the `IObjectFactory` (including `IApplicationContext`). So for a given `IFactoryObject` with an id of

'myObject', invoking `GetObject("myObject")` on the container will return the product of the `IFactoryObject`, but invoking `GetObject("&myObject")` will return the `IFactoryObject` instance itself.

5.9.3.1. IConfigurableFactoryObject

The `Spring.Objects.Factory.IConfigurableFactoryObject` interface inherits from `IFactoryObject` interface and adds the following property.

- `IOBJECTDefinition ProductTemplate` : Gets the template object definition that should be used to configure the instance of the object managed by this factory.

`IConfigurableFactoryObject` implementations you already have examples of in Section 27.3, "Client-side" are `WebServiceProxyFactory`.

5.10. The IApplicationContext

While the `Spring.Objects` namespace provides basic functionality for managing and manipulating objects, often in a programmatic way, the `Spring.Context` namespace introduces the `IApplicationContext` interface, which enhances the functionality provided by the `IOBJECTFactory` interface in a more *framework-oriented style*. Many users will use `ApplicationContext` in a completely declarative fashion, not even having to create it manually, but instead relying on support classes such as the .NET configuration section handlers such as `ContextHandler` and `WebContextHandler` together to declaratively define the `ApplicationContext` and retrieve it though a `ContextRegistry`. (Of course it is still possible to create an `IApplicationContext` Programatically).

The basis for the context module is the `IApplicationContext` interface, located in the `Spring.Context` namespace. Deriving from the `IOBJECTFactory` interface, it provides all the functionality of the `IOBJECTFactory`. To be able to work in a more framework-oriented fashion, using layering and hierarchical contexts, the `Spring.Context` namespace also provides the following functionality

- *Loading of multiple (hierarchical) contexts*, allowing some of them to be focused and used on one particular layer, for example the web layer of an application.
- *Access to localized resources* at the application level by implementing `IMessageSource`.
- *Uniform access to resources* that can be read in as an `InputStream`, such as URLs and files by implementing `IResourceLoader`
- *Loosely Coupled Event Propagation*. Publishers and subscribers of events do not have to be directly aware of each other as they register their interest indirectly through the application context.

5.10.1. IOBJECTFactory or IApplicationContext?

Short version: use an `IApplicationContext` unless you have a really good reason for not doing so. For those of you that are looking for slightly more depth as to the 'but why' of the above recommendation, keep reading.

As the `IApplicationContext` includes all the functionality the object factory via its inheritance of the `IOBJECTFactory` interface, it is generally recommended to be used over the `IOBJECTFactory` except for a few limited situations where memory consumption might be critical. This may become more important if the .NET Compact Framework is supported. The history of `IOBJECTFactory` comes from the Spring Java framework, where the use of Spring in Applets was a concern to reduce memory consumption. However, for most 'typical' enterprise applications and systems, the `IApplicationContext` is what you will want to use. Spring generally makes heavy use of the `IOBJECTPostProcessor` extension point (to effect proxying and suchlike), and if you are using just a plain `IOBJECTFactory` then a fair amount of support such as transactions and AOP will not take effect (at least not without some extra steps on your part), which could be confusing because nothing will actually be wrong with the configuration.

Find below a feature matrix that lists what features are provided by the `IObjectFactory` and `IApplicationContext` interfaces (and attendant implementations). The following sections describe functionality that `IApplicationContext` adds to the basic `IObjectFactory` capabilities in a lot more depth than the said feature matrix.)

Table 5.7. Feature Matrix

Feature	<code>IObjectFactory</code>	<code>IApplicationContext</code>
Object instantiation/wiring	Yes	Yes
Automatic <code>IObjectPostProcessor</code> registration	No	Yes
Automatic <code>IObjectFactoryPostProcessor</code> registration	No	Yes
Convenient <code>IMessageSource</code> access	No	Yes
<code>ApplicationEvent</code> publication	No	Yes
Singleton service locator style access	No	Yes
Declarative registration of custom resource protocol handler, XML Parsers for object definitions, and type aliases	No	Yes

5.11. Configuration of `IApplicationContext`

Well known locations in the .NET application configuration file are used to register resource handlers, custom parsers, type alias, and custom type converts in addition to the context and objects sections mentioned previously. A sample .NET application configuration file showing all these features is shown below. Each section requires the use of a custom configuration section handler. Note that the types shown for resource handlers and parsers are fictional.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">

      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />

      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
      <section name="resources" type="Spring.Context.Support.ResourceHandlersSectionHandler, Spring.Core"/>
      <section name="typeAliases" type="Spring.Context.Support.TypeAliasesSectionHandler, Spring.Core"/>
      <section name="typeConverters" type="Spring.Context.Support.TypeConvertersSectionHandler, Spring.Core"/>

    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
      <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
    </parsers>
  </spring>
</configuration>
```

```

</parsers>

<resources>
  <handler protocol="db" type="MyCompany.MyApp.Resources.MyDbResource"/>
</resources>

<context caseSensitive="false">
  <resource uri="config://spring/objects"/>
  <resource uri="db://user:pass@dbName/MyDefinitionsTable"/>
</context>

<typeAliases>
  <alias name="WebServiceExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web"/>
  <alias name="DefaultPointcutAdvisor" type="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop"/>
  <alias name="AttributePointcut" type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop"/>
  <alias name="CacheAttribute" type="Spring.Attributes.CacheAttribute, Spring.Core"/>
  <alias name="MyType" type="MyCompany.MyProject.MyNamespace.MyType, MyAssembly"/>
</typeAliases>

<typeConverters>
  <converter for="Spring.Expressions.IExpression, Spring.Core" type="Spring.Objects.TypeConverters.ExpressionConverter, MyAssembly"/>
  <converter for="MyTypeAlias" type="MyCompany.MyProject.Converters.MyTypeConverter, MyAssembly"/>
</typeConverters>

<objects xmlns="http://www.springframework.net">
  ...
</objects>

</spring>

</configuration>

```

The new sections are described below. The attribute `caseSensitive` allows the for both `IObjectFactory` and `IApplicationContext` implementations to not pay attention to the case of the object names. This is important in web applications so that ASP.NET pages can be resolved in a case independent manner. The default value is true.

5.11.1. Registering custom parsers

Instead of using the default XML schema that is generic in nature to define an object's properties and dependencies, you can create your own XML schema specific to an application domain. This has the benefit of being easier to type and getting XML intellisense for the schema being used. The downside is that you need to write code that will transform this XML into Spring object definitions. One would typically implement a custom parser by deriving from the class `ObjectsNamespaceParser` and overriding the methods `int ParseRootElement(XmlElement root, XmlResourceReader reader)` and `int ParseElement(XmlElement element, XmlResourceReader reader)`. Registering custom parsers outside of `App.config` will be addressed in a future release.

To register a custom parser register a section handler of the type `Spring.Context.Support.NamespaceParsersSectionHandler` in the `configSections` section of `App.config`. The parser configuration section contains one or more `<parser>` elements each with a `type` attribute. Below is an example that registers all the namespaces provided in Spring.



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in `App.config`. You will still need to register custom namespace parsers if you are writing your own.

```

<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other configuration section handler defined here -->
    </sectionGroup>
  </configSections>

```

```

    <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
  </sectionGroup>
</configSections>

<spring>
  <parsers>
    <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
    <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
    <parser type="Spring.Transaction.Config.TxNamespaceParser, Spring.Data" />
    <parser type="Spring.Validation.Config.ValidationNamespaceParser, Spring.Core" />
    <parser type="Spring.Remoting.Config.RemotingNamespaceParser, Spring.Services" />
  </parsers>
</spring>

</configuration>

```

You can also register custom parser programmatically using the `NamespaceParserRegistry`. Here is an example taken from the code used in the Transactions Quickstart application.

```

NamespaceParserRegistry.RegisterParser(typeof(DatabaseNamespaceParser));
NamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParser));
NamespaceParserRegistry.RegisterParser(typeof(AopNamespaceParser));

IApplicationContext context =
    new XmlApplicationContext("assembly://Spring.TxQuickStart.Tests/Spring.TxQuickStart/system-test-local-config.xml");

```

5.11.2. Registering custom resource handlers

Creating a custom resource handler means implementing the `IResource` interface. The base class `AbstractResource` is a useful starting point. Look at the Spring source for classes such as `FileSystemResource` or `AssemblyResource` for implementation tips. You can register your custom resource handler either within `App.config`, as shown in the program listing at the start of this section using a `.ResourceHandlersSectionHandler` or define an object of the type `Spring.Objects.Factory.Config.ResourceHandlerConfigurer` as you would any other Spring managed object. An example of the latter is shown below:

```

<object id="myResourceHandlers" type="Spring.Objects.Factory.Config.ResourceHandlersSectionHandler, Spring.Core">
  <property name="ResourceHandlers">
    <dictionary>
      <entry key="db" value="MyCompany.MyApp.Resources.MyDbResource, MyAssembly"/>
    </dictionary>
  </property>
</object>

```

5.11.3. Registering Type Aliases

Type aliases allow you to simplify Spring configuration file by replacing fully qualified type name with an alias for frequently used types. Aliases can be registered both within a config file and programatically and can be used anywhere in the context config file where a fully qualified type name is expected. Type aliases can also be defined for generic types.

One way to configure a type alias is to define them in a custom config section in the `Web/App.config` file for your application, as well as the custom configuration section handler. See the previous XML configuration listing for an example that makes an alias for the `WebServiceExporter` type. Once you have aliases defined, you can simply use them anywhere where you would normally specify a fully qualified type name:

```

<object id="MyWebService" type="WebServiceExporter">
  ...
</object>

<object id="cacheAspect" type="DefaultPointcutAdvisor">
  <property name="Pointcut">
    <object type="AttributePointcut">
      <property name="Attribute" value="CacheAttribute"/>
    </object>
  </property>
</object>

```



```

    </object>
  </property>
  <property name="Advice" ref="aspNetCacheAdvice"/>
</object>

```

To register a type alias register a section handler of the type `Spring.Context.Support.TypeAliasesSectionHandler` in the `configSections` section of `App.config`. The type alias configuration section contains one or more `<alias>` elements each with a name and a type attribute. Below is an example that registers the alias for `WebServiceExporter`

```

<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other configuration section handler defined here -->
      <section name="parsers" type="Spring.Context.Support.TypeAliasesSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <typeAliases>
      <alias name="WebServiceExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web"/>
    </typeAliases>
  </spring>

</configuration>

```

For an example showing type aliases for generic types see Section 5.2.6, “Object creation of generic types”.

Another way is to define an object of the type `Spring.Objects.Factory.Config.TypeAliasConfigurer` within the regular `<objects>` section of any standard Spring configuration file. This approach allows for more modularity in defining type aliases, for example if you can't access `App.config/Web.config`. An example of registration using a `TypeAliasConfigurer` is shown below

```

<object id="myTypeAlias" type="Spring.Objects.Factory.Config.TypeAliasConfigurer, Spring.Core">
  <property name="TypeAliases">
    <dictionary>
      <entry key="WebServiceExporter" value="Spring.Web.Services.WebServiceExporter, Spring.Web"/>
      <entry key="DefaultPointcutAdvisor" value="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop"/>
      <entry key="MyType" value="MyCompany.MyProject.MyNamespace.MyType, MyAssembly"/>
    </dictionary>
  </property>
</object>

```

5.11.4. Registering Type Converters

The standard .NET mechanism for specifying a type converter is to add a `TypeConverter` attribute to a type definition to specify the type of the Converter. This is the preferred way of defining type converters if you control the source code for the type that you want to define a converter for. However, this configuration section allows you to specify converters for the types that you don't control, and it also allows you to override some of the standard type converters, such as the ones that are defined for some of the types in the .NET Base Class Library.

You can specify the type converters in `App.config` by using `Spring.Context.Support.TypeConvertersSectionHandler` as shown before or define an object of the type `Spring.Objects.Factory.Config.CustomConverterConfigurer`. An example of registration using a `CustomConverterConfigurer` is shown below

```

<object id="myTypeConverters" type="Spring.Objects.Factory.Config.CustomConverterConfigurer, Spring.Core">
  <property name="CustomConverters">
    <dictionary>
      <entry key="System.Date" value="MyCompany.MyProject.MyNamespace.MyCustomDateConverter, MyAssembly"/>
    </dictionary>
  </property>
</object>

```

```
</object>
```

5.12. Added functionality of the `ApplicationContext`

As already stated in the previous section, the `ApplicationContext` has a couple of features that distinguish it from the `ObjectFactory`. Let us review them one-by-one.

5.12.1. Context Hierarchies

You can structure the configuration information of application context into hierarchies that naturally reflect the internal layering of your application. As an example, abstract object definitions may appear in a parent application context configuration file, possibly as an embedded assembly resource so as not to invite accidental changes.

```
<spring>
  <context>
    <resource uri="assembly://MyAssembly/MyProject/root-objects.xml"/>
    <context name="mySubContext">
      <resource uri="file://objects.xml"/>
    </context>
  </context>
</spring>
```

The nesting of `context` elements reflects the parent-child hierarchy you are creating. The nesting can be to any level though it is unlikely one would need a deep application hierarchy. The xml file must contain the `<objects>` as the root name. Another example of a hierarchy, but using sections in the application configuration file is shown below.

```
<configSections>
  <sectionGroup name="spring">
    <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    <sectionGroup name="child">
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </sectionGroup>
</configSections>

<spring>

  <context name="ParentContext">
    <resource uri="config://spring/objects"/>
    <context name="ChildContext">
      <resource uri="config://spring/child/objects"/>
    </context>
  </context>

  <objects xmlns="http://www.springframework.net">

    ...

  </objects>

  <child>
    <objects xmlns="http://www.springframework.net">

      ...

    </objects>
  </child>

</spring>
```

As a reminder, the `type` attribute of the `context` tag is optional and defaults to `Spring.Context.Support.XmlApplicationContext`. The name of the context can be used in conjunction with the service locator class, `ContextRegistry`, discussed in Section 5.15, “Service Locator access”

5.12.2. Using `IMessageSource`

The `IApplicationContext` interface extends an interface called `IMessageSource` and provides localization (i18n or internationalization) services for text messages and other resource data types such as images. This functionality makes it easier to use .NET's localization features at an application level and also offers some performance enhancements due to caching of retrieved resources. Together with the `NestingMessageSource`, capable of hierarchical message resolving, these are the basic interfaces Spring.NET provides for localization. Let's quickly review the methods defined there:

- `string GetMessage(string name):` retrieves a message from the `IMessageSource` and using `CurrentUICulture`.
- `string GetMessage(string name, CultureInfo cultureInfo):` retrieves a message from the `IMessageSource` using a specified culture.
- `string GetMessage(string name, params object[] args):` retrieves a message from the `IMessageSource` using a variable list of arguments as replacement values in the message. The `CurrentUICulture` is used to resolve the message.
- `string GetMessage(string name, CultureInfo cultureInfo, params object[] args):` retrieves a message from the `IMessageSource` using a variable list of arguments as replacement values in the message. The specified culture is used to resolve the message.
- `string GetMessage(string name, string defaultMessage, CultureInfo culture, params object[] arguments):` retrieves a message from the `IMessageSource` using a variable list of arguments as replacement values in the message. The specified culture is used to resolve the message. If no message can be resolved, the default message is used.
- `string GetMessage(IMessageSourceResolvable resolvable, CultureInfo culture) :` all properties used in the methods above are also wrapped in a class - the `MessageSourceResolvable`, which you can use in this method.
- `object GetResourceObject(string name):` Get a localized resource object, i.e. Icon, Image, etc. given the resource name. The `CurrentUICulture` is used to resolve the resource object.
- `object GetResourceObject(string name, CultureInfo cultureInfo):` Get a localized resource object, i.e. Icon, Image, etc. given the resource name. The specified culture is used to resolve the resource object.
- `void ApplyResources(object value, string objectName, CultureInfo cultureInfo):` Uses a `ComponentResourceManager` to apply resources to all object properties that have a matching key name. Resource key names are of the form `objectName.propertyName`

When an `IApplicationContext` gets loaded, it automatically searches for an `IMessageSource` object defined in the context. The object has to have the name `messageSource`. If such an object is found, all calls to the methods described above will be delegated to the message source that was found. If no message source was found, the `IApplicationContext` checks to see if it has a parent containing a similar object, with a similar name. If so, it uses that object as the `IMessageSource`. If it can't find any source for messages, an empty `StaticMessageSource` will be instantiated in order to be able to accept calls to the methods defined above.



Fallback behavior

The fallback rules for localized resources seem to have a bug that is fixed by applying Service Pack 1 for .NET 1.1. This affects the use of `IMessageSource.GetMessage` methods that specify `CultureInfo`. The core of the issue in the .NET BCL is the method `ResourceManager.GetObject` that accepts `CultureInfo`.

Spring.NET provides two `IMessageSource` implementations. These are `ResourceSetMessageSource` and `StaticMessageSource`. Both implement `IHierarchicalMessageSource` to resolve messages hierarchically. The

`StaticMessageSource` is hardly ever used but provides programmatic ways to add messages to the source. The `ResourceSetMessageSource` is more interesting and an example is provided for in the distribution and discussed more extensively in the Chapter 35, *IoC Quickstarts* section. The `ResourceSetMessageSource` is configured by providing a list of `ResourceManagers`. When a message code is to be resolved, the list of `ResourceManagers` is searched to resolve the code. For each `ResourceManager` a `ResourceSet` is retrieved and asked to resolve the code. Note that this search does not replace the standard hub-and-spoke search for localized resources. The `ResourceManagers` list specifies the multiple 'hubs' where the standard search starts.

```
<object name="messageSource" type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
  <property name="resourceManagers">
    <list>
      <value>Spring.Examples.AppContext.MyResource, Spring.Examples.AppContext</value>
    </list>
  </property>
</object>
```

You can specify the arguments to construct a `ResourceManager` as a two part string value containing the base name of the resource and the assembly name. This will be converted to a `ResourceManager` via the `ResourceManagerConverter` `TypeConverter`. This converter can be similarly used to set a property on any object that is of the type `ResourceManager`. You may also specify an instance of the `ResourceManager` to use via an object reference. The convenience class `Spring.Objects.Factory.Config.ResourceManagerFactoryObject` can be used to conveniently create an instance of a `ResourceManager`.

```
<object name="myResourceManager" type="Spring.Objects.Factory.Config.ResourceManagerFactoryObject, Spring.Core">
  <property name="baseName">
    <value>Spring.Examples.AppContext.MyResource</value>
  </property>
  <property name="assemblyName">
    <value>Spring.Examples.AppContext</value>
  </property>
</object>
```

In application code, a call to `GetMessage` will retrieve a properly localized message string based on a code value. Any arguments present in the retrieved string are replaced using `String.Format` semantics. The `ResourceManagers`, `ResourceSets` and retrieved strings are cached to provide quicker lookup performance. The key 'HelloMessage' is contained in the resource file with a value of `Hello {0} {1}`. The following call on the application context will return the string `Hello Mr. Anderson`. *Note* that the caching of `ResourceSets` is via the concatenation of the `ResourceManager` base name and the `CultureInfo` string. This combination must be unique.

```
string msg = ctx.GetMessage("HelloMessage",
    new object[] { "Mr.", "Anderson" },
    CultureInfo.CurrentCulture );
```

It is possible to chain the resolution of messages by passing arguments that are themselves messages to be resolved giving you greater flexibility in how you can structure your message resolution. This is achieved by passing as an argument a class that implements `IMessageResolvable` instead of a string literal. The convenience class `DefaultMessageResolvable` is available for this purpose. As an example if the resource file contains a key name `error.required` that has the value `'{0} is required {1}'` and another key name `field.firstname` with the value `'First name'`. The following code will create the string `'First name is required dude!'`

```
string[] codes = {"field.firstname"};
DefaultMessageResolvable dmr = new DefaultMessageResolvable(codes, null);
ctx.GetMessage("error.required",
    new object[] { dmr, "dude!" },
    CultureInfo.CurrentCulture );
```

The examples directory in the distribution contains an example program, `Spring.Examples.AppContext`, that demonstrates usage of these features.

The `IMessageSourceAware` interface can also be used to acquire a reference to any `IMessageSource` that has been defined. Any object that is defined in an `IApplicationContext` that implements the `IMessageSourceAware` interface will be injected with the application context's `IMessageSource` when it (the object) is being created and configured.

5.12.3. Using resources within Spring.NET

A lot of applications need to access resources. Resources here, might mean files, but also news feeds from the Internet or normal web pages. Spring.NET provides a clean and transparent way of accessing resources in a protocol independent way. The `IApplicationContext` has a method (`GetResource(string)`) to take care of this. Refer to Section 7.1, “Introduction” for more information on the string format to use and the `IResource` abstraction in general.

5.12.4. Loosely coupled events

The Eventing Registry allows developers to utilize a loosely coupled event wiring mechanism. By decoupling the event publication and the event subscription, most of the mundane event wiring is handled by the IoC container. Event publishers can publish their event to a central registry, either all of their events or a subset based on criteria such as delegate type, name, return value, etc... Event subscribers can choose to subscribe to any number of published events. Subscribers can subscribe to events based on the type of object exposing them, allowing one subscriber to handle all events of a certain type without regards to how many different instances of that type are created.

The `Spring.Objects.Events.IEventRegistry` interface represents the central registry and defines publish and subscribe methods.

- `void PublishEvents(object sourceObject)`: publishes all events of the source object to subscribers that implement the correct handler methods.
- `void Subscribe(object subscriber)`: The subscriber receives all events from the source object for which it has matching handler methods.
- `void Subscribe(object subscriber, Type targetSourceType)`: The subscriber receives all events from a source object of a particular type for which it has matching handler methods.
- `void Unsubscribe(object subscriber)`: Unsubscribe all events from the source object for which it has matching handler methods.
- `void Unsubscribe(object subscriber, Type targetSourceType)`: Unsubscribe all events from a source object of a particular type for which it has matching handler methods.

`IApplicationContext` implements this interface and delegates the implementation to an instance of `Spring.Objects.Events.Support.EventRegistry`. You are free to create and use as many `EventRegistries` as you like but since it is common to use only one in an application, `IApplicationContext` provides convenient access to a single instance.

Within the `example/Spring/Spring.Examples.EventRegistry` directory you will find an example on how to use this functionality. When you open up the project, the most interesting file is the `EventRegistryApp.cs` file. This application loads a set of object definitions from the application configuration file into an `IApplicationContext` instance. From there, three objects are loaded up: one publisher and two subscribers. The publisher publishes its events to the `IApplicationContext` instance:

```
// Create the Application context using configuration file
IApplicationContext ctx = ContextRegistry.GetContext();

// Gets the publisher from the application context
MyEventPublisher publisher = (MyEventPublisher)ctx.GetObject("MyEventPublisher");
```

```
// Publishes events to the context.
ctx.PublishEvents( publisher );
```

One of the two subscribers subscribes to all events published to the `IApplcationContext` instance, using the publisher type as the filter criteria.

```
// Gets first instance of subscriber
MyEventSubscriber subscriber = (MyEventSubscriber)ctx.GetObject("MyEventSubscriber");

// Gets second instance of subscriber
MyEventSubscriber subscriber2 = (MyEventSubscriber)ctx.GetObject("MyEventSubscriber");

// Subscribes the first instance to the any events published by the type MyEventPublisher
ctx.Subscribe( subscriber, typeof(MyEventPublisher) );
```

This will wire the first subscriber to the original event publisher. Anytime the event publisher fires an event, (`publisher.ClientMethodThatTriggersEvent1();`) the first subscriber will handle the event, but the second subscriber will not. This allows for selective subscription, regardless of the original prototype definition.

5.12.5. Event notification from `IApplcationContext`

Event handling in the `IApplcationContext` is provided through the `IApplcationEventListener` interface that contains the single method `void HandleApplicationEvent(object source, ApplicationEventArgs applicationEventArgs)`. Classes that implement the `IApplcationEventListener` interface are automatically registered as a listener with the `IApplcationContext`. Publishing an event is done via the context's `PublishEvent(ApplicationEventArgs eventArgs)` method. This implementation is based on the traditional *Observer* design pattern.

The event argument type, `ApplicationEventArgs`, adds the time of the event firing as a property. The derived class `ContextEventArgs` is used to notify observers on the lifecycle events of the application context. It contains a property `ContextEvent Event` that returns the enumeration `Refreshed` or `Closed`. The `Refreshed` enumeration value indicated that the `IApplcationContext` was either initialized or refreshed. Initialized here means that all objects are loaded, singletons are pre-instantiated and the `IApplcationContext` is ready for use. The `Closed` is published when the `IApplcationContext` is closed using the `Dispose()` method on the `IConfigurableApplicationContext` interface. `Closed` here means that singletons are destroyed.

Implementing custom events can be done as well. Simply call the `PublishEvent` method on the `IApplcationContext`, specifying a parameter which is an instance of your custom event argument subclass.

Let's have a look at an example. First, the `IApplcationContext`:

```
<object id="emailer" type="Example.EmailObject">
  <property name="blackList">
    <list>
      <value>black@list.org</value>
      <value>white@list.org</value>
      <value>john@doe.org</value>
    </list>
  </property>
</object>

<object id="blackListListener" type="Example.BlackListNotifier">
  <property name="notificationAddress">
    <value>spam@list.org</value>
  </property>
</object>
```

and then, the actual objects:

```
public class EmailObject : IApplcationContextAware {

  // the blacklist
```

```

private IList blackList;

public IList BlackList
{
    set { this.blackList = value; }
}

public IApplicationContext ApplicationContext
{
    set { this.ctx = value; }
}

public void SendEmail(string address, string text) {
    if (blackList.Contains(address))
    {
        BlackListEvent evt = new BlackListEvent(address, text);
        ctx.publishEvent(evt);
        return;
    }
    // send email...
}

public class BlackListNotifier : IApplicationEventListener
{
    // notification address
    private string notificationAddress;

    public string NotificationAddress
    {
        set { this.notificationAddress = value; }
    }

    public void HandleApplicationEvent(ApplicationEvent evt)
    {
        if (evt instanceof BlackListEvent)
        {
            // notify appropriate person
        }
    }
}

```

5.13. Customized behavior in the ApplicationContext

The `IObjFactory` already offers a number of mechanisms to control the lifecycle of objects deployed in it (such as marker interfaces like `IInitializingObject` and `System.IDisposable`, their configuration only equivalents such as `init-method` and `destroy-method`) attributes in an `XmlObjFactory` configuration, and object post-processors. In an `IApplicationContext`, all of these still work, but additional mechanisms are added for customizing behavior of objects and the container.

5.13.1. The `IApplicationContextAware` marker interface

All marker interfaces available with `ObjectFactories` still work. The `IApplicationContext` does add one extra marker interface which objects may implement, `IApplicationContextAware`. An object which implements this interface and is deployed into the context will be called back on creation of the object, using the interface's `ApplicationContext` property, and provided with a reference to the context, which may be stored for later interaction with the context.

5.13.2. The `IObjPostProcessor`

Object post-processors are classes which implement the `Spring.Objects.Factory.Config.IObjPostProcessor` interface, have already been mentioned. It is worth mentioning again here though, that post-processors are much more convenient to use in `IApplicationContexts`

than in plain `IObjecTFactory` instances. In an `IApplicationContext`, any deployed object which implements the above marker interface is automatically detected and registered as an object post-processor, to be called appropriately at creation time for each object in the factory.

5.13.3. The `IObjecTFactoryPostProcessor`

Object factory post-processors are classes which implement the `Spring.Objects.Factory.Config.IObjecTFactoryPostProcessor` interface, have already been mentioned. It is worth mentioning again here though, that object factory post-processors are much more convenient to use in `IApplicationContexts`. In an `IApplicationContext`, any deployed object which implements the above marker interface is automatically detected as an object factory post-processor, to be called at the appropriate time.

5.13.4. The `PropertyPlaceholderConfigurer`

The `PropertyPlaceholderConfigurer` has already been described in the context of its use within an `IObjecTFactory`. It is worth mentioning here though, that it is generally more convenient to use it with an `IApplicationContext`, since the context will automatically recognize and apply any object factory post-processors, such as this one, when they are simply deployed into it like any other object. There is no need for a manual step to execute it.

5.14. Configuration of `ApplicationContext` without using XML

The class `GenericApplicationContext` can be used as a basis for creating an `IApplicationContext` implementation that read the container metadata from sources other than XML. This could be by scanning objects in a .DLL for known attributes or a scripting language that leverages a DSL to create terse `IObjecTDefinitions`. There is a class, `Spring.Objects.Factory.Support.ObjectDefinitionBuilder` offers some convenience methods for creating an `IObjecTDefinition` in a less verbose manner than using the `RootObjectDefinition` API. The following shows how to configure the `GenericApplicationContext` to read from XML, just so show familiar API usage

```
GenericApplicationContext ctx = new GenericApplicationContext();
XmlObjectDefinitionReader reader = new XmlObjectDefinitionReader(ctx);
reader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Spring.Context.Support/contextB.xml");
reader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Spring.Context.Support/contextC.xml");
reader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Spring.Context.Support/contextA.xml");
ctx.Refresh();
```

The implementation of `IObjecTDefinitionReader` is responsible for creating the configuration metadata, i.e., implementations of `RootObjectDefinition`, etc. Note a web version of this application class has not yet been implemented.

An example, with a *yet to be created* DLL scanner, that would get configuration metadata from the .dll named `MyAssembly.dll` located in the runtime path, would look something like this

```
GenericApplicationContext ctx = new GenericApplicationContext();
ObjectDefinitionScanner scanner = new ObjectDefinitionScanner(ctx);
scanner.scan("MyAssembly.dll");
ctx.refresh();
```

Refer to the Spring API documentation for more information.

5.15. Service Locator access

The majority of the code inside an application is best written in a Dependency Injection (Inversion of Control) style, where that code is served out of an `IObjecTFactory` or `IApplicationContext` container, has its own

dependencies supplied by the container when it is created, and is completely unaware of the container. However, there is sometimes a need for singleton (or quasi-singleton) style access to an `IObjFactory` or `IApplicationContext`. For example, third party code may try to construct a new object directly without the ability to force it to get these objects out of the `IObjFactory`. Similarly, nested user control components in a WinForms application are created inside the generated code in `InitializeComponent`. If this user control would like to obtain references to objects contained in the container it can use the service locator style approach and 'reach out' from inside the code to obtain the object it requires. (Note support for DI in WinForms is under development.)

The `Spring.Context.Support.ContextRegistry` class allows you to obtain a reference to an `IApplicationContext` via a static locator method. The `ContextRegistry` is initialized when creating an `IApplicationContext` through use of the `ContextHandler` discussed previously. The simple static method `GetContext()` can then be used to retrieve the context. Alternatively, if you create an `IApplicationContext` through other means you can register it with the `ContextRegistry` via the method `void RegisterContext(IApplicationContext context)` in the start-up code of your application. Hierarchical context retrieval is also supported through the use of the `GetContext(string name)` method, for example:

```
IApplicationContext ctx = ContextRegistry.GetContext("mySubContext");
```

This would retrieve the nested context for the context configuration shown previously.

```
<spring>
  <context>
    <resource uri="assembly://MyAssembly/MyProject/root-objects.xml"/>
    <context name="mySubContext">
      <resource uri="file://objects.xml"/>
    </context>
  </context>
</spring>
```

Do not call `ContextRegistry.GetContext` within a constructor as it will result in an endless recursion. (This is scheduled to be fixed in 1.1.1) In this case it is quite likely you can use the `IApplicationContextAware` interface and then retrieve other objects in a service locator style inside an initialization method.

The `ContextRegistry.Clear()` method will remove all contexts. On .NET 2.0, this will also call the `ConfigurationManager's RefreshSection` method so that the Spring context configuration section will be reread from disk when it is retrieved again. Note that in a web application `RefreshSection` will not work as advertised and you will need to touch the web.config files to reload a configuration.

5.16. Stereotype attributes

Beginning with Spring 1.2, the `[Repository]` attribute was introduced as a marker for any class that fulfills the role or stereotype of a repository (a.k.a. Data Access Object or DAO). Among the possibilities for leveraging such a marker is the automatic translation of exceptions as described in [Exception Translation](#).

Spring 1.2 introduces further stereotype annotations: `[Component]` and `[Service]`. `[Component]` serves as a generic stereotype for any Spring-managed component; whereas, `[Repository]` and `[Service]` serve as specializations of `[Component]` for more specific use cases (e.g., in the persistence and service layers, respectively). The ASP.NET MVC `[Controller]` attribute will serve this purpose for the controller layer. What this means is that you can annotate your component classes with `[Component]`, but by annotating them with `[Repository]` or `[Service]` your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. Of course, it is also possible that `[Repository]` and `[Service]` may carry additional semantics in future releases of the Spring Framework. Thus, if you are making a decision between using `[Component]` or `[Service]` for your service layer, `[Service]` is clearly the better choice. Similarly, as stated above, `[Repository]` is already supported as a marker for automatic exception translation in your persistence layer.

The next version of Spring will use the `[Component]` attribute to perform attribute based autowiring by-type as in the Spring Java Framework.

Chapter 6. The `IObjecWrapper` and Type conversion

6.1. Introduction

The concepts encapsulated by the `IObjecWrapper` interface are fundamental to the workings of the core Spring.NET libraries. The typical application developer most probably will not ever have the need to use the `IObjecWrapper` directly... because this is reference documentation however, we felt that some explanation of this core interface might be right. The `IObjecWrapper` is explained in this chapter since if you were going to use it at all, you would probably do that when trying to bind data to objects, which, nicely enough, is precisely the area that the `IObjecWrapper` addresses.

6.2. Manipulating objects using the `IObjecWrapper`

One quite important concept of the `Spring.Objects` namespace is encapsulated in the definition `IObjecWrapper` interface and its corresponding implementation, the `ObjecWrapper` class. The functionality offered by the `IObjecWrapper` includes methods to set and get property values (either individually or in bulk), get property descriptors (instances of the `System.Reflection.PropertyInfo` class), and to query the readability and writability of properties. The `IObjecWrapper` also offers support for nested properties, enabling the setting of properties on subproperties to an unlimited depth. The `IObjecWrapper` usually isn't used by application code directly, but by framework classes such as the various `IObjecFactory` implementations.

The way the `IObjecWrapper` works is partly indicated by its name: *it wraps an object* to perform actions on a wrapped object instance... such actions would include the setting and getting of properties exposed on the wrapped object.

Note: the concepts explained in this section are not important to you if you're not planning to work with the `IObjecWrapper` directly.

6.2.1. Setting and getting basic and nested properties

Setting and getting properties is done using the `SetPropertyValue()` and `GetPropertyValue()` methods, for which there are a couple of overloaded variants. The details of the various overloads (including return values and method parameters) are all described in the extensive API documentation supplied as a part of the Spring.NET distribution.

The aforementioned `SetPropertyValue()` and `GetPropertyValue()` methods have a number of conventions for indicating the path of a property. A property path is an expression that implementations of the `IObjecWrapper` interface can use to look up the properties of the wrapped object; some examples of property paths include...

Table 6.1. Examples of property paths

Path	Explanation
name	Indicates the <code>name</code> property of the wrapped object.
account.name	Indicates the nested property <code>name</code> of the <code>account</code> property of the wrapped object.

Path	Explanation
account[2]	Indicates the <i>third</i> element of the <code>account</code> property of the wrapped object. Indexed properties are typically collections such as <code>lists</code> and <code>dictionaries</code> , but can be any class that exposes an indexer.

Below you'll find some examples of working with the `IObjectWrapper` to get and set properties. Consider the following two classes:

```
[C#]
public class Company
{
    private string name;
    private Employee managingDirector;

    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

    public Employee ManagingDirector
    {
        get { return this.managingDirector; }
        set { this.managingDirector = value; }
    }
}
```

```
[C#]
public class Employee
{
    private string name;
    private float salary;

    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

    public float Salary
    {
        get { return salary; }
        set { this.salary = value; }
    }
}
```

The following code snippets show some examples of how to retrieve and manipulate some of the properties of `IObjectWrapper`-wrapped `Company` and `Employee` instances.

```
[C#]
Company c = new Company();
IObjectWrapper owComp = new ObjectWrapper(c);
// setting the company name...
owComp.SetPropertyValue("name", "Salina Inc.");
// can also be done like this...
PropertyValue v = new PropertyValue("name", "Salina Inc.");
owComp.SetPropertyValue(v);

// ok, let's create the director and bind it to the company...
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("name", "Don Fabrizio");
owComp.SetPropertyValue("managingDirector", don);

// retrieving the salary of the ManagingDirector through the company
float salary = (float)owComp.GetPropertyValue("managingDirector.salary");
```

Note that since the various Spring.NET libraries are compliant with the Common Language Specification (CLS), the resolution of arbitrary strings to properties, events, classes and such is performed in a case-insensitive fashion. The previous examples were all written in the C# language, which is a case-sensitive language, and yet the `Name` property of the `Employee` class was set using the all-lowercase `'name'` string identifier. The following example (using the classes defined previously) should serve to illustrate this...

```
[C#]
// ok, let's create the director and bind it to the company...
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("naMe", "Don Fabrizio");
owDon.GetPropertyValue("nAmE"); // gets "Don Fabrizio"

IObjectWrapper owComp = new ObjectWrapper(new Company());
owComp.SetPropertyValue("MaNaGiNgdirecToR", don);
owComp.SetPropertyValue("maNaGiNgdirector.salARy", 80000);
Console.WriteLine(don.Salary); // puts 80000
```

The case-insensitivity of the various Spring.NET libraries (dictated by the CLS) is not usually an issue... if you happen to have a class that has a number of properties, events, or methods that differ only by their case, then you might want to consider refactoring your code, since this is generally regarded as poor programming practice.

6.2.2. Other features worth mentioning

In addition to the features described in the preceding sections there a number of features that might be interesting to you, though not worth an entire section.

- *determining readability and writability*: using the `IsReadable()` and `IsWritable()` methods, you can determine whether or not a property is readable or writable.
- *retrieving `PropertyInfo` instances*: using `GetPropertyInfo(string)` and `GetPropertyInfos()` you can retrieve instances of the `System.Reflection.PropertyInfo` class, that might come in handy sometimes when you need access to the property metadata specific to the object being wrapped.

6.3. Type conversion

If you associate a `TypeConverter` with the definition of a custom `Type` using the standard .NET mechanism (see the example code below), Spring.NET will use the associated `TypeConverter` to do the conversion.

```
[C#]
[TypeConverter (typeof (FooTypeConverter))]
public class Foo
{
}
```

The `TypeConverter` class from the `System.ComponentModel` namespace of the .NET BCL is used extensively by the various classes in the `Spring.Core` library, as said class “... provides a unified way of converting types of values to other types, as well as for accessing standard values and subproperties.”¹

For example, a date can be represented in a human readable format (such as 30th August 1984), while we're still able to convert the human readable form to the original date format or (even better) to an instance of the `System.DateTime` class. This behavior can be achieved by using the standard .NET idiom of decorating a class with the `TypeConverterAttribute`. Spring.NET also offers another means of associating a `TypeConverters` with a class. You might want to do this to achieve a conversion that is not possible using standard idiom... for example,

¹More information about creating custom `TypeConverter` implementations can be found online at Microsoft's MSDN website, by searching for *Implementing a Type Converter*.

the `Spring.Core` library contains a custom `TypeConverter` that converts comma-delimited strings to `String` array instances. Registering custom converters on an `IObjectWrapper` instance gives the wrapper the knowledge of how to convert properties to the desired `Type`.

An example of where property conversion is used in `Spring.NET` is the setting of properties on objects, accomplished using the aforementioned `TypeConverters`. When mentioning `System.String` as the value of a property of some object (declared in an XML file for instance), `Spring.NET` will (if the type of the associated property is `System.Type`) use the `RuntimeTypeConverter` class to try to resolve the property value to a `Type` object. The example below demonstrates this automatic conversion of the `Example.Xml.SAXParser` (a string) into the corresponding `Type` instance for use in this factory-style class.

```
<objects xmlns="http://www.springframework.net">
  <object id="parserFactory" type="Example.XmlParserFactory, ExamplesLibrary"
    destroy-method="Close">
    <property name="ParserClass" value="Example.Xml.SAXParser, ExamplesLibrary"/>
  </object>
</objects>
```

```
[C#]
public class XmlParserFactory
{
    private Type parserClass;

    public Type ParserClass
    {
        get { return this.parserClass; }
        set { this.parserClass = value; }
    }

    public XmlParser GetParser ()
    {
        return Activator.CreateInstance (ParserClass);
    }
}
```

6.3.1. Type Conversion for Enumerations

The default type converter for enumerations is the `System.ComponentModel.EnumConverter` class. To specify the value for an enumerated property, simply use the name of the property. For example the `TestObject` class has a property of the enumerated type `FileMode`. One of the values for this enumeration is named `Create`. The following XML fragment shows how to configure this property

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="name" value="Rod"/>
  <property name="FileMode" value="Create"/>
</object>
```

6.4. Built-in TypeConverters

`Spring.NET` has a number of built-in `TypeConverters` to make life easy. Each of those is listed below and they are all located in the `Spring.Objects.TypeConverters` namespace of the `Spring.Core` library.

Table 6.2. Built-in TypeConverters

Type	Explanation
<code>RuntimeTypeConverter</code>	Parses strings representing <code>System.Types</code> to actual <code>System.Types</code> and the other way around.
<code>FileInfoConverter</code>	Capable of resolving strings to a <code>System.IO.FileInfo</code> object.

Type	Explanation
<code>StringArrayConverter</code>	Capable of resolving a comma-delimited list of strings to a string-array and vice versa.
<code>UriConverter</code>	Capable of resolving a string representation of a URI to an actual <code>Uri</code> -object.
<code>FileInfoConverter</code>	Capable of resolving a string representation of a <code>FileInfo</code> to an actual <code>FileInfo</code> -object.
<code>StreamConverter</code>	Capable of resolving Spring <code>IResource</code> URI (string) to its corresponding <code>InputStream</code> -object.
<code>ResourceConverter</code>	Capable of resolving Spring <code>IResource</code> URI (string) to an <code>IResource</code> object.
<code>ResourceManagerConverter</code>	Capable of resolving a two part string (resource name, assembly name) to a <code>System.Resources.ResourceManager</code> object.
<code>RgbColorConverter</code>	Capable of resolving a comma separated list of Red, Green, Blue integer values to a <code>System.Drawing.Color</code> structure.
<code>RegexConverter</code>	Converts string representation of regular expression into an instance of <code>System.Text.RegularExpressions.Regex</code>

Spring.NET uses the standard .NET mechanisms for the resolution of `System.Types`, including, but not limited to checking any configuration files associated with your application, checking the Global Assembly Cache (GAC), and assembly probing.

6.4.1. Custom type converters

You can register a custom type converter either Programmatically using the class `TypeConverterRegistry` or through configuration of Spring's container and described in the section [Registering Type Converters](#).

Chapter 7. Resources

7.1. Introduction

The `IResource` interface contained in the `Spring.Core.IO` namespace provides a common interface to describe and access data from diverse resource locations. This abstraction lets you treat the `InputStream` from a file and from a URL in a polymorphic and protocol-independent manner... the .NET BCL does not provide such an abstraction. The `IResource` interface inherits from `IInputStream` that provides a single property `Stream` `InputStream`. The `IResource` interface adds descriptive information about the resource via a number of additional properties. Several implementations for common resource locations, i.e. file, assembly, uri, are provided and you may also register custom `IResource` implementations.

7.2. The `IResource` interface

The `IResource` interface is shown below

```
public interface IResource : IInputStreamSource
{
    bool IsOpen { get; }

    Uri Uri { get; }

    FileInfo File { get; }

    string Description { get; }

    bool Exists { get; }

    IResource CreateRelative(string relativePath);
}
```

Table 7.1. `IResource` Properties

Property	Explanation
<code>InputStream</code>	Inherited from <code>IInputStream</code> . Opens and returns a <code>System.IO.Stream</code> . It is expected that each invocation returns a fresh <code>Stream</code> . It is the responsibility of the caller to close the stream.
<code>Exists</code>	returns a boolean indicating whether this resource actually exists in physical form.
<code>IsOpen</code>	returns a boolean indicating whether this resource represents a handle with an open stream. If true, the <code>InputStream</code> cannot be read multiple times, and must be read once only and then closed to avoid resource leaks. Will be false for all usual resource implementations, with the exception of <code>InputStreamResource</code> .
<code>Description</code>	Returns a description of the resource, such as the fully qualified file name or the actual URL.
<code>Uri</code>	The Uri representation of the resource.
<code>File</code>	Returns a <code>System.IO.FileInfo</code> for this resource if it can be resolved to an absolute file path.

and the methods

Table 7.2. IResource Methods

Method	Explanation
<code>IResource CreateRelative</code> (string relativePath)	Creates a resource relative to this resource using relative path like notation (<code>./</code> and <code>../</code>).

You can obtain an actual URL or File object representing the resource if the underlying implementation is compatible and supports that functionality.

The Resource abstraction is used extensively in Spring itself, as an argument type in many method signatures when a resource is needed. Other methods in some Spring APIs (such as the constructors to various `IApplicationContext` implementations), take a String which is used to create a Resource appropriate to that context implementation

While the Resource interface is used a lot with Spring and by Spring, it's actually very useful to use as a general utility class by itself in your own code, for access to resources, even when your code doesn't know or care about any other parts of Spring. While this couples your code to Spring, it really only couples it to this small set of utility classes and can be considered equivalent to any other library you would use for this purpose

7.3. Built-in IResource implementations

The resource implementations provided are

- `AssemblyResource` accesses data stored as .NET resources inside an assembly. Uri syntax is `assembly://<AssemblyName>/<NameSpace>/<ResourceName>`
- `ConfigSectionResource` accesses Spring.NET configuration data stored in a custom configuration section in the .NET application configuration file (i.e. App.config). Uri syntax is `config://<path to section>`
- `FileSystemResource` accesses file system data. Uri syntax is `file://<filename>`
- `InputStreamResource` a wrapper around a raw `System.IO.Stream`. Uri syntax is not supported.
- `UriResource` accesses data from the standard System.Uri protocols such as http and https. In .NET 2.0 you can use this also for the ftp protocol. Standard Uri syntax is supported.

Refer to the MSDN documentation for more information on [supported Uri scheme types](#).

7.3.1. Registering custom IResource implementations

The configuration section handler, `ResourceHandlersSectionHandler`, is used to register any custom `IResource` implementations you have created. In the configuration section you list the type of `IResource` implementation and the protocol prefix. Your custom `IResource` implementation must provide a constructor that takes a string as it's sole argument that represents the URI string. Refer to the SDK documentation for `ResourceHandlersSectionHandler` for more information. An example of the `ResourceHandlersSectionHandler` is shown below for a fictional `IResource` implementation that interfaces with a database.

```
<configuration>
  <configSections>
    <sectionGroup name="spring">

      <section name='context' type='Spring.Context.Support.ContextHandler, Spring.Core' />

      <section name="resourceHandlers"
        type="Spring.Context.Support.ResourceHandlersSectionHandler, Spring.Core" />

    </sectionGroup>
```

```

</configSections>

<spring>

  <resourceHandlers>
    <handler protocol="db" type="MyCompany.MyApp.Resources.MyDbResource, MyAssembly"/>
  </resourceHandlers>

  <context>
    <resource uri="db://user:pass@dbName/MyDefinitionsTable"/>
  </context>

</spring>
</configuration>

```

7.4. The `IResourceLoader`

To load resources given their Uri syntax, an implementation of the `IResourceLoader` is used. The default implementation is `ConfigurableResourceLoader`. Typically you will not need to access this class directly since the `IApplicationContext` implements the `IResourceLoader` interface that contains the single method `IResource getResource(string location)`. The provided implementations of `IApplicationContext` delegate this method to an instance of `ConfigurableResourceLoader` which supports the Uri protocols/schemes listed previously. If you do not specify a protocol then the file protocol is used. The following shows some sample usage.

```

IResource resource = appContext.getResource("http://www.springframework.net/license.html");
resource = appContext.getResource("assembly://Spring.Core.Tests/Spring/TestResource.txt");
resource = appContext.getResource("https://sourceforge.net/");
resource = appContext.getResource("file:///C:/WINDOWS/ODBC.INI");

StreamReader reader = new StreamReader(resource.InputStream);
Console.WriteLine(reader.ReadToEnd());

```

Other protocols can be registered along with a new implementations of an `IResource` that must correctly parse a Uri string in its constructor. An example of this can be seen in the `Spring.Web` namespace that uses `Server.MapPath` to resolve the filename of a resource.

The `CreateRelative` method allows you to easily load resources based on a relative path name. In the case of relative assembly resources, the relative path navigates the namespace within an assembly. For example:

```

IResource res = new AssemblyResource("assembly://Spring.Core.Tests/Spring/TestResource.txt");
IResource res2 = res.CreateRelative("../IO/TestIOResource.txt");

```

This loads the resource `TestResource.txt` and then navigates to the `Spring.Core.IO` namespace and loads the resource `TestIOResource.txt`

7.5. The `IResourceLoaderAware` interface

The `IResourceLoaderAware` interface is a special marker interface, identifying objects that expect to be provided with a `IResourceLoader` reference.

```

public interface IResourceLoaderAware
{
    IResourceLoader ResourceLoader
    {
        set;
        get;
    }
}

```

When a class implements `IResourceLoaderAware` and is deployed into an application context (as a Spring-managed object), it is recognized as `IResourceLoaderAware` by the application context. The application context

will then invoke the `ResourceLoader` property, supplying itself as the argument (remember, all application contexts in Spring implement the `IResourceLoader` interface).

Of course, since an `IApplicationContext` is a `IResourceLoader`, the object could also implement the `IApplicationContextAware` interface and use the supplied application context directly to load resources, but in general, it's better to use the specialized `IResourceLoader` interface if that's all that's needed. The code would just be coupled to the resource loading interface, which can be considered a utility interface, and not the whole Spring `IApplicationContext` interface.

7.6. Application contexts and `IResource` paths

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location path(s) of the resource(s) such as XML files that make up the definition of the context. For example, you can create an `XmlApplicationContext` from two resources as follows:

```
IApplicationContext context = new XmlApplicationContext(  
    "file://objects.xml", "assembly://MyAssembly/MyProject/objects-dal-layer.xml");
```

Chapter 8. Threading and Concurrency Support

8.1. Introduction

The purpose of the `Spring.Threading` namespace is to provide a place to keep useful concurrency abstractions that augment those in the BCL. Since Doug Lea has provided a wealth of mature public domain concurrency abstractions in his Java based 'EDU.oswego.cs.dl.util.concurrent' libraries we decided to port a few of his abstractions to .NET. So far, we've only ported three classes, the minimum necessary to provide basic object pooling functionality to support an AOP based pooling aspect and to provide a Semaphore class that was mistakenly not included in .NET 1.0/1.1.

There is also an important abstraction, `IThreadStorage`, for performing thread local storage.

8.2. Thread Local Storage

Depending on your runtime environment there are different strategies to use for storing objects in thread local storage. If you are in web applications a single Request may be executed on different threads. As such, the location to store thread local objects is in `HttpContext.Current`. For other environments `System.Runtime.Remoting.Messaging.CallContext` is used. For more background information on the motivation behind these choices, say as compared to the attribute `[ThreadStatic]` refer to "Piers7"'s [blog](#) and this [forum post](#). The interface `IThreadStorage` serves as the basis for the thread local storage abstraction and various implementations can be selected from depending on your runtime requirements. Configuring the implementation of `IThreadStorage` makes it easier to have more portability across runtime environments.

The API is quite simple and shown below

```
public interface IThreadStorage
{
    object GetData(string name)

    void SetData(string name, object value)

    void FreeNamedDataSlot(string name)
}
```

The methods `GetData` and `SetData` are responsible for retrieving and setting the object that is to be bound to thread local storage and associating it with a name. Clearing the thread local storage is done via the method `FreeNamedDataSlot`.

In `Spring.Core` is the implementation, `CallContextStorage`, that directly uses `CallContext` and also the implementation `LogicalThreadContext` which by default uses `CallContextStorage` but can be configured via the static method `SetStorage(IThreadStorage)`. The methods on `CallContextStorage` and `LogicalThreadContext` are static.

In `Spring.Web` is the implementation `HttpContextStorage` which uses the `HttpContext` to store thread local data and `HybridContextStorage` that uses `HttpContext` if within a web environment, i.e. `HttpContext.Current != null`, and `CallContext` otherwise.

Spring internally uses `LogicalThreadContext` as this doesn't require a coupling to the `System.Web` namespace. In the case of Spring based web applications, Spring's `WebSupportModule` sets the storage strategy of `LogicalThreadContext` to be `HybridContextStorage`.

8.3. Synchronization Primitives

When you take a look at these synchronization classes, you'll wonder why it's even necessary when `System.Threading` provides plenty of synchronization options. Although `System.Threading` provides great synchronization classes, it doesn't provide well-factored abstractions and interfaces for us. Without these abstractions, we will tend to code at a low-level. With enough experience, you'll eventually come up with some abstractions that work well. Doug Lea has already done a lot of that research and has a class library that we can take advantage of.

8.3.1. ISync

`ISync` is the central interface for all classes that control access to resources from multiple threads. It's a simple interface which has two basic use cases. The first case is to block indefinitely until a condition is met:

```
void ConcurrentRun(ISync lock) {
    lock.Acquire(); // block until condition met
    try {
        // ... access shared resources
    }
    finally {
        lock.Release();
    }
}
```

The other case is to specify a maximum amount of time to block before the condition is met:

```
void ImpatientConcurrentRun(ISync lock) {
    // block for at most 10 milliseconds for condition
    if ( lock.Attempt(10) ) {
        try {
            // ... access shared resources
        }
        finally {
            lock.Release();
        }
    } else {
        // complain of time out
    }
}
```

8.3.2. SyncHolder

The `SyncHolder` class implements the `System.IDisposable` interface and so provides a way to use an `ISync` with the using C# keyword: the `ISync` will be automatically Acquired and then Released on exiting from the block.

This should simplify the programming model for code using (!) an `ISync`:

```
ISync sync = ...
...
using (new SyncHolder(sync))
{
    // ... code to be executed
    // holding the ISync lock
}
```

There is also the timed version, a little more cumbersome as you must deal with timeouts:

```
ISync sync = ...
long msec = 100;
...
// try to acquire the ISync for msec milliseconds
try
{
    using (new SyncHolder(sync, msec))
    {
        // ... code to be executed
        // holding the ISync lock
    }
}
catch (TimeoutException)
{
    // deal with failed lock acquisition
}
```

8.3.3. Latch

The `Latch` class implements the `ISync` interface and provides an implementation of a *latch*. A latch is a boolean condition that is set at most once, ever. Once a single release is issued, all acquires will pass. It is similar to a `ManualResetEvent` initialized unsignalled (Reset) and can only be `Set()`. A typical use is to act as a start signal for a group of worker threads.

```
class Boss {
    Latch _startPermit;

    void Worker() {
        // very slow worker initialization ...
        // ... attach to messaging system
        // ... connect to database
        _startPermit.Acquire();
        // ... use resources initialized in Mush
        // ... do real work
    }

    void Mush() {
        _startPermit = new Latch();
        for (int i=0; i<10; ++i) {
            new Thread(new ThreadStart(Worker)).Start();
        }
        // very slow main initialization ...
        // ... parse configuration
        // ... initialize other resources used by workers
        _startPermit.Release();
    }
}
```

8.3.4. Semaphore

The `Semaphore` class implements the `ISync` interface and provides an implementation of a semaphore. Conceptually, a semaphore maintains a set of permits. Each `Acquire()` blocks if necessary until a permit is available, and then takes it. Each `Release()` adds a permit. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly. A typical use is to control access to a pool of shared objects.

```
class LimitedConcurrentUploader {
    // ensure we don't exceed maxUpload simultaneous uploads
    Semaphore _available;
    public LimitedConcurrentUploader(maxUploads) {
        _available = new Semaphore(maxUploads);
    }
    // no matter how many threads call this method no more
```

```
// than maxUploads concurrent uploads will occur.
public Upload(IDataTransfer upload) {
    _available.Acquire();
    try {
        upload.TransferData();
    }
    finally {
        _available.Release();
    }
}
```

Chapter 9. Object Pooling

9.1. Introduction

The `Spring.Pool` namespace contains a generic API for implementing pools of objects. Object pooling is a well known technique to minimize the creation of objects that can take a significant amount of time. Common examples are to create a pool of database connections such that each request to the database can reuse an existing connection instead of creating one per client request. Threads are also another common candidate for pooling in order to increase responsiveness of an application to multiple concurrent client requests.

.NET contains support for object pooling in these common scenarios. Support for database connection pools is directly supported by ADO.NET data providers as a configuration option. Similarly, thread pooling is supported via the `System.ThreadPool` class. Support for pooling of other objects can be done using the CLR managed API to COM+ found in the `System.EnterpriseServices` namespace.

Despite this built-in support there are scenarios where you would like to use alternative pool implementations. This may be because the default implementations, such as `System.ThreadPool`, do not meet your requirements. (For a discussion on advanced `ThreadPool` usage see [Smart Thread Pool](#) by Ami Bar.) Alternatively, you may want to pool classes that do not inherit from `System.EnterpriseServices.ServicedComponent`. Instead of making changes to the object model to meet this inheritance requirement, Spring .NET provides similar support for pooling, but for any object, by using AOP proxies and a generic pool API for managing object instances.

Note, that if you are concerned only with applying pooling to an existing object, the pooling APIs discussed here are not very important. Instead the use and configuration of `Spring.Aop.Target.SimplePoolTargetSource` is more relevant. Pooling of objects can either be done Programatically or through the XML configuration of the Spring .NET container. Attribute support for pooling, similar to the `ServicedComponent` approach, will be available in a future release of Spring.NET.

Chapter 35, *IoC Quickstarts* contains an example that shows the use of the pooling API independent of AOP functionality.

9.2. Interfaces and Implementations

The `Spring.Pool` namespace provides two simple interfaces to manage pools of objects. The first interface, `IObjectPool` describes how to take and put back an object from the pool. The second interface `IPoolableObjectFactory` is meant to be used in conjunction with implementations of the `IObjectPool` to provide guidance in calling various lifecycle events on the objects managed by the pool. These interfaces are based on the Jakarta Commons Pool API. `Spring.Pool.Support.SimplePool` is a default implementation of `IObjectPool` and `Spring.Aop.Target.SimplePoolTargetSource` is the implementation of `IPoolableObjectFactory` for use with AOP. The current goal of the `Spring.Pool` namespace is not to provide a one-for-one replacement of the Jakarta Commons Pool API, but rather to support basic object pooling needs for common AOP scenarios. Consequently, other interfaces and base classes available in the Jakarta package are not available.

Chapter 10. Spring.NET miscellanea

10.1. Introduction

This chapter contains miscellanea information on features, goodies, caveats that does not belong to any particular area.

10.2. PathMatcher

`Spring.Util.PathMatcher` provides Ant/NAnt-like path name matching features.

To do the match, you use the method:

```
static bool Match(string pattern, string path)
```

If you want to decide if case is important or not use the method:

```
static bool Match(string pattern, string path, bool ignoreCase)
```

10.2.1. General rules

To build your pattern, you use the `*`, `?` and `**` building blocks:

- `*`: matches any number of non slash characters;
- `?`: matches exactly 1 (one) non slash/dot character;
- `**`: matches any subdirectory, without taking care of the depth;

10.2.2. Matching filenames

A file name can be matched using the following notation:

```
foo?bar.*
```

matches:

```
fooAbar.txt  
foolbar.txt  
foo_bar.txt  
foo-bar.txt
```

does not match:

```
foo.bar.txt  
foo/bar.txt  
foo\bar.txt
```

The classical all files pattern:

```
*,*
```

matches:

```
foo.db  
.db  
foo  
foo.bar.db  
foo.db.db  
db.db.db
```

does not match:

```
c:/
c:/foo.db
c:/foo
c:/db
c:/foo.foo.db
//server/foo
```

10.2.3. Matching subdirectories

A directory name can be matched at any depth level using the following notation:

```
**/db/**
```

That pattern matches the following paths:

```
/db
//server/db
c:/db
c:/spring/app/db/foo.db
//Program Files/App/spaced dir/db/foo.db
/home/spring/spaced dir/db/v1/foo.db
```

but does not match these:

```
c:/spring/app/db-v1/foo.db
/home/spring/spaced dir/db-v1/foo.db
```

You can compose subdirectories to match like this:

```
**/bin/**/tmp/**
```

That pattern matches the following paths:

```
c:/spring/foo/bin/bar/tmp/a
c:/spring/foo/bin/tmp/a/b.c
```

but does not match these:

```
c:/spring/foo/bin/bar/temp/a
c:/tmp/foo/bin/bar/a/b.c
```

You can use more advanced patterns:

```
**/.spring-assemblies**
```

matches:

```
c/.spring-assemblies
c/.spring-assembliesabcd73xs
c/app/.spring-assembliesabcd73xs
c/app/.spring-assembliesabcd73xs/foo.dll
//server/app/.spring-assembliesabcd73xs
```

does not match:

```
c:/app/.spring-assemblie
```

10.2.4. Case does matter, slashes don't

.NET is expected to be a cross-platform development ... platform. So, `PathMatcher` will match taking care of the case of the pattern and the case of the path. For example:

```
**/db/**/*.*.DB
```

matches:

```
c:/spring/service/deploy/app/db/foo.DB
```

but does not match:

```
c:/spring/service/deploy/app/DB/foo.DB  
c:/spring/service/deploy/app/spaced dir/DB/foo.DB  
//server/share/service/deploy/app/DB/backup/foo.db
```

If you do not matter about case, you should explicitly tell the `PathMatcher`.

Back and forward slashes, in the very same cross-platform spirit, are not important:

```
spring/foo.bar
```

matches all the following paths:

```
c:\spring\foo.bar  
c:/spring\foo.bar  
c:/spring/foo.bar  
/spring/foo.bar  
\spring\foo.bar
```

Chapter 11. Expression Evaluation

11.1. Introduction

The Spring.Expressions namespace provides a powerful expression language for querying and manipulating an object graph at runtime. The language supports setting and getting of property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection, as well as common list aggregators.

The functionality provided in this namespace serves as the foundation for a variety of other features in Spring.NET such as enhanced property evaluation in the XML based configuration of the IoC container, a Data Validation framework, and a Data Binding framework for ASP.NET. You will likely find other cool uses for this library in your own work where run-time evaluation of criteria based on an object's state is required. For those with a Java background, the Spring.Expressions namespace provides functionality similar to the Java based Object Graph Navigation Language, [OGNL](#).

This chapter covers the features of the expression language using an `Inventor` and `Inventor's Society` class as the target objects for expression evaluation. The class declarations and the data used to populate them are listed at the end of the chapter in section Section 11.4, "Classes used in the examples". These classes are blatantly taken from the NUnit tests for the Expressions namespace which you can refer to for additional example usage.

11.2. Evaluating Expressions

The simplest, but not the most efficient way to perform expression evaluation is by using one of the static convenience methods of the `ExpressionEvaluator` class:

```
public static object GetValue(object root, string expression);

public static object GetValue(object root, string expression, IDictionary variables)

public static void SetValue(object root, string expression, object newValue)

public static void SetValue(object root, string expression, IDictionary variables, object newValue)
```

The first argument is the 'root' object that the expression string (2nd argument) will be evaluated against. The third argument is used to support variables in the expression and will be discussed later. Simple usage to get the value of an object property is shown below using the `Inventor` class. You can find the class listing in section Section 11.4, "Classes used in the examples".

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1856, 7, 9), "Serbian");

tesla.PlaceOfBirth.City = "Smiljan";

string evaluatedName = (string) ExpressionEvaluator.GetValue(tesla, "Name");

string evaluatedCity = (string) ExpressionEvaluator.GetValue(tesla, "PlaceOfBirth.City");
```

The value of 'evaluatedName' is 'Nikola Tesla' and that of 'evaluatedCity' is 'Smiljan'. A period is used to navigate the nested properties of the object. Similarly to set the property of an object, say we want to rewrite history and change Tesla's city of birth, we would simply add the following line

```
ExpressionEvaluator.SetValue(tesla, "PlaceOfBirth.City", "Novi Sad");
```

A much better way to evaluate expressions is to parse them once and then evaluate as many times as you want using `ExpressionClass`. Unlike `ExpressionEvaluator`, which parses expression every time you invoke one of its methods, `Expression` class will cache the parsed expression for increased performance. The methods of this class are listed below:

```
public static IExpression Parse(string expression)

public override object Get(object context, IDictionary variables)

public override void Set(object context, IDictionary variables, object newValue)
```

The retrieval of the `Name` property in the previous example using the `Expression` class is shown below

```
IExpression exp = Expression.Parse("Name");

string evaluatedName = (string) exp.GetValue(tesla, null);
```

The difference in performance between the two approaches, when evaluating the same expression many times, is several orders of magnitude, so you should only use convenience methods of the `ExpressionEvaluator` class when you are doing one-off expression evaluations. In all other cases you should parse the expression first and then evaluate it as many times as you need.

There are a few exception classes to be aware of when using the `ExpressionEvaluator`. These are `InvalidPropertyException`, when you refer to a property that doesn't exist, `NullValueInNestedPathException`, when a null value is encountered when traversing through the nested property list, and `ArgumentException` and `NotSupportedException` when you pass in values that are in error in some other manner.

The expression language is based on a grammar and uses [ANTLR](#) to construct the lexer and parser. Errors relating to bad syntax of the language will be caught at this level of the language implementation. For those interested in the digging deeper into the implementation, the grammar file is named `Expression.g` and is located in the `src` directory of the namespace. As a side note, the release version of the ANTLR DLL included with Spring.NET was signed with the Spring.NET key, which means that you should always use the included version of `antlr.runtime.dll` within your application. Upcoming releases of ANTLR will provide strongly signed assemblies, which will remove this requirement.

11.3. Language Reference

11.3.1. Literal expressions

The types of literal expressions supported are strings, dates, numeric values (int, real, and hex), boolean and null. String are delimited by single quotes. To put a single quote itself in a string use the backslash character. The following listing shows simple usage of literals. Typically they would not be used in isolation like this, but as part of a more complex expression, for example using a literal on one side of a logical comparison operator.

```
string helloWorld = (string) ExpressionEvaluator.GetValue(null, "Hello World"); // evals to "Hello World"

string tonyPizza = (string) ExpressionEvaluator.GetValue(null, "Tony\\'s Pizza"); // evals to "Tony's Pizza"

double avogadrosNumber = (double) ExpressionEvaluator.GetValue(null, "6.0221415E+23");

int maxValue = (int) ExpressionEvaluator.GetValue(null, "0xFFFFFFFF"); // evals to 2147483647

DateTime birthday = (DateTime) ExpressionEvaluator.GetValue(null, "date('1974/08/24')");

DateTime exactBirthday =
    (DateTime) ExpressionEvaluator.GetValue(null, " date('19740824T131030', 'yyyyMMddTHH:mm:ss')");

bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "true");
```

```
object nullValue = ExpressionEvaluator.GetValue(null, "null");
```

Note that the extra backslash character in Tony's Pizza is to satisfy C# escape syntax. Numbers support the use of the negative sign, exponential notation, and decimal points. By default real numbers are parsed using `Double.Parse` unless the format character "M" or "F" is supplied, in which case `Decimal.Parse` and `Single.Parse` would be used respectfully. As shown above, if two arguments are given to the date literal then `DateTime.ParseExact` will be used. Note that all parse methods of classes that are used internally reference the `CultureInfo.InvariantCulture`.

11.3.2. Properties, Arrays, Lists, Dictionaries, Indexers

As shown in the previous example in Section 11.2, “Evaluating Expressions”, navigating through properties is easy, just use a period to indicate a nested property value. The instances of `Inventor` class, *pupin* and *tesla*, were populated with data listed in section Section 11.4, “Classes used in the examples”. To navigate "down" and get Tesla's year of birth and Pupin's city of birth the following expressions are used

```
int year = (int) ExpressionEvaluator.GetValue(tesla, "DOB.Year"); // 1856

string city = (string) ExpressionEvaluator.GetValue(pupin, "PlaCeOfBirTh.CiTy"); // "Idvor"
```

For the sharp-eyed, that isn't a typo in the property name for place of birth. The expression uses mixed cases to demonstrate that the evaluation is case insensitive.

The contents of arrays and lists are obtained using square bracket notation.

```
// Inventions Array
string invention = (string) ExpressionEvaluator.GetValue(tesla, "Inventions[3]"); // "Induction motor"

// Members List
string name = (string) ExpressionEvaluator.GetValue(ieee, "Members[0].Name"); // "Nikola Tesla"

// List and Array navigation
string invention = (string) ExpressionEvaluator.GetValue(ieee, "Members[0].Inventions[6]") // "Wireless communication"
```

The contents of dictionaries are obtained by specifying the literal key value within the brackets. In this case, because keys for the *Officers* dictionary are strings, we can specify string literal.

```
// Officer's Dictionary
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers['president']");

string city = (string) ExpressionEvaluator.GetValue(ieee, "Officers['president'].PlaceOfBirth.City"); // "Idvor"

ExpressionEvaluator.SetValue(ieee, "Officers['advisors'][0].PlaceOfBirth.Country", "Croatia");
```

You may also specify non literal values in place of the quoted literal values by using another expression inside the square brackets such as variable names or static properties/methods on other types. These features are discussed in other sections.

Indexers are similarly referenced using square brackets. The following is a small example that shows the use of indexers. Multidimensional indexers are also supported.

```
public class Bar
{
    private int[] numbers = new int[] {1, 2, 3};

    public int this[int index]
    {
        get { return numbers[index]; }
        set { numbers[index] = value; }
    }
}
```

```
Bar b = new Bar();

int val = (int) ExpressionEvaluator.GetValue(bar, "[1]") // evaluated to 2

ExpressionEvaluator.SetValue(bar, "[1]", 3); // set value to 3
```

11.3.2.1. Defining Arrays, Lists and Dictionaries Inline

In addition to accessing arrays, lists and dictionaries by navigating the graph for the context object, Spring.NET Expression Language allows you to define them inline, within the expression. Inline lists are defined by simply enclosing a comma separated list of items with curly brackets:

```
{1, 2, 3, 4, 5}
{'abc', 'xyz'}
```

If you want to ensure that a strongly typed array is initialized instead of a weakly typed list, you can use array initializer instead:

```
new int[] {1, 2, 3, 4, 5}
new string[] {'abc', 'xyz'}
```

Dictionary definition syntax is a bit different: you need to use a # prefix to tell expression parser to expect key/value pairs within the brackets and to specify a comma separated list of key/value pairs within the brackets:

```
#{'key1' : 'Value 1', 'today' : DateTime.Today}
#{1 : 'January', 2 : 'February', 3 : 'March', ...}
```

Arrays, lists and dictionaries created this way can be used anywhere where arrays, lists and dictionaries obtained from the object graph can be used, which we will see later in the examples.

Keep in mind that even though examples above use literals as array/list elements and dictionary keys and values, that's only to simplify the examples -- you can use any valid expression wherever literals are used.

11.3.3. Methods

Methods are invoked using typical C# programming syntax. You may also invoke methods on literals.

```
//string literal
char[] chars = (char[]) ExpressionEvaluator.GetValue(null, "'test'.ToCharArray(1, 2)") // 't','e'

//date literal
int year = (int) ExpressionEvaluator.GetValue(null, "date('1974/08/24').AddYears(31).Year") // 2005

// object usage, calculate age of tesla navigating from the IEEE society.
ExpressionEvaluator.GetValue(ieee, "Members[0].GetAge(date('2005-01-01'))" // 149 (eww...a big anniversary is coming up ;)
```

11.3.4. Operators

11.3.4.1. Relational operators

The relational operators; equal, not equal, less than, less than or equal, greater than, and greater than or equal are supported using standard operator notation. These operators take into account if the object implements the `IComparable` interface. Enumerations are also supported but you will need to register the enumeration type, as described in Section Section 11.3.8, “Type Registration”, in order to use an enumeration value in an expression if it is not contained in the `mscorlib`.

```
ExpressionEvaluator.GetValue(null, "2 == 2") // true

ExpressionEvaluator.GetValue(null, "date('1974-08-24') != DateTime.Today") // true
```

```

ExpressionEvaluator.GetValue(null, "2 < -5.0") // false

ExpressionEvaluator.GetValue(null, "DateTime.Today <= date('1974-08-24')") // false

ExpressionEvaluator.GetValue(null, "'Test' >= 'test'") // true

```

Enumerations can be evaluated as shown below

```

FooColor fColor = new FooColor();

ExpressionEvaluator.SetValue(fColor, "Color", KnownColor.Blue);

bool trueValue = (bool) ExpressionEvaluator.GetValue(fColor, "Color == KnownColor.Blue"); //true

```

Where FooColor is the following class.

```

public class FooColor
{
    private KnownColor knownColor;

    public KnownColor Color
    {
        get { return knownColor; }
        set { knownColor = value; }
    }
}

```

In addition to standard relational operators, Spring.NET Expression Language supports some additional, very useful operators that were "borrowed" from SQL, such as *in*, *like* and *between*, as well as *is* and *matches* operators, which allow you to test if object is of a specific type or if the value matches a regular expression.

```

ExpressionEvaluator.GetValue(null, "3 in {1, 2, 3, 4, 5}") // true

ExpressionEvaluator.GetValue(null, "'Abc' like '[A-Z]b*'" // true

ExpressionEvaluator.GetValue(null, "'Abc' like '?'") // false

ExpressionEvaluator.GetValue(null, "1 between {1, 5}") // true

ExpressionEvaluator.GetValue(null, "'efg' between {'abc', 'xyz'}") // true

ExpressionEvaluator.GetValue(null, "'xyz' is int") // false

ExpressionEvaluator.GetValue(null, "{1, 2, 3, 4, 5} is IList") // true

ExpressionEvaluator.GetValue(null, "'5.0067' matches '^~?\d+(\.\d{2})?$',") // false

ExpressionEvaluator.GetValue(null, @"'5.00' matches '^~?\d+(\.\d{2})?$',") // true

```

Note that the Visual Basic and not SQL syntax is used for the *like* operator pattern string.

11.3.4.2. Logical operators

The logical operators that are supported are *and*, *or*, and *not*. Their use is demonstrated below

```

// AND
bool falseValue = (bool) ExpressionEvaluator.GetValue(null, "true and false"); //false

string expression = @"IsMember('Nikola Tesla') and IsMember('Mihajlo Pupin')";
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, expression); //true

// OR
bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "true or false"); //true

string expression = @"IsMember('Nikola Tesla') or IsMember('Albert Einstien')";
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, expression); // true

// NOT
bool falseValue = (bool) ExpressionEvaluator.GetValue(null, "!true");

```



```
// AND and NOT
string expression = @"IsMember('Nikola Tesla') and !IsMember('Mihajlo Pupin')";
bool falseValue = (bool) ExpressionEvaluator.GetValue(ieee, expression);
```

11.3.4.3. Bitwise operators

The bitwise operators that are supported are *and*, *or*, *xor* and *not*. Their use is demonstrated below. Note, that the logical and bitwise operators are the same and their interpretation depends if you pass in integral values or boolean values.

```
// AND
int result = (int) ExpressionEvaluator.GetValue(null, "1 and 3"); // 1 & 3

// OR
int result = (int) ExpressionEvaluator.GetValue(null, "1 or 3"); // 1 | 3

// XOR
int result = (int) ExpressionEvaluator.GetValue(null, "1 xor 3"); // 1 ^ 3

// NOT
int result = (int) ExpressionEvaluator.GetValue(null, "!1"); // ~1
```

11.3.4.4. Mathematical operators

The addition operator can be used on numbers, strings and dates. Subtraction can be used on numbers and dates. Multiplication and division can be used only on numbers. Other mathematical operators supported are modulus (%) and exponential power (^). Standard operator precedence is enforced. These operators are demonstrated below

```
// Addition
int two = (int)ExpressionEvaluator.GetValue(null, "1 + 1"); // 2

String testString = (String)ExpressionEvaluator.GetValue(null, "'test' + ' ' + 'string'"); //'test string'

DateTime dt = (DateTime)ExpressionEvaluator.GetValue(null, "date('1974-08-24') + 5"); // 8/29/1974

// Subtraction
int four = (int) ExpressionEvaluator.GetValue(null, "1 - -3"); //4

Decimal dec = (Decimal) ExpressionEvaluator.GetValue(null, "1000.00m - 1e4"); // 9000.00

TimeSpan ts = (TimeSpan) ExpressionEvaluator.GetValue(null, "date('2004-08-14') - date('1974-08-24')"); //10948.00:00:00

// Multiplication
int six = (int) ExpressionEvaluator.GetValue(null, "-2 * -3"); // 6

int twentyFour = (int) ExpressionEvaluator.GetValue(null, "2.0 * 3e0 * 4"); // 24

// Division
int minusTwo = (int) ExpressionEvaluator.GetValue(null, "6 / -3"); // -2

int one = (int) ExpressionEvaluator.GetValue(null, "8.0 / 4e0 / 2"); // 1

// Modulus
int three = (int) ExpressionEvaluator.GetValue(null, "7 % 4"); // 3

int one = (int) ExpressionEvaluator.GetValue(null, "8.0 % 5e0 % 2"); // 1

// Exponent
int sixteen = (int) ExpressionEvaluator.GetValue(null, "-2 ^ 4"); // 16

// Operator precedence
int minusFortyFive = (int) ExpressionEvaluator.GetValue(null, "1+2-3*8^2/2/2"); // -45
```

11.3.5. Assignment

Setting of a property is done by using the assignment operator. This would typically be done within a call to `GetValue` since in the simple case `SetValue` offers the same functionality. Assignment in this manner is useful when combining multiple operators in an expression list, discussed in the next section. Some examples of assignment are shown below

```
Inventor inventor = new Inventor();
String aleks = (String) ExpressionEvaluator.GetValue(inventor, "Name = 'Aleksandar Seovic'");
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(inventor, "DOB = date('1974-08-24')");

//Set the vice president of the society
Inventor tesla = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers['vp'] = Members[0]");
```

11.3.6. Expression lists

Multiple expressions can be evaluated against the same context object by separating them with a semicolon and enclosing the entire expression within parentheses. The value returned is the value of the last expression in the list. Examples of this are shown below

```
//Perform property assignments and then return Name property.

String pupin = (String) ExpressionEvaluator.GetValue(ieee.Members,
    "( [1].PlaceOfBirth.City = 'Beograd'; [1].PlaceOfBirth.Country = 'Serbia'; [1].Name )");

// pupin = "Mihajlo Pupin"
```

11.3.7. Types

In many cases, you can reference types by simply specifying type name:

```
ExpressionEvaluator.GetValue(null, "1 is int")

ExpressionEvaluator.GetValue(null, "DateTime.Today")

ExpressionEvaluator.GetValue(null, "new string[] { 'abc', 'efg' }")
```

This is possible for all standard types from `mscorlib`, as well as for any other type that is registered with the `TypeRegistry` as described in the next section.

For all other types, you need to use special `T(typeName)` expression:

```
Type dateType = (Type) ExpressionEvaluator.GetValue(null, "T(System.DateTime)")

Type evalType = (Type) ExpressionEvaluator.GetValue(null, "T(Spring.Expressions.ExpressionEvaluator, Spring.Core)")

bool trueValue = (bool) ExpressionEvaluator.GetValue(tesla, "T(System.DateTime) == DOB.GetType()")
```



Note

The implementation delegates to Spring's `ObjectUtils.ResolveType` method for the actual type resolution, which means that the types used within expressions are resolved in the exactly the same way as the types specified in Spring configuration files.

11.3.8. Type Registration

To refer to a type within an expression that is not in the `mscorlib` you need to register it with the `TypeRegistry`. This will allow you to refer to a shorthand name of the type within your expressions. This is commonly used in expression that use the new operator or refer to a static properties of an object. Example usage is shown below.

```
TypeRegistry.RegisterType("Society", typeof(Society));

Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers[Society.President]");
```

Alternatively, you can register types using `typeAliases` configuration section.

11.3.9. Constructors

Constructors can be invoked using the `new` operator. For classes outside `microsoft` you will need to register your types so they can be resolved. Examples of using constructors are shown below:

```
// simple ctor
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(null, "new DateTime(1974, 8, 24)");

// Register Inventor type then create new inventor instance within Add method inside an expression list.
// Then return the new count of the Members collection.

TypeRegistry.RegisterType(typeof(Inventor));
int three = (int) ExpressionEvaluator.GetValue(ieee.Members, "{ Add(new Inventor('Aleksandar Seovic', date('1974-08-24')), 'S
```

As a convenience, Spring.NET also allows you to define named constructor arguments, which are used to set object's properties after instantiation, similar to the way standard .NET attributes work. For example, you could create an instance of the `Inventor` class and set its `Inventions` property in a single statement:

```
Inventor aleks = (Inventor) ExpressionEvaluator.GetValue(null, "new Inventor('Aleksandar Seovic', date('1974-08-24'), 'Serbi
```

The only rule you have to follow is that named arguments should be specified *after* standard constructor arguments, just like in the .NET attributes.

While we are on the subject, Spring.NET Expression Language also provides a convenient syntax for .NET attribute instance creation. Instead of using standard constructor syntax, you can use a somewhat shorter and more familiar syntax to create an instance of a .NET attribute class:

```
WebMethodAttribute webMethod = (WebMethodAttribute) ExpressionEvaluator.GetValue(null, "@[WebMethod(true, CacheDuration = 60
```

As you can see, with the exception of the `@` prefix, syntax is exactly the same as in C#.

Slightly different syntax is not the only thing that differentiates an attribute expression from a standard constructor invocation expression. In addition to that, attribute expression uses slightly different type resolution mechanism and will attempt to load both the specified type name and the specified type name with an `Attribute` suffix, just like the C# compiler.

11.3.10. Variables

Variables can be referenced in the expression using the syntax `#variableName`. The variables are passed in and out of the expression using the dictionary parameter in `ExpressionEvaluator`'s `GetValue` or `SetValue` methods.

```
public static object GetValue(object root, string expression, IDictionary variables)

public static void SetValue(object root, string expression, IDictionary variables, object newValue)
```

The variable name is the key value of the dictionary. Example usage is shown below;

```
IDictionary vars = new Hashtable();
vars["newName"] = "Mike Tesla";
ExpressionEvaluator.GetValue(tesla, "Name = #newName", vars);
```

You can also use the dictionary as a place to store values of the object as they are evaluated inside the expression. For example to change Tesla's first name back again and keep the old value;

```
ExpressionEvaluator.GetValue(tesla, "{ #oldName = Name; Name = 'Nikola Tesla' }", vars);
String oldName = (String)vars["oldName"]; // Mike Tesla
```

Variable names can also be used inside indexers or maps instead of literal values. For example;

```
vars["prez"] = "president";
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers[#prez]", vars);
```

11.3.10.1. The '#this' and '#root' variables

There are two special variables that are always defined and can be references within the expression: #this and #root.

The #this variable can be used to explicitly refer to the context for the node that is currently being evaluated:

```
// sets the name of the president and returns its instance
ExpressionEvaluator.GetValue(ieee, "Officers['president'].( #this.Name = 'Nikola Tesla'; #this )")
```

Similarly, the #root variable allows you to refer to the root context for the expression:

```
// removes president from the Officers dictionary and returns removed instance
ExpressionEvaluator.GetValue(ieee, "Officers['president'].( #root.Officers.Remove('president'); #this )")
```

11.3.11. Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. A minimal example is;

```
String aTrueString = (String) ExpressionEvaluator.GetValue(null, "false ? 'trueExp' : 'falseExp'") // trueExp
```

In this case, the boolean false results in returning the string value 'trueExp'. A less artificial example is shown below

```
ExpressionEvaluator.SetValue(ieee, "Name", "IEEE");
IDictionary vars = new Hashtable();
vars["queryName"] = "Nikola Tesla";

string expression = @"IsMember(#queryName)
    ? #queryName + ' is a member of the ' + Name + ' Society'
    : #queryName + ' is not a member of the ' + Name + ' Society';

String queryResultString = (String) ExpressionEvaluator.GetValue(ieee, expression, vars);

// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

11.3.12. List Projection and Selection

List projection and selection are very powerful expression language features that allow you to transform the source list into another list by either *projecting* across its "columns", or *selecting* from its "rows". In other words, projection can be thought of as a column selector in a SQL SELECT statement, while selection would be comparable to the WHERE clause.

For example, let's say that we need a list of the cities where our inventors were born. This could be easily obtained by projecting on the PlaceOfBirth.City property:

```
IList placesOfBirth = (IList) ExpressionEvaluator.GetValue(ieee, "Members.![PlaceOfBirth.City]") // { 'Smiljan', 'Idvor' }
```

Or we can get the list of officers' names:

```
ILList officersNames = (ILList) ExpressionEvaluator.GetValue(ieee, "Officers.Values.!.{Name}") // { 'Nikola Tesla', 'Mihajlo P
```

As you can see from the examples, projection uses `!.{projectionExpression}` syntax and will return a new list of the same length as the original list but typically with the elements of a different type.

On the other hand, selection, which uses `?{projectionExpression}` syntax, will filter the list and return a new list containing a subset of the original element list. For example, selection would allow us to easily get a list of Serbian inventors:

```
ILList serbianInventors = (ILList) ExpressionEvaluator.GetValue(ieee, "Members.?.{Nationality == 'Serbian'}") // { tesla, pupin }
```

Or to get a list of inventors that invented sonar:

```
ILList sonarInventors = (ILList) ExpressionEvaluator.GetValue(ieee, "Members.?.{'Sonar' in Inventions}") // { pupin }
```

Or we can combine selection and projection to get a list of sonar inventors' names:

```
ILList sonarInventorsNames = (ILList) ExpressionEvaluator.GetValue(ieee, "Members.?.{'Sonar' in Inventions}.!.{Name}") // { 'Mihajlo Pupin' }
```

As a convenience, Spring.NET Expression Language also supports a special syntax for selecting the first or last match. Unlike regular selection, which will return an empty list if no matches are found, first or last match selection expression will either return an instance of the matched element, or `null` if no matching elements were found. In order to return a first match you should prefix your selection expression with `^{` instead of `?{`, and to return last match you should use `${` prefix:

```
ExpressionEvaluator.GetValue(ieee, "Members.^{Nationality == 'Serbian'}.Name") // 'Nikola Tesla'
ExpressionEvaluator.GetValue(ieee, "Members.${Nationality == 'Serbian'}.Name") // 'Mihajlo Pupin'
```

Notice that we access the `Name` property directly on the selection result, because an actual matched instance is returned by the first and last match expression instead of a filtered list.

11.3.13. Collection Processors and Aggregators

In addition to list projection and selection, Spring.NET Expression Language also supports several collection processors, such as `distinct`, `nonNull` and `sort`, as well as a number of commonly used aggregators, such as `max`, `min`, `count`, `sum` and `average`.

The difference between processors and aggregators is that processors return a new or transformed collection, while aggregators return a single value. Other than that, they are very similar -- both processors and aggregators are invoked on a collection node using standard method invocation expression syntax, which makes them very simple to use and allows easy chaining of multiple processors.

11.3.13.1. Count Aggregator

The count aggregator is a safe way to obtain a number of items in a collection. It can be applied to a collection of any type, including arrays, which helps eliminate the decision on whether to use `Count` or `Length` property depending on the context. Unlike its standard .NET counterparts, count aggregator can also be invoked on the `null` context without throwing a `NullReferenceException`. It will simply return zero in this case, which makes it much safer than standard .NET properties within larger expression.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3}.count()") // 3
ExpressionEvaluator.GetValue(null, "count()") // 0
```

11.3.13.2. Sum Aggregator

The sum aggregator can be used to calculate a total for the list of numeric values. If numbers within the list are not of the same type or precision, it will automatically perform necessary conversion and the result will

be the highest precision type. If any of the collection elements is not a number, this aggregator will throw an `InvalidArgumentException`.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.sum()") // 13 (int)
ExpressionEvaluator.GetValue(null, "{5, 5.8, 12.2, 1}.sum()") // 24.0 (double)
```

11.3.13.3. Average Aggregator

The average aggregator will return the average for the collection of numbers. It will use the same type coercion rules, as the sum aggregator in order to be as precise as possible. Just like the sum aggregator, if any of the collection elements is not a number, it will throw an `InvalidArgumentException`.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -4, 10}.average()") // 3
ExpressionEvaluator.GetValue(null, "{1, 5, -2, 10}.average()") // 3.5
```

11.3.13.4. Minimum Aggregator

The minimum aggregator will return the smallest item in the list. In order to determine what "the smallest" actually means, this aggregator relies on the assumption that the collection items are of the uniform type and that they implement the `IComparable` interface. If that is not the case, this aggregator will throw an `InvalidArgumentException`.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.min()") // -3
ExpressionEvaluator.GetValue(null, "{ 'abc', 'efg', 'xyz' }.min()") // 'abc'
```

11.3.13.5. Maximum Aggregator

The maximum aggregator will return the largest item in the list. In order to determine what "the largest" actually means, this aggregator relies on the assumption that the collection items are of the uniform type and that they implement `IComparable` interface. If that is not the case, this aggregator will throw an `InvalidArgumentException`.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.max()") // 10
ExpressionEvaluator.GetValue(null, "{ 'abc', 'efg', 'xyz' }.max()") // 'xyz'
```

11.3.13.6. Non-null Processor

A non-null processor is a very simple collection processor that eliminates all `null` values from the collection.

```
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', null, 'abc', 'def', null }.nonNull()") // { 'abc', 'xyz', 'abc', 'def' }
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', null, 'abc', 'def', null }.nonNull().distinct().sort()") // { 'abc', 'def', 'xyz' }
```

11.3.13.7. Distinct Processor

A distinct processor is very useful when you want to ensure that you don't have duplicate items in the collection. It can also accept an optional `Boolean` argument that will determine whether `null` values should be included in the results. The default is `false`, which means that they will not be included.

```
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', 'abc', 'def', null, 'def' }.distinct(true).sort()") // { null, 'abc', 'def', 'xyz' }
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', 'abc', 'def', null, 'def' }.distinct(false).sort()") // { 'abc', 'def', 'xyz' }
```

11.3.13.8. Sort Processor

The sort processor can be used to sort uniform collections of elements that implement `IComparable`.

```
ExpressionEvaluator.GetValue(null, "{1.2, 5.5, -3.3}.sort()") // { -3.3, 1.2, 5.5 }
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', 'abc', 'def', null, 'def' }.sort()") // { null, 'abc', 'abc', 'def', 'def', 'xyz' }
```

The sort processor also accepts a boolean value as an argument to determine sort order, `sort(false)` will sort the collection in decending order.

11.3.13.9. Type Conversion Processor

The convert processor can be used to convert a collection of elements to a given Type.

```
object[] arr = new object[] { "0", 1, 1.1m, "1.1", 1.1f };
decimal[] result = (decimal[]) ExpressionEvaluator.GetValue(arr, "convert(decimal)");
```

11.3.13.10. Reverse Processor

The reverse processor returns the reverse order of elements in the list

```
object[] arr = new object[] { "0", 1, 2.1m, "3", 4.1f };
object[] result = new ArrayList( (ICollection) ExpressionEvaluator.GetValue(arr, "reverse()") ).ToArray(); // { 4.1f, "3", 2
```

11.3.13.11. OrderBy Processor

Collections can be ordered in three ways, an expression, a SpEL lamda expreession, or a delegate.

```
// orderBy expression
IExpression exp = Expression.Parse("orderBy('ToString()')");
object[] input = new object[] { 'b', 1, 2.0, "a" };
object[] ordered = exp.GetValue(input); // { 1, 2.0, "a", 'b' }

// SpEL lambda expressions
IExpression exp = Expression.Parse("orderBy(|a,b| $a.ToString().CompareTo($b.ToString()))");
object[] input = new object[] { 'b', 1, 2.0, "a" };
object[] ordered = exp.GetValue(input); // { 1, 2.0, "a", 'b' }

Hashtable vars = new Hashtable();
Expression.RegisterFunction( "compare", "{|a,b| $a.ToString().CompareTo($b.ToString())}", vars);
exp = Expression.Parse("orderBy(#compare)");
ordered = exp.GetValue(input, vars); // { 1, 2.0, "a", 'b' }

// .NET delegate
private delegate int CompareCallback(object x, object y);
private int CompareObjects(object x, object y)
{
    if (x == y) return 0;
    return x.ToString().CompareTo(""+y);
}

Hashtable vars = new Hashtable();
vars["compare"] = new CompareCallback(CompareObjects);

IExpression exp = Expression.Parse("orderBy(#compare)");
object[] input = new object[] { 'b', 1, 2.0, "a" };
object[] ordered = exp.GetValue(input); // { 1, 2.0, "a", 'b' }
```

11.3.13.12. User Defined Collection Processor

You can register your own collection processor for use in evaluation a collection. Here is an example of a `ICollectionProcessor` implementation that sums only the even numbers of an integer list

```
public class IntEvenSumCollectionProcessor : ICollectionProcessor
{
    public object Process(ICollection source, object[] args)
    {
        object total = 0d;
        foreach (object item in source)
        {
            if (item != null)
            {
                if (NumberUtils.IsInteger(item))
            
```

```

        {
            if ((int)item % 2 == 0)
            {
                total = NumberUtils.Add(total, item);
            }
            else
            {
                throw new ArgumentException("Sum can only be calculated for a collection of numeric values.");
            }
        }
    }

    return total;
}

public void DoWork()
{
    Hashtable vars = new Hashtable();
    vars["EvenSum"] = new IntEvenSumCollectionProcessor();
    int result = (int)ExpressionEvaluator.GetValue(null, "{1, 2, 3, 4}.EvenSum()", vars); // 6
}

```

11.3.14. Spring Object References

Expressions can refer to objects that are declared in Spring's application context using the syntax `@(contextName:objectName)`. If no `contextName` is specified the default root context name (`Spring.RootContext`) is used. Using the application context defined in the *MovieFinder* example from Chapter 35, *IoC Quickstarts*, the following expression returns the number of movies directed by Roberto Benigni.

```

public static void Main()
{
    . . .

    // Retrieve context defined in the spring/context section of
    // the standard .NET configuration file.
    IApplicationContext ctx = ContextRegistry.GetContext();

    int numMovies = (int) ExpressionEvaluator.GetValue(null,
        @"(MyMovieLister).MoviesDirectedBy('Roberto Benigni').Length");

    . . .
}

```

The variable `numMovies` is evaluated to 2 in this example.

11.3.15. Lambda Expressions

A somewhat advanced, but a very powerful feature of Spring.NET Expression Language are lambda expressions. Lambda expressions allow you to define inline functions, which can then be used within your expressions just like any other function or method. You may also use .NET delegates as described in the next section.

The syntax for defining lambda expressions is:

```
#functionName = { |argList| functionBody }
```

For example, you could define a `max` function and call it like this:

```
ExpressionEvaluator.GetValue(null, "(#max = { |x,y| $x > $y ? $x : $y }; #max(5,25))", new Hashtable()) // 25
```

As you can see, any arguments defined for the expression can be referenced within the function body using a *local variable* syntax, `$varName`. Invocation of the function defined using lambda expression is as simple as specifying the comma-separated list of function arguments in parentheses, after the function name.

Lambda expressions can be recursive, which means that you can invoke the function within its own body:

```
ExpressionEvaluator.GetValue(null, "(#fact = {|n| $n <= 1 ? 1 : $n * #fact($n-1) }; #fact(5))", new Hashtable()) // 120
```

Notice that in both examples above we had to specify a `variables` parameter for the `GetValue` method. This is because lambda expressions are actually nothing more than parameterized variables and we need variables dictionary in order to store them. If you don't specify a valid `IDictionary` instance for the `variables` parameter, you will get a runtime exception.

Also, in both examples above we used an expression list in order to define and invoke a function in a single expression. However, more likely than not, you will want to define your functions once and then use them within as many expressions as you need. Spring.NET provides an easy way to pre-register your lambda expressions by exposing a static `Expression.RegisterFunction` method, which takes function name, lambda expression and variables dictionary to register function in as parameters:

```
IDictionary vars = new Hashtable();
Expression.RegisterFunction("sqrt", "{|n| Math.Sqrt($n)}", vars);
Expression.RegisterFunction("fact", "{|n| $n <= 1 ? 1 : $n * #fact($n-1)}", vars);
```

Once the function registration is done, you can simply evaluate an expression that uses these functions, making sure that the `vars` dictionary is passed as a parameter to expression evaluation engine:

```
ExpressionEvaluator.GetValue(null, "#fact(5)", vars) // 120
ExpressionEvaluator.GetValue(null, "#sqrt(9)", vars) // 3
```

Finally, because lambda expressions are treated as variables, they can be assigned to other variables or passed as parameters to other lambda expressions. In the following example we are defining a delegate function that accepts function `f` as the first argument and parameter `n` that will be passed to function `f` as the second. Then we invoke the functions registered in the previous example, as well as the lambda expression defined inline, through our delegate:

```
Expression.RegisterFunction("delegate", "{|f, n| $f($n) }", vars);
ExpressionEvaluator.GetValue(null, "#delegate(#sqrt, 4)", vars) // 2
ExpressionEvaluator.GetValue(null, "#delegate(#fact, 5)", vars) // 120
ExpressionEvaluator.GetValue(null, "#delegate({|n| $n ^ 2 }, 5)", vars) // 25
```

While this particular example is not particularly useful, it does demonstrate that lambda expressions are indeed treated as nothing more than parameterized variables, which is important to remember.

11.3.16. Delegate Expressions

Delegate expressions allow you to refer to .NET delegates which can then be used within your expressions just like any other function or method.

For example, you can define a max delegate and call it like this

```
private delegate double DoubleFunctionTwoArgs(double arg1, double arg2);

private double Max(double arg1, double arg2)
{
    return Math.Max(arg1, arg2);
}

public void DoWork()
{
    Hashtable vars = new Hashtable();
    vars["max"] = new DoubleFunctionTwoArgs(Max);
    double result = (double) ExpressionEvaluator.GetValue(null, "#max(5,25)", vars); // 25
}
```

11.3.17. Null Context

If you do not specify a root object, i.e. pass in null, then the expressions evaluated either have to be literal values, i.e. `ExpressionEvaluator.GetValue(null, "2 + 3.14")`, refer to classes that have static methods or properties, i.e. `ExpressionEvaluator.GetValue(null, "DateTime.Today")`, create new instances of objects, i.e. `ExpressionEvaluator.GetValue(null, "new DateTime(2004, 8, 14)")` or refer to other objects such as those in the variable dictionary or in the IoC container. The latter two usages will be discussed later.

11.4. Classes used in the examples

The following simple classes are used to demonstrate the functionality of the expression language.

```
public class Inventor
{
    public string Name;
    public string Nationality;
    public string[] Inventions;
    private DateTime dob;
    private Place pob;

    public Inventor() : this(null, DateTime.MinValue, null)
    {}

    public Inventor(string name, DateTime dateOfBirth, string nationality)
    {
        this.Name = name;
        this.dob = dateOfBirth;
        this.Nationality = nationality;
        this.pob = new Place();
    }

    public DateTime DOB
    {
        get { return dob; }
        set { dob = value; }
    }

    public Place PlaceOfBirth
    {
        get { return pob; }
    }

    public int GetAge(DateTime on)
    {
        // not very accurate, but it will do the job ;-)
        return on.Year - dob.Year;
    }
}

public class Place
{
    public string City;
    public string Country;
}

public class Society
{
    public string Name;
    public static string Advisors = "advisors";
    public static string President = "president";

    private IList members = new ArrayList();
    private IDictionary officers = new Hashtable();

    public IList Members
    {
        get { return members; }
    }
}
```

```

public IDictionary Officers
{
    get { return officers; }
}

public bool IsMember(string name)
{
    bool found = false;
    foreach (Inventor inventor in members)
    {
        if (inventor.Name == name)
        {
            found = true;
            break;
        }
    }
    return found;
}
}

```

The code listings in this chapter use instances of the data populated with the following information.

```

Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1856, 7, 9), "Serbian");
tesla.Inventions = new string[]
{
    "Telephone repeater", "Rotating magnetic field principle",
    "Polyphase alternating-current system", "Induction motor",
    "Alternating-current power transmission", "Tesla coil transformer",
    "Wireless communication", "Radio", "Fluorescent lights"
};
tesla.PlaceOfBirth.City = "Smiljan";

Inventor pupin = new Inventor("Mihajlo Pupin", new DateTime(1854, 10, 9), "Serbian");
pupin.Inventions = new string[] { "Long distance telephony & telegraphy", "Secondary X-Ray radiation", "Sonar" };
pupin.PlaceOfBirth.City = "Idvor";
pupin.PlaceOfBirth.Country = "Serbia";

Society ieee = new Society();
ieee.Members.Add(tesla);
ieee.Members.Add(pupin);
ieee.Officers["president"] = pupin;
ieee.Officers["advisors"] = new Inventor[] { tesla, pupin };

```

Chapter 12. Validation Framework

12.1. Introduction

Data validation is a very important part of any enterprise application. ASP.NET has a validation framework but it is very limited in scope and starts falling apart as soon as you need to perform more complex validations. Problems with the out of the box ASP.NET validation framework are well [documented](#) by Peter Blum on his web site, so we are not going to repeat them here. Peter has also built a nice replacement for the standard ASP.NET validation framework, which is worth looking into if you prefer the standard ASP.NET validation mechanism to the one offered by Spring.NET for some reason. Both frameworks will allow you to perform very complex validations but we designed the Spring.NET validation framework differently for the reasons described below.

On the Windows Forms side the situation is even worse. Out of the box data validation features are completely inadequate as pointed out by Ian Griffiths in this [article](#). One of the major problems we saw in most validation frameworks available today, both open source and commercial, is that they are tied to a specific presentation technology. The ASP.NET validation framework uses ASP.NET controls to define validation rules, so these rules end up in the HTML markup of your pages. Peter Blum's framework uses the same approach. In our opinion, validation is not applicable only to the presentation layer so there is no reason to tie it to any particular technology. As such, the Spring.NET Validation Framework is designed in a way that enables data validation in different application layers using the same validation rules.

The goals of the validation framework are the following:

1. Allow for the validation of any object, whether it is a UI control or a domain object.
2. Allow the same validation framework to be used in both Windows Forms and ASP.NET applications, as well as in the service layer (to validate parameters passed to the service, for example).
3. Allow composition of the validation rules so arbitrarily complex validation rule sets can be constructed.
4. Allow validators to be conditional so they only execute if a specific condition is met.

The following sections will describe in more detail how these goals were achieved and show you how to use the Spring.NET Validation Framework in your applications.

12.2. Example Usage

Decoupling validation from presentation was the major goal that significantly influenced design of the validation framework. We wanted to be able to define a set of validation rules that are completely independent from the presentation so we can reuse them (or at least have the ability to reuse them) in different application layers. This meant that the approach taken by Microsoft ASP.NET team would not work and custom validation controls were not an option. The approach taken was to configure validation rules just like any other object managed by Spring - within the application context. However, due to possible complexity of the validation rules we decided not to use the standard Spring.NET configuration schema for validator definitions but to instead provide a more specific and easier to use custom configuration schema for validation. Note that the validation framework is not tied to the use of XML, you can use its API Programatically. The following example shows validation rules defined for the Trip object in the SpringAir sample application:

```
<objects xmlns="http://www.springframework.net" xmlns:v="http://www.springframework.net/validation">
```

```

<object type="TripForm.aspx" parent="standardPage">
  <property name="TripValidator" ref="tripValidator" />
</object>

<v:group id="tripValidator">

  <v:required id="departureAirportValidator" test="StartingFrom.AirportCode">
    <v:message id="error.departureAirport.required" providers="departureAirportErrors, validationSummary"/>
  </v:required>

  <v:group id="destinationAirportValidator">
    <v:required test="ReturningFrom.AirportCode">
      <v:message id="error.destinationAirport.required" providers="destinationAirportErrors, validationSummary"/>
    </v:required>
    <v:condition test="ReturningFrom.AirportCode != StartingFrom.AirportCode" when="ReturningFrom.AirportCode != ''">
      <v:message id="error.destinationAirport.sameAsDeparture" providers="destinationAirportErrors, validationSummary"/>
    </v:condition>
  </v:group>

  <v:group id="departureDateValidator">
    <v:required test="StartingFrom.Date">
      <v:message id="error.departureDate.required" providers="departureDateErrors, validationSummary"/>
    </v:required>
    <v:condition test="StartingFrom.Date >= DateTime.Today" when="StartingFrom.Date != DateTime.MinValue">
      <v:message id="error.departureDate.inThePast" providers="departureDateErrors, validationSummary"/>
    </v:condition>
  </v:group>

  <v:group id="returnDateValidator" when="Mode == 'RoundTrip'">
    <v:required test="ReturningFrom.Date">
      <v:message id="error.returnDate.required" providers="returnDateErrors, validationSummary"/>
    </v:required>
    <v:condition test="ReturningFrom.Date >= StartingFrom.Date" when="ReturningFrom.Date != DateTime.MinValue">
      <v:message id="error.returnDate.beforeDeparture" providers="returnDateErrors, validationSummary"/>
    </v:condition>
  </v:group>

</v:group>
</objects>

```

There are a few things to note in the example above:

- You need to reference the validation schema by adding a `xmlns:v="http://www.springframework.net/validation"` namespace declaration to the root element.
- You can mix standard object definitions and validator definitions in the same configuration file as long as both schemas are referenced.
- The Validator defined in the configuration file is identified by an `id` attribute and can be referenced in the standard Spring way, i.e. the injection of `tripValidator` into `TripForm.aspx` page definition in the first `<object>` tag above.
- The validation framework uses Spring's powerful expression evaluation engine to evaluate both validation rules and applicability conditions for the validator. As such, any valid Spring expression can be specified within the `test` and `when` attributes of any validator.

The example above shows many of the features of the framework, so let's discuss them one by one in the following sections.

12.3. Validator Groups

Validators can be grouped together. This is important for many reasons but the most typical usage scenario is to group multiple validation rules that apply to the same value. In the example above there is a validator group for

almost every property of the Trip instance. There is also a top-level group for the Trip object itself that groups all other validators.

There are three types of validator groups each with a different behavior:

While the first type (AND) is definitely the most useful, the other two allow you to implement some specific validation scenarios in a very simple way, so you should keep them in mind when designing your validation rules.

Table 12.1. Validator Groups

Type	XML Tag	Behavior
AND	group	Returns true only if all contained validators return true . This is the most commonly used validator group.
OR	any	Returns true if one or more of the contained validators return true .
XOR	exclusive	Returns true if only one of the contained validators return true.

One thing to remember is that a validator group is a validator like any other and can be used anywhere validator is expected. You can nest groups within other groups and reference them using validator reference syntax (described later), so they really allow you to structure your validation rules in the most reusable way.

12.4. Validators

Ultimately, you will have one or more validator definitions for each piece of data that you want to validate. Spring.NET has several built-in validators that are sufficient for most validations, even fairly complex ones. The framework is extensible so you can write your own custom validators and use them in the same way as the built-in ones.

12.4.1. Condition Validator

The condition validator evaluates any logical expression that is supported by Spring's evaluation engine. The syntax is

```
<v:condition id="id" test="testCondition" when="applicabilityCondition" parent="parentValidator">
  actions
</v:condition>
```

An example is shown below

```
<v:condition test="StartingFrom.Date >= DateTime.Today" when="StartingFrom.Date != DateTime.MinValue">
  <v:message id="error.departureDate.inThePast" providers="departureDateErrors, validationSummary"/>
</v:condition>
```

In this example the StartingFrom property of the Trip object is compared to see if it is later than the current date, i.e. DateTime but only when the date has been set (the initial value of StartingFrom.Date was set to DateTime.MinValue).

The condition validator could be considered "the mother of all validators". You can use it to achieve almost anything that can be achieved by using other validator types, but in some cases the test expression might be very complex, which is why you should use more specific validator type if possible. However, condition validator is still your best bet if you need to check whether particular value belongs to a particular range, or perform a similar test, as those conditions are fairly easy to write.



Note

Keep in mind that Spring.NET Validation Framework typically works with domain objects. This is after data binding from the controls has been performed so that the object being validated is strongly typed. This means that you can easily compare numbers and dates without having to worry if the string representation is comparable.

12.4.2. Required Validator

This validator ensures that the specified test value is not empty. The syntax is

```
<v:required id="id" test="requiredValue" when="applicabilityCondition" parent="parentValidator">
  actions
</v:required>
```

An example is shown below

```
<v:required test="ReturningFrom.AirportCode">
  <v:message id="error.destinationAirport.required" providers="destinationAirportErrors, validationSummary"/>
</v:required>
```

The specific tests done to determine if the required value is set is listed below

Table 12.2. Rules to determine if required value is valid

System.Type	Test
System.Type	Type exists
System.String	not null or an empty string
system.DateTime	Not System.DateTime.MinValue and not system.DateTime.MaxValue
One of the number types.	not zero
System.Char	Not System.Char.MinValue or whitespace.
Any reference type other than System.String	not null

Required validator is also one of the most commonly used ones, and it is much more powerful than the ASP.NET Required validator, because it works with many other data types other than strings. For example, it will allow you to validate `DateTime` instances (both `MinValue` and `MaxValue` return `false`), integer and decimal numbers, as well as any reference type, in which case it returns `true` for a non-null value and `false` for `{null}`s.

The test attribute for the required validator will typically specify an expression that resolves to a property of a domain object, but it could be any valid expression that returns a value, including a method call.

12.4.3. Regular Expression Validator

The syntax is

```
<v:regex id="id" test="valueToEvaluate" when="applicabilityCondition" parent="parentValidator">
  <v:property name="Expression" value="regularExpressionToMatch"/>
  <v:property name="Options" value="regexOptions"/>
  actions
</v:regex>
```

An example is shown below

```
<v:regex test="ReturningFrom.AirportCode">
  <v:property name="Expression" value="[A-Z][A-Z][A-Z]" />
  <v:message id="error.destinationAirport.threeCharacters" providers="destinationAirportErrors, validationSummary" />
</v:regex>
```

Regular expression validator is very useful when validating values that need to conform to some predefined format, such as telephone numbers, email addresses, URLs, etc.

One major difference of the regular expression validator compared to other built-in validator types is that you need to set a required `Expression` property to a regular expression to match against.

12.4.4. Generic Validator

The syntax is

```
<v:validator id="id" test="requiredValue" when="applicabilityCondition" type="validatorType" parent="parentValidator">
  actions
</v:validator>
```

An example is shown below

```
<v:validator test="ReturningFrom.AirportCode" type="MyNamespace.MyAirportCodeValidator, MyAssembly">
  <v:message id="error.destinationAirport.invalid" providers="destinationAirportErrors, validationSummary" />
</v:required>
```

Generic validator allows you to plug in your custom validator by specifying its type name. Custom validators are very simple to implement, because all you need to do is extend `BaseValidator` class and implement abstract `bool Validate(object objectToValidate)` method. Your implementation simply needs to return `true` if it determines that object is valid, or `false` otherwise

12.4.5. Conditional Validator Execution

As you can see from the examples above, each validator (and validator group) allows you to define its applicability condition by specifying a logical expression as the value of the `when` attribute. This feature is very useful and is one of the major deficiencies in the standard ASP.NET validation framework, because in many cases specific validators need to be turned on or off based on the values of the object being validated.

For example, when validating a `Trip` object we need to validate return date only if the `Trip.Mode` property is set to the `TripMode.RoundTrip` enum value. In order to achieve that we created following validator definition:

```
<v:group id="returnDateValidator" when="Mode == 'RoundTrip'">
  // nested validators
</v:group>
```

Validators within this group will only be evaluated for round trips.



Note

You should also note that you can compare enums using the string value of the enumeration. You can also use fully qualified enum name, such as:

```
Mode == TripMode.RoundTrip
```

However, in this case you need to make sure that alias for the `TripMode` enum type is registered using Spring's standard type aliasing mechanism.

12.5. Validator Actions

Validation actions are executed every time the containing validator is executed. They allow you to do anything you want based on the result of the validation. By far the most common use of the validation action is to add validation error message to the errors collection, but theoretically you could do anything you want. Because adding validation error messages to the errors collection is such a common scenario, Spring.NET validation schema defines a separate XML tag for this type of validation action.

12.5.1. Error Message Action

The syntax is

```
<v:message id="messageId" providers="errorProviderList" when="messageApplicabilityCondition">
  <v:param value="paramExpression"/>
</v:message>
```

An example is shown below

```
<v:message id="error.departureDate.inThePast" providers="departureDateErrors, validationSummary">
  <v:param value="StartingFrom.Date.ToString('D')"/>
  <v:param value="DateTime.Today.ToString('D')"/>
</v:message>
```

There are several things that you have to be aware of when dealing with error messages:

- `id` is used to look up the error message in the appropriate Spring.NET message source.
- `providers` specifies a comma separated list of "error buckets" particular error message should be added to. These "buckets" will later be used by the particular presentation technology in order to display error messages as necessary.
- a message can have zero or more parameters. Each parameter is an expression that will be resolved using current validation context and the resolved values will be passed as parameters to `IMessageSource.GetMessage` method, which will return the fully resolved message.

12.5.2. Exception Action

If you would like an exception to be thrown when validation fails use the exception action.

```
<v:exception/>
```

This will throw an exception of the type `ValidationException` and you can access error information via its `ValidationErrors` property. To throw your own custom exception, provide a SpEL fragment that instantiates the custom exception.

```
<v:exception throw='new System.InvalidOperationException("invalid")'/>
```

12.5.3. Generic Actions

The syntax is

```
<v:action type="actionType" when="actionApplicabilityCondition">
  properties
</v:action>
```

An example is shown below

```
<v:action type="Spring.Validation.Actions.ExpressionAction, Spring.Core" when="#page != null">
```

```
<v:property name="Valid" value="#page.myPanel.Visible = true"/>
<v:property name="Invalid" value="#page.myPanel.Visible = false"/>
</v:action>
```

Generic actions can be used to perform all kinds of validation actions. In simple cases, such as in the example above where we turn control's visibility on or off depending on the validation result, you can use the built-in `ExpressionAction` class and simply specify expressions to be evaluated based on the validator result.

In other situations you may want to create your own action implementation, which is fairly simple thing to do – all you need to do is implement `IVValidationAction` interface:

```
public interface IValidationAction
{
    /// <summary>
    /// Executes the action.
    /// </summary>
    /// <param name="isValid">Whether associated validator is valid or not.</param>
    /// <param name="validationContext">Validation context.</param>
    /// <param name="contextParams">Additional context parameters.</param>
    /// <param name="errors">Validation errors container.</param>
    void Execute(bool isValid, object validationContext, IDictionary contextParams, ValidationErrors errors);
}
```

12.6. Validator References

Sometimes it is not possible (or desirable) to nest all the validation rules within a single top-level validator group. For example, if you have an object graph where both `ObjectA` and `ObjectB` have a reference to `ObjectC`, you might want to set up validation rules for `ObjectC` only once and reference them from the validation rules for both `ObjectA` and `ObjectB`, instead of duplicating them within both definitions.

The syntax is shown below

```
<v:ref name="referencedValidatorId" context="validationContextForTheReferencedValidator"/>
```

An example is shown below

```
<v:group id="objectA.validator">
    <v:ref name="objectC.validator" context="MyObjectC"/>
    // other validators for ObjectA
</v:group>

<v:group id="objectB.validator">
    <v:ref name="objectC.validator" context="ObjectCProperty"/>
    // other validators for ObjectB
</v:group>

<v:group id="objectC.Validator">
    // validators for ObjectC
</v:group>
```

It is as simple as that — you define validation rules for `ObjectC` separately and reference them from within other validation groups. Important thing to realize that in most cases you will also want to "narrow" the context for the referenced validator, typically by specifying the name of the property that holds referenced object. In the example above, `ObjectA.MyObjectC` and `ObjectB.ObjectCProperty` are both of type `ObjectC`, which `objectC.validator` expects to receive as the validation context.

12.7. Programmatic usage

You can also create Validators programmatically using the API. An example is shown below

```
UserInfo userInfo = new UserInfo(); // has Name and Password props
```

```
ValidatorGroup userInfoValidator = new ValidatorGroup();

userInfoValidator.Validators
    .Add(new RequiredValidator("Name", null));

userInfoValidator.Validators
    .Add(new RequiredValidator("Password", null));

ValidationErrors errors = new ValidationErrors();
bool userInfoIsValid = userInfoValidator.Validate(userInfo, errors);
```

No matter if you create your validators programmatically or declaratively, you can invoke them in service side code via the 'Validate' method shown above and then handle error conditions. Spring provides AOP parameter validation advice as part of the aspect library which may also be useful for performing server-side validation.

12.8. Usage tips within ASP.NET

Now that you know how to configure validation rules, let's see what it takes to evaluate those rules within your typical ASP.NET application and to display error messages.

The first thing you need to do is inject validators you want to use into your ASP.NET page, as shown in the example below:

```
<objects xmlns="http://www.springframework.net" xmlns:v="http://www.springframework.net/validation">

  <object type="TripForm.aspx" parent="standardPage">
    <property name="TripValidator" ref="tripValidator" />
  </object>

  <v:group id="tripValidator">
    <v:required id="departureAirportValidator" test="StartingFrom.AirportCode">
      <!-- write error message to 2 providers -->
      <v:message id="error.departureAirport.required" providers="departureAirportErrors, errorSummary"/>
    </v:required>

    <v:group id="destinationAirportValidator">
      <v:required test="ReturningFrom.AirportCode">
        <!-- write error message to 2 providers -->
        <v:message id="error.destinationAirport.required" providers="destinationAirportErrors, errorSummary"/>
      </v:required>
    </v:group>
  </v:group>

</objects>
```

Once that's done, you need to perform validation in one or more of the page event handlers, which typically looks similar to this:

```
public void SearchForFlights(object sender, EventArgs e)
{
    if (Validate(Controller.Trip, tripValidator))
    {
        Process.SetView(Controller.SearchForFlights());
    }
}
```



Note

Keep in mind that your ASP.NET page needs to extend `Spring.Web.UI.Page` in order for the code above to work.

Finally, you need to define where validation errors should be displayed by adding one or more `<spring:validationError/>` and `<spring:validationSummary/>` controls to the ASP.NET form:

```

<!-- code snippet taken from the SpringAir sample application -->
<%@ Page Language="c#" MasterPageFile="~/Web/StandardTemplate.master" Inherits="TripForm" CodeFile="TripForm.aspx.cs" %>

    <!-- render all error messages sent to 'errorSummary' provider -->
    <spring:ValidationSummary ID="summary" Provider="errorSummary" runat="server" />
    <table>
        <tr>
            <td>
                <asp:Label ID="leavingFrom" runat="server" /></td>
            <td>
                <asp:DropDownList ID="leavingFromAirportCode" AutoPostBack="true" runat="server" />
                <!-- render error messages sent to 'departureAirportErrors' provider -->
                <spring:ValidationError ID="leavingFromError" Provider="departureAirportErrors" runat="server" />
            </td>
            <td>
                <asp:Label ID="goingTo" runat="server" /></td>
            <td>
                <asp:DropDownList ID="goingToAirportCode" AutoPostBack="true" runat="server" />
                <!-- render error messages sent to 'destinationAirportErrors' provider -->
                <spring:ValidationError ID="goingToError" Provider="destinationAirportErrors" runat="server" />
            </td>
        </tr>
    </table>

```

12.8.1. Rendering Validation Errors

Spring.NET allows you to render validation errors within the page in several different ways, and if none of them suits your needs you can implement your own validation errors renderer. Implementations of the `Spring.Web.Validation.IValidationErrorsRenderer` that ship with the framework are:

Table 12.3. Validation Renderers

Name	Class	Description
Block	<code>Spring.Web.Validation.DivValidationErrorsRenderer</code>	Renders validation errors as list items within a <code><div></code> tag. Default renderer for <code><spring:validationSummary></code> control.
Inline	<code>Spring.Web.Validation.SpanValidationErrorsRenderer</code>	Renders validation errors within a <code></code> tag. Default renderer for <code><spring:validationError></code> control.
Icon	<code>Spring.Web.Validation.IconValidationErrorsRenderer</code>	Renders validation errors as error icon, with error messages displayed in a tooltip. Best option when saving screen real estate is important.

These three error renderers should be sufficient for most applications, but in case you want to display errors in some other way you can write your own renderer by implementing `Spring.Web.Validation.IValidationErrorsRenderer` interface:

```

namespace Spring.Web.Validation
{
    /// <summary>
    /// This interface should be implemented by all validation errors renderers.
    /// </summary>
    /// <remarks>
    /// <para>
    /// Validation errors renderers are used to decouple rendering behavior from the
    /// validation errors controls such as <see cref="ValidationError"/> and
    /// <see cref="ValidationSummary"/>.
    /// </para>
    /// <para>
    /// This allows users to change how validation errors are rendered by simply plugging in
    /// appropriate renderer implementation into the validation errors controls using
    /// Spring.NET dependency injection.
    /// </para>
    /// </remarks>
    public interface IValidationErrorsRenderer
    {

```

```

    /// <summary>
    /// Renders validation errors using specified <see cref="HtmlTextWriter"/>.
    /// </summary>
    /// <param name="page">Web form instance.</param>
    /// <param name="writer">An HTML writer to use.</param>
    /// <param name="errors">The list of validation errors.</param>
    void RenderErrors(Page page, HtmlTextWriter writer, IList errors);
}

```

12.8.1.1. Configuring which Error Renderer to use.

The best part of the errors renderer mechanism is that you can easily change it across the application by modifying configuration templates for `<spring:validationSummary>` and `<spring:validationError>` controls:

```

<!-- Validation errors renderer configuration -->
<object id="Spring.Web.UI.Controls.ValidationSummary" abstract="true">
  <property name="Renderer">
    <object type="Spring.Web.Validation.IconValidationErrorsRenderer, Spring.Web">
      <property name="IconSrc" value="validation-error.gif"/>
    </object>
  </property>
</object>

<object id="Spring.Web.UI.Controls.ValidationSummary" abstract="true">
  <property name="Renderer">
    <object type="Spring.Web.Validation.DivValidationErrorsRenderer, Spring.Web">
      <property name="CssClass" value="validationError"/>
    </object>
  </property>
</object>

```

It's as simple as that!

12.8.2. How Validate() and Validation Controls play together

Validation Controls (ValidationSummary, ValidationError) need to somehow get the list of errors collected during validation. Both, `Spring.Web.UI.Page` and `Spring.Web.UI.UserControl` come with a `ValidationErrors` property and implement `IVValidationContainer`. ValidationControls will automatically pick the `IVValidationContainer` control they are placed on:

```

// ASPX / ASCX Template Code
<%@ Control Language="C#" %>

    <!-- render all error messages sent to 'errorSummary' provider -->
    <spring:ValidationSummary ID="summary" Provider="errorSummary" runat="server" />

    <asp:DropDownList ID="leavingFromAirportCode" AutoPostBack="true" runat="server" />
    <!-- render error messages sent to 'departureAirportErrors' provider -->
    <spring:ValidationSummary ID="leavingFromError" Provider="departureAirportErrors" runat="server" />

<script language="C#" runat="server">
public void SearchForFlights(object sender, EventArgs e)
{
    if (Validate(Controller.Trip, tripValidator))
    {
        Process.SetView(Controller.SearchForFlights());
    }
}
</script>

```

If you need to render errors from a UserControl not in the hierarchy of your Validation control, you can specify the name of the target validation container control:

```

// ASPX / ASCX Template Code

```

```
<%@ Page Language="c#" %>
<%@ Register TagPrefix="user" TagName="EmployeeInfoEditor" Src="EmployeeInfoEditor.ascx" %>

    <spring:ValidationSummary ID="summary" Provider="summary" ValidationContainerName="editor" runat="server" />
    <user:EmployeeInfoEditor ID="editor" runat="server" />
```

Chapter 13. Aspect Oriented Programming with Spring.NET

13.1. Introduction

Aspect-Oriented Programming (AOP) complements OOP by providing another way of thinking about program structure. Whereas OO decomposes applications into a hierarchy of objects, AOP decomposes programs into *aspects* or *concerns*. This enables the modularization of concerns such as transaction management that would otherwise cut across multiple objects (such concerns are often termed *crosscutting* concerns).

One of the key components of Spring.NET is the *AOP framework*. While the Spring.NET IoC container does not depend on AOP, meaning you don't need to use AOP if you don't want to, AOP complements Spring.NET IoC to provide a very capable middleware solution.

AOP is used in Spring.NET:

- To provide declarative enterprise services, especially as a replacement for COM+ declarative services. The most important such service is *declarative transaction management*, which builds on Spring.NET's transaction abstraction. This functionality is planned for an upcoming release of Spring.NET
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Thus you can view Spring.NET AOP as either an enabling technology that allows Spring.NET to provide declarative transaction management without COM+; or use the full power of the Spring.NET AOP framework to implement custom aspects.

For those who would like to hit the ground running and start exploring how to use Spring's AOP functionality, head on over to Chapter 36, *AOP QuickStart*.

13.1.1. AOP concepts

Let us begin by defining some central AOP concepts. These terms are not Spring.NET-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring.NET used its own terminology.

- *Aspect*: A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in enterprise applications. Aspects are implemented using Spring.NET as Advisors or interceptors.
- *Joinpoint*: Point during the execution of a program, such as a method invocation or a particular exception being thrown.
- *Advice*: Action taken by the AOP framework at a particular joinpoint. Different types of advice include "around," "before" and "throws" advice. Advice types are discussed below. Many AOP frameworks, including Spring.NET, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.
- *Pointcut*: A set of joinpoints specifying when an advice should fire. An AOP framework must allow developers to specify pointcuts: for example, using regular expressions.
- *Introduction*: Adding methods or fields to an advised class. Spring.NET allows you to introduce new interfaces to any advised object. For example, you could use an introduction to make any object implement an *IAuditable* interface, to simplify the tracking of changes to an object's state.

- *Target object*: Object containing the joinpoint. Also referred to as *advised* or *proxied* object.
- *AOP proxy*: Object created by the AOP framework, including advice. In Spring.NET, an AOP proxy is a dynamic proxy that uses IL code generated at runtime.
- *Weaving*: Assembling aspects to create an advised object. This can be done at compile time (using the Gripper-Loom.NET compiler, for example), or at runtime. Spring.NET performs weaving at runtime.

Different advice types include:

- *Around advice*: Advice that surrounds a joinpoint such as a method invocation. This is the most powerful kind of advice. Around advice will perform custom behaviour before and after the method invocation. They are responsible for choosing whether to proceed to the joinpoint or to shortcut executing by returning their own return value or throwing an exception.
- *Before advice*: Advice that executes before a joinpoint, but which does not have the ability to prevent execution flow proceeding to the joinpoint (unless it throws an exception).
- *Throws advice*: Advice to be executed if a method throws an exception. Spring.NET provides strongly typed throws advice, so you can write code that catches the exception (and subclasses) you're interested in, without needing to cast from Exception.
- *After returning advice*: Advice to be executed after a joinpoint completes normally: for example, if a method returns without throwing an exception.

Spring.NET provides a full range of advice types. We recommend that you use the least powerful advice type that can implement the required behaviour. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you don't need to invoke the `proceed()` method on the `IMethodInvocation` used for around advice, and hence can't fail to invoke it.

The pointcut concept is the key to AOP, distinguishing AOP from older technologies offering interception. Pointcuts enable advice to be targeted independently of the OO hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects. Thus pointcuts provide the structural element of AOP.

13.1.2. Spring.NET AOP capabilities

Spring.NET AOP is implemented in pure C#. There is no need for a special compilation process - all weaving is done at runtime. Spring.NET AOP does not need to control or modify the way in which assemblies are loaded, nor does it rely on unmanaged APIs, and is thus suitable for use in any CLR environment.

Spring.NET currently supports interception of method invocations. Field interception is not implemented, although support for field interception could be added without breaking the core Spring.NET AOP APIs.

Field interception arguably violates OO encapsulation. We don't believe it is wise in application development.

Spring.NET provides classes to represent pointcuts and different advice types. Spring.NET uses the term *advisor* for an object representing an aspect, including both an advice and a pointcut targeting it to specific joinpoints.

Different advice types are `IMethodInterceptor` (from the AOP Alliance interception API); and the advice interfaces defined in the `Spring.Aop` namespace. All advices must implement the `AopAlliance.Aop.IAdvice`

tag interface. Advices supported out the box are `IMethodInterceptor` ; `IThrowsAdvice`; `IBeforeAdvice`; and `IAfterReturningAdvice`. We'll discuss advice types in detail below.

Spring.NET provides a .NET translation of the Java interfaces defined by the [AOP Alliance](#). Around advice must implement the AOP Alliance `AopAlliance.Interceptor.IMethodInterceptor` interface. Whilst there is wide support for the AOP Alliance in Java, Spring.NET is currently the only .NET AOP framework that makes use of these interfaces. In the short term, this will provide a consistent programming model for those doing development in both .NET and Java, and in the longer term, we hope to see more .NET projects adopt the AOP Alliance interfaces.

The aim of Spring.NET AOP support is not to provide a comprehensive AOP implementation on par with the functionality available in AspectJ. However, Spring.NET AOP provides an excellent solution to most problems in .NET applications that are amenable to AOP.

Thus, it is common to see Spring.NET's AOP functionality used in conjunction with a Spring.NET IoC container. AOP advice is specified using normal object definition syntax (although this allows powerful "autoproxying" capabilities); advice and pointcuts are themselves managed by Spring.NET IoC.

13.1.3. AOP Proxies in Spring.NET

Spring.NET generates AOP proxies at runtime using classes from the `System.Reflection.Emit` namespace to create necessary IL code for the proxy class. This results in proxies that are very efficient and do not impose any restrictions on the inheritance hierarchy.

Another common approach to AOP proxy implementation in .NET is to use `ContextBoundObject` and the .NET remoting infrastructure as an interception mechanism. We are not very fond of `ContextBoundObject` approach because it requires classes that need to be proxied to inherit from the `ContextBoundObject` either directly or indirectly. In our opinion this an unnecessary restriction that influences how you should design your object model and also excludes applying AOP to "3rd party" classes that are not under your direct control. Context-bound proxies are also an order of magnitude slower than IL-generated proxies, due to the overhead of the context switching and .NET remoting infrastructure.

Spring.NET AOP proxies are also "smart" - in that because proxy configuration is known during proxy generation, the generated proxy can be optimized to invoke target methods via reflection only when necessary (i.e. when there are advices applied to the target method). In all other cases the target method will be called directly, thus avoiding performance hit caused by the reflective invocation.

Finally, Spring.NET AOP proxies will never return a raw reference to a target object. Whenever a target method returns a raw reference to a target object (i.e. "return this;"), AOP proxy will recognize what happened and will replace the return value with a reference to itself instead.

The current implementation of the AOP proxy generator uses object composition to delegate calls from the proxy to a target object, similar to how you would implement a classic Decorator pattern. This means that classes that need to be proxied have to implement one or more interfaces, which is in our opinion not only a less-intruding requirement than `ContextBoundObject` inheritance requirements, but also a good practice that should be followed anyway for the service classes that are most common targets for AOP proxies.

In a future release we will implement proxies using inheritance, which will allow you to proxy classes without interfaces as well and will remove some of the remaining raw reference issues that cannot be solved using composition-based proxies.

13.2. Pointcut API in Spring.NET

Let's look at how Spring.NET handles the crucial pointcut concept.

13.2.1. Concepts

Spring.NET's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The `Spring.Aop.IPointcut` interface is the central interface, used to target advices to particular types and methods. The complete interface is shown below:

```
public interface IPointcut
{
    ITypeFilter TypeFilter { get; }

    IMethodMatcher MethodMatcher { get; }
}
```

Splitting the `IPointcut` interface into two parts allows reuse of type and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ITypeFilter` interface is used to restrict the pointcut to a given set of target classes. If the `Matches()` method always returns true, all target types will be matched:

```
public interface ITypeFilter
{
    bool Matches(Type type);
}
```

The `IMethodMatcher` interface is normally more important. The complete interface is shown below:

```
public interface IMethodMatcher
{
    bool IsRuntime { get; }

    bool Matches(MethodInfo method, Type targetType);

    bool Matches(MethodInfo method, Type targetType, object[] args);
}
```

The `Matches(MethodInfo, Type)` method is used to test whether this pointcut will ever match a given method on a target type. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the `IsRuntime` property for the `IMethodMatcher` returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most `IMethodMatchers` are static, meaning that their `IsRuntime` property returns false. In this case, the 3-argument `Matches` method will never be invoked.

Whenever possible, try to make pointcuts static... this allows the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

13.2.2. Operations on pointcuts

Spring.NET supports operations on pointcuts: notably, *union* and *intersection*.

Union means the methods that either pointcut matches.

Intersection means the methods that both pointcuts match.

Union is usually more useful.

Pointcuts can be composed using the static methods in the *Spring.Aop.Support.Pointcuts* class, or using the *ComposablePointcut* class in the same namespace.

13.2.3. Convenience pointcut implementations

Spring.NET provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

13.2.3.1. Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient--and best--for most usages. It's possible for Spring.NET to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.NET.

13.2.3.1.1. Regular expression pointcuts

One obvious way to specify static pointcuts is using regular expressions. Several AOP frameworks besides Spring.NET make this possible. The *Spring.Aop.Support.SdkRegularExpressionMethodPointcut* class is a generic regular expression pointcut, that uses the regular expression classes from the .NET BCL.

Using this class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true (so the result is effectively the union of these pointcuts.). The matching is done against the full class name so you can use this pointcut if you would like to apply advice to all the classes in a particular namespace.

The usage is shown below:

```
<object id="settersAndAbsquatulatePointcut"
  type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut, Spring.Aop">
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</object>
```

As a convenience, Spring provides the *RegularExpressionMethodPointcutAdvisor* class that allows us to reference an *IAdvice* instance as well as defining the pointcut rules (remember that an *IAdvice* instance can be an interceptor, before advice, throws advice etc.) This simplifies wiring, as the one object serves as both pointcut and advisor, as shown below:

```
<object id="settersAndAbsquatulateAdvisor"
  type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor, Spring.Aop">
  <property name="advice">
    <ref local="objectNameOfAopAllianceInterceptor"/>
  </property>
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</object>
```

The *RegularExpressionMethodPointcutAdvisor* class can be used with any Advice type.

If you only have one pattern you can use the property name `pattern` and specify a single value instead of using the property name `patterns` and specifying a list.

You may also specify a `Regex` object from the `System.Text.RegularExpressions` namespace. The built in `RegexConverter` class will perform the conversion. See Section 6.4, “Built-in TypeConverters” for more information on Spring's built in type converters. The `Regex` object is created as any other object within the IoC container. Using an inner-object definition for the `Regex` object is a handy way to keep the definition close to the `PointcutAdvisor` declaration. Note that the class `SdkRegularExpressionMethodPointcut` has a `DefaultOptions` property to set the regular expression options if they are not explicitly specified in the constructor.

13.2.3.1.2. Attribute pointcuts

Pointcuts can be specified by matching an attribute type that is associated with a method. Advice associated with this pointcut can then read the metadata associated with the attribute to configure itself. The class `AttributeMatchMethodPointcut` provides this functionality. Sample usage that will match all methods that have the attribute `Spring.Attributes.CacheAttribute` is shown below.

```
<object id="cachePointcut" type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop">
  <property name="Attribute" value="Spring.Attributes.CacheAttribute, Spring.Core"/>
</object>
```

This can be used with a `DefaultPointcutAdvisor` as shown below

```
<object id="cacheAspect" type="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop">
  <property name="Pointcut">
    <object type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop">
      <property name="Attribute" value="Spring.Attributes.CacheAttribute, Spring.Core"/>
    </object>
  </property>
  <property name="Advice" ref="aspNetCacheAdvice"/>
</object>
```

where `aspNetCacheAdvice` is an implementation of an `IMethodInterceptor` that caches method return values. See the SDK docs for `Spring.Aop.Advice.CacheAdvice` for more information on this particular advice.

As a convenience the class `AttributeMatchMethodPointcutAdvisor` is provided to defining an attribute based Advisor as a somewhat shorter alternative to using the generic `DefaultPointcutAdvisor`. An example is shown below.

```
<object id="AspNetCacheAdvice" type="Spring.Aop.Support.AttributeMatchMethodPointcutAdvisor, Spring.Aop">
  <property name="advice">
    <object type="Aspect.AspNetCacheAdvice, Aspect"/>
  </property>
  <property name="attribute" value="Framework.AspNetCacheAttribute, Framework" />
</object>
```

13.2.3.2. Dynamic Pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

13.2.3.2.1. Control Flow Pointcuts

Spring.NET control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below another pointcut.). A control flow pointcut is dynamic because it is evaluated against the current call stack for each method invocation. For example, if method `ClassA.A()` calls `ClassB.B()` then the execution of `ClassB.B()` has occurred in `ClassA.A()`'s control flow. A control flow pointcut allows advice to be applied to the method `ClassA.A()` but only when called from `ClassB.B()`

and not when `ClassA.A()` is executed from another call stack. Control flow pointcuts are specified using the `Spring.Aop.Support.ControlFlowPointcut` class.



Note

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts.

When using control flow point cuts some attention should be paid to the fact that at runtime the JIT compiler can inline the methods, typically for increased performance, but with the consequence that the method no longer appears in the current call stack. This is because inlining takes the callee's IL code and inserts it into the caller's IL code effectively removing the method call. The information returned from `System.Diagnostics.StackTrace`, used in the implementation of `ControlFlowPointcut` is subject to these optimizations and therefore a control flow pointcut will not match if the method has been inlined.

Generally speaking, a method will be a candidate for inlining when its code is 'small', just a few lines of code (less than 32 bytes of IL). For some interesting reading on this process read David Notario's blog entries ([JIT Optimizations I](#) and [JIT Optimizations II](#)). Additionally, when an assembly is compiled with a Release configuration the assembly metadata instructs the CLR to enable JIT optimizations. When compiled with a Debug configuration the CLR will disable (some?) these optimizations. Empirically, method inlining is turned off in a Debug configuration.

The way to ensure that your control flow pointcut will not be overlooked because of method inlining is to apply the `System.Runtime.CompilerServices.MethodImplAttribute` attribute with the value `MethodImplOptions.NoInlining`. In this (somewhat artificial) simple example, if the code is compiled in release mode it will not match a control flow pointcut for the method "GetAge".

```
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

However, applying the attributes as shown below will prevent the method from being inlined even in a release build.

```
[MethodImpl(MethodImplOptions.NoInlining)]
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

13.2.4. Custom pointcuts

Because pointcuts in Spring.NET are .NET types, rather than language features (as in AspectJ) it is possible to declare custom pointcuts, whether static or dynamic. However, there is no support out of the box for the sophisticated pointcut expressions that can be coded in the AspectJ syntax. However, custom pointcuts in Spring.NET can be as arbitrarily complex as any object model.

Spring.NET provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are the most common and generally useful pointcut type, you'll probably subclass `StaticMethodMatcherPointcut`, as shown below. This requires you to implement just one abstract method (although it is possible to override other methods to customize behaviour):

```
public class TestStaticPointcut : StaticMethodMatcherPointcut {
```

```

public override bool Matches(MethodInfo method, Type targetType) {
    // return true if custom criteria match
}
}

```

13.3. Advice API in Spring.NET

Let's now look at how Spring.NET AOP handles advice.

13.3.1. Advice Lifecycle

Spring.NET advices can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

13.3.2. Advice types

Spring.NET provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

13.3.2.1. Interception Around Advice

The most fundamental advice type in Spring.NET is *interception around advice*.

Spring.NET is compliant with the AOP Alliance interface for around advice using method interception. Around advice is implemented using the following interface:

```

public interface IMethodInterceptor : IInterceptor
{
    object Invoke(IMethodInvocation invocation);
}

```

The `IMethodInvocation` argument to the `Invoke()` method exposes the method being invoked; the target joinpoint; the AOP proxy; and the arguments to the method. The `Invoke()` method should return the invocation's result: the return value of the joinpoint.

A simple `IMethodInterceptor` implementation looks as follows:

```

public class DebugInterceptor : IMethodInterceptor {

    public object Invoke(IMethodInvocation invocation) {
        Console.WriteLine("Before: invocation=[{0}]", invocation);
        object rval = invocation.Proceed();
        Console.WriteLine("Invocation returned");
        return rval;
    }
}

```

Note the call to the `IMethodInvocation`'s `Proceed()` method. This proceeds down the interceptor chain towards the joinpoint. Most interceptors will invoke this method, and return its return value. However, an `IMethodInterceptor`, like any around advice, can return a different value or throw an exception rather than invoke the `Proceed()` method. However, you don't want to do this without good reason!

13.3.2.2. Before advice

A simpler advice type is a **before advice**. This does not need an `IMethodInvocation` object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the `Proceed()` method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The `IMethodBeforeAdvice` interface is shown below.

```
public interface IMethodBeforeAdvice : IBeforeAdvice
{
    void Before(MethodInfo method, object[] args, object target);
}
```

Note the return type is `void`. Before advice can insert custom behaviour before the joinpoint executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring.NET, which counts all methods that return normally:

```
public class CountingBeforeAdvice : IMethodBeforeAdvice {

    private int count;

    public void Before(MethodInfo method, object[] args, object target) {
        ++count;
    }

    public int Count {
        get { return count; }
    }
}
```

Before advice can be used with any pointcut.

13.3.2.3. Throws advice

Throws advice is invoked after the return of the joinpoint if the joinpoint threw an exception. The `Spring.Aop.IThrowsAdvice` interface does not contain any methods: it is a tag interface identifying that the implementing advice object implements one or more typed throws advice methods. These throws advice methods must be of the form:

```
AfterThrowing([MethodInfo method, Object[] args, Object target], Exception subclass)
```

Throws-advice methods must be named 'AfterThrowing'. The return value will be ignored by the Spring.NET AOP framework, so it is typically `void`. With regard to the method arguments, only the last argument is required. Thus there are *exactly* one or four arguments, depending on whether the advice method is interested in the method, method arguments and the target object.

The following method snippets show examples of throws advice.

This advice will be invoked if a `RemotingException` is thrown (including subclasses):

```
public class RemoteThrowsAdvice : IThrowsAdvice {

    public void AfterThrowing(RemotingException ex) {
        // Do something with remoting exception
    }
}
```



```
    }
}
```

The following advice is invoked if a `SQLException` is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

```
public class SQLExceptionThrowsAdviceWithArguments : IThrowsAdvice {

    public void AfterThrowing(MethodInfo method, object[] args, object target, SQLException ex) {
        // Do something with all arguments
    }
}
```

The final example illustrates how these two methods could be used in a single class, which handles both `RemotingException` and `SQLException`. Any number of throws advice methods can be combined in a single class, as can be seen in the following example.

```
public class CombinedThrowsAdvice : IThrowsAdvice {

    public void AfterThrowing(RemotingException ex) {
        // Do something with remoting exception
    }

    public void AfterThrowing(MethodInfo method, object[] args, object target, SQLException ex) {
        // Do something with all arguments
    }
}
```

Finally, it is worth stating that throws advice is only applied to the actual exception being thrown. What does this mean? Well, it means that if you have defined some throws advice that handles `RemotingExceptions`, the applicable `AfterThrowing` method will **only** be invoked if the type of the thrown exception is `RemotingException`... if a `RemotingException` has been thrown and subsequently wrapped inside another exception before the exception bubbles up to the throws advice interceptor, then the throws advice that handles `RemotingExceptions` will **never** be called. Consider a business method that is advised by throws advice that handles `RemotingExceptions`; if during the course of a method invocation said business method throws a `RemoteException`... and subsequently wraps said `RemotingException` inside a business-specific `BadConnectionException` (see the code snippet below) before throwing the exception, then the throws advice will never be able to respond to the `RemotingException`... because all the throws advice sees is a `BadConnectionException`. The fact that the `RemotingException` is wrapped up inside the `BadConnectionException` is immaterial.

```
public void BusinessMethod()
{
    try
    {
        // do some business operation...
    }
    catch (RemotingException ex)
    {
        throw new BadConnectionException("Couldn't connect.", ex);
    }
}
```



Note

Please note that throws advice can be used with any pointcut.

13.3.2.4. After Returning advice

An after returning advice in Spring.NET must implement the `Spring.Aop.IAfterReturningAdvice` interface, shown below:


```
public interface IAfterReturningAdvice : IAdvice
{
    void AfterReturning(object returnValue, MethodBase method, object[] args, object target);
}
```

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

```
public class CountingAfterReturningAdvice : IAfterReturningAdvice {
    private int count;

    public void AfterReturning(object returnValue, MethodBase m, object[] args, object target) {
        ++count;
    }

    public int Count {
        get { return count; }
    }
}
```

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.



Note

Please note that after-returning advice can be used with any pointcut.

13.3.2.5. Advice Ordering

When multiple pieces of advice want to run on the same joinpoint the precedence is determined by having the advice implement the `IOrdered` interface or by specifying order information on an advisor.

13.3.2.6. Introduction advice

Spring.NET allows you to add new methods and properties to an advised class. This would typically be done when the functionality you wish to add is a crosscutting concern and want to introduce this functionality as a change to the static structure of the class hierarchy. For example, you may want to cast objects to the introduction interface in your code. Introductions are also a means to emulate multiple inheritance.

Introduction advice is defined by using a normal interface declaration that implements the tag interface `IAdvice`.



Note

The need for implementing this marker interface will likely be removed in future versions.

As an example, consider the interface `IAuditable` that describes the last modified time of an object.

```
public interface IAuditable : IAdvice
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

where

```
public interface IAdvice
{
}
```

Access to the advised object can be obtained by implementing the interface `ITargetAware`

```
public interface ITargetAware
{
    IAopProxy TargetProxy
    {
        set;
    }
}
```

with the `IAopProxy` reference providing a layer of indirection through which the advised object can be accessed.

```
public interface IAopProxy
{
    object GetProxy();
}
```

A simple class that demonstrates this functionality is shown below.

```
public interface IAuditable : IAdvice, ITargetAware
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

A class that implements this interface is shown below.

```
public class AuditableMixin : IAuditable
{
    private DateTime date;
    private IAopProxy targetProxy;

    public AuditableMixin()
    {
        date = new DateTime();
    }

    public DateTime LastModifiedDate
    {
        get { return date; }
        set { date = value; }
    }

    public IAopProxy TargetProxy
    {
        set { targetProxy = value; }
    }
}
```

Introduction advice is not associated with a pointcut, since it applies at the class and not the method level. As such, introductions use their own subclass of the interface `IAdvisor`, namely `IIntroductionAdvisor`, to specify the types that the introduction can be applied to.

```
public interface IIntroductionAdvisor : IAdvisor
{
    ITypeFilter TypeFilter { get; }

    Type[] Interfaces { get; }

    void ValidateInterfaces();
}
```

The `TypeFilter` property returns the filter that determines which target classes this introduction should apply to.

The `Interfaces` property returns the interfaces introduced by this advisor.

The `ValidateInterfaces()` method is used internally to see if the introduced interfaces can be implemented by the introduction advice.

Spring.NET provides a default implementation of this interface (the `DefaultIntroductionAdvisor` class) that should be sufficient for the majority of situations when you need to use introductions. The most simple implementation of an introduction advisor is a subclass that simply passes a new instance the base constructor. Passing a new instance is important since we want a new instance of the mixin classed used for each advised object.

```
public class AuditableAdvisor : DefaultIntroductionAdvisor
{
    public AuditableAdvisor() : base(new AuditableMixin())
    {
    }
}
```

Other constructors let you explicitly specify the interfaces of the class that will be introduced. See the SDK documentation for more details.

We can apply this advisor Programatically, using the `IAdvised.AddIntroduction()`, method, or (the recommended way) in XML configuration using the `IntroductionNames` property on `ProxyFactoryObject`, which will be discussed later.

Unlike the AOP implementation in the Spring Framework for Java, introduction advice in Spring.NET is not implemented as a specialized type of interception advice. The advantage of this approach is that introductions are not kept in the interceptor chain, which allows some significant performance optimizations. When a method is called that has no interceptors, a direct call is used instead of reflection regardless of whether the target method is on the target object itself or one of the introductions. This means that introduced methods perform the same as target object methods, which could be useful for adding introductions to fine grained objects. The disadvantage is that if the mixin functionality would benefit from having access to the calling stack, it is not available. Introductions with this functionality will be addressed in a future version of Spring.NET AOP.

13.4. Advisor API in Spring.NET

In Spring.NET, an advisor is a modularization of an aspect. Advisors typically incorporate both an advice and a pointcut.

Apart from the special case of introductions, any advisor can be used with any advice. The `Spring.Aop.Support.DefaultPointcutAdvisor` class is the most commonly used advisor implementation. For example, it can be used with a `IMethodInterceptor`, `IBeforeAdvice` or `IThrowsAdvice` and any pointcut definition.

Other convenience implementations provided are: `AttributeMatchMethodPointcutAdvisor` shown in usage previously in Section 13.2.3.1.2, “Attribute pointcuts” for use with attribute based pointcuts. `RegularExpressionMethodPointcutAdvisor` that will apply pointcuts based on the matching a regular expression to method names.

It is possible to mix advisor and advice types in Spring.NET in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring.NET will automatically create the necessary interceptor chain.

13.5. Using the ProxyFactoryObject to create AOP proxies

If you're using the Spring.NET IoC container for your business objects - generally a good idea - you will want to use one of Spring.NET's AOP-specific `IFactoryObject` implementations (remember that a factory object

introduces a layer of indirection, enabling it to create objects of a different type - Section 5.3.9, “Setting a reference using the members of other objects and classes.”).

The basic way to create an AOP proxy in Spring.NET is to use the `Spring.Aop.Framework.ProxyFactoryObject` class. This gives complete control over ordering and application of the pointcuts and advice that will apply to your business objects. However, there are simpler options that are preferable if you don't need such control.

13.5.1. Basics

The `ProxyFactoryObject`, like other Spring.NET `IFactoryObject` implementations, introduces a level of indirection. If you define a `ProxyFactoryObject` with name `foo`, what objects referencing `foo` see is not the `ProxyFactoryObject` instance itself, but an object created by the `ProxyFactoryObject`'s implementation of the `GetObject()` method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a `ProxyFactoryObject` or other IoC-aware classes that create AOP proxies, is that it means that advice and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

13.5.2. ProxyFactoryObject Properties

Like most `IFactoryObject` implementations provided with Spring.NET, the `ProxyFactoryObject` is itself a Spring.NET configurable object. Its properties are used to:

- Specify the target object that is to be proxied.
- Specify the advice that is to be applied to the proxy.

Some key properties are inherited from the `Spring.Aop.Framework.ProxyConfig` class: this class is the superclass for all AOP proxy factories in Spring.NET. Some of the key properties include:

- `ProxyTargetType`: a boolean value that should be set to true if the target class is to be proxied directly, as opposed to just proxying the interfaces exposed on the target class.
- `Optimize`: whether to apply aggressive optimization to created proxies. Don't use this setting unless you understand how the relevant AOP proxy handles optimization. The exact meaning of this flag will differ between proxy implementations and will generally result in a trade off between proxy creation time and runtime performance. Optimizations may be ignored by certain proxy implementations and may be disabled silently based on the value of other properties such as `ExposeProxy`.
- `IsFrozen`: whether advice changes should be disallowed once the proxy factory has been configured. The default is false.
- `ExposeProxy`: whether the current proxy should be exposed via the `AopContext` so that it can be accessed by the target. (It's available via the `IMethodInvocation` without the need for the `AopContext`.) If a target needs to obtain the proxy and `ExposeProxy` is true, the target can use the `AopContext.CurrentProxy` property.
- `AopProxyFactory`: the implementation of `IAopProxyFactory` to use when generating a proxy. Offers a way of customizing whether to use remoting proxies, IL generation or any other proxy strategy. The default implementation will use IL generation to create composition-based proxies.

Other properties specific to the `ProxyFactoryObject` class include:

- `ProxyInterfaces`: the array of `string` interface names we're proxying.

- **InterceptorNames:** string array of `IAdvisor`, interceptor or other advice names to apply. Ordering is significant... first come, first served that is. The first interceptor in the list will be the first to be able to intercept the invocation (assuming it concerns a regular `MethodInterceptor` or `BeforeAdvice`).

The names are object names in the current container, including object names from container hierarchies. You can't mention object references here since doing so would result in the `ProxyFactoryObject` ignoring the singleton setting of the advice.

- **IntroductionNames:** The names of objects in the container that will be used as introductions to the target object. If the object referred to by name does not implement the `IIntroductionAdvisor` it will be passed to the default constructor of `DefaultIntroductionAdvisor` and all of the objects interfaces will be added to the target object. Objects that implement the `IIntroductionAdvisor` interface will be used as is, giving you a finer level of control over what interfaces you may want to expose and the types for which they will be matched against.
- **IsSingleton:** whether or not the factory should return a single proxy object, no matter how often the `GetObject()` method is called. Several `IFactoryObject` implementations offer such a method. The default value is `true`. If you would like to be able to apply advice on a per-proxy object basis, use a `IsSingleton` value of `false` and a `IsFrozen` value of `false`. If you want to use stateful advice--for example, for stateful mixins--use prototype advices along with a `IsSingleton` value of `false`.

13.5.3. Proxying Interfaces

Let's look at a simple example of `ProxyFactoryObject` in action. This example involves:

- A target object that will be proxied. This is the "personTarget" object definition in the example below.
- An `IAdvisor` and an `IInterceptor` used to provide advice.
- An AOP proxy object definition specifying the target object (the personTarget object) and the interfaces to proxy, along with the advices to apply.

```
<object id="personTarget" type="MyCompany.MyApp.Person, MyCompany">
  <property name="name" value="Tony"/>
  <property name="age" value="51"/>
</object>

<object id="myCustomInterceptor" type="MyCompany.MyApp.MyCustomInterceptor, MyCompany">
  <property name="customProperty" value="configuration string"/>
</object>

<object id="debugInterceptor" type="Spring.Aop.Advice.DebugAdvice, Spring.Aop">
</object>

<object id="person" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

  <property name="proxyInterfaces" value="MyCompany.MyApp.IPerson"/>

  <property name="target" ref="personTarget"/>

  <property name="interceptorNames">
    <list>
      <value>debugInterceptor</value>
      <value>myCustomInterceptor</value>
    </list>
  </property>

</object>
```

Note that the `InterceptorNames` property takes a list of strings: the object names of the interceptor or advisors in the current context. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.

You might be wondering why the list doesn't hold object references. The reason for this is that if the `ProxyFactoryObject`'s `singleton` property is set to `false`, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it's necessary to be able to obtain an instance of the prototype from the context; holding a reference isn't sufficient.

The "person" object definition above can be used in place of an `IPerson` implementation, as follows:

```
IPerson person = (IPerson) factory.GetObject("person");
```

Other objects in the same IoC context can express a strongly typed dependency on it, as with an ordinary .NET object:

```
<object id="personUser" type="MyCompany.MyApp.PersonUser, MyCompany">
  <property name="person" ref="person"/>
</object>
```

The `PersonUser` class in this example would expose a property of type `IPerson`. As far as it's concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its type would be a proxy type. It would be possible to cast it to the `IAdvised` interface (discussed below).

It's possible to conceal the distinction between target and proxy using an anonymous *inline object*, as follows. (for more information on inline objects see Section 5.3.2.3, "Inner objects".) Only the `ProxyFactoryObject` definition is different; the advice is included only for completeness:

```
<object id="myCustomInterceptor" type="MyCompany.MyApp.MyCustomInterceptor, MyCompany">
  <property name="customProperty" value="configuration string"/>
</object>

<object id="debugInterceptor" type="Spring.Aop.Advice.DebugAdvice, Spring.Aop">
</object>

<object id="person" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

  <property name="proxyInterfaces" value="MyCompany.MyApp.IPerson"/>

  <property name="target">
    <!-- Instead of using a reference to target, just use an inline object -->
    <object type="MyCompany.MyApp.Person, MyCompany">
      <property name="name" value="Tony"/>
      <property name="age" value="51"/>
    </object>
  </property>

  <property name="interceptorNames">
    <list>
      <value>debugInterceptor</value>
      <value>myCustomInterceptor</value>
    </list>
  </property>

</object>
```

This has the advantage that there's only one object of type `Person`: useful if we want to prevent users of the application context obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There's also arguably an advantage in that the `ProxyFactoryObject` definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

13.5.1. Applying advice on a per-proxy basis.

Let's look at an example of configuring the proxy objects retrieved from `ProxyFactoryObject`.

```

<!-- create the object to reference -->
<object id="RealObjectTarget" type="MyRealObject" singleton="false"/>
<!-- create the proxied object for everyone to use-->
<object id="MyObject" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="proxyInterfaces" value="MyInterface" />
  <property name="isSingleton" value="false"/>
  <property name="targetName" value="RealObjectTarget" />
</object>

```

If you are using a prototype as the target you must set the `TargetName` property with the name/object id of your object and not use the property `Target` with a reference to that object. This will then allow a new proxy to be created around a new prototype target instance.

Consider the above Spring.Net object configuration. Notice that the `IsSingleton` property of the `ProxyFactoryObject` instance is set to false. This means that each proxy object will be unique. Thus, you can configure each proxy object with its' own individual advice(s) using the following syntax

```

// Will return un-advised instance of proxy object
MyInterface myProxyObject1 = (MyInterface)ctx.GetObject("MyObject");

// myProxyObject1 instance now has an advice attached to it.
IAdvised advised = (IAdvised)myProxyObject1;
advised.AddAdvice( new DebugAdvice() );

// Will return a new, un-advised instance of proxy object
MyInterface myProxyObject2 = (MyInterface)ctx.GetObject("MyObject");

```

13.5.4. Proxying Classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no `IPerson` interface, rather we needed to advise a class called `Person` that didn't implement any business interface. In this case the `ProxyFactoryObject` will proxy all public virtual methods and properties if no interfaces are explicitly specified or if no interfaces are found to be present on the target object. One can configure Spring.NET to force the use of class proxies, rather than interface proxies, by setting the `ProxyTargetType` property on the `ProxyFactoryObject` above to true.

Class proxying works by generating a subclass of the target class at runtime. Spring.NET configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the *Decorator* pattern, weaving in the advice.

Class proxying should generally be transparent to users. However, there is an important issue to consider: *Non-virtual methods can't be advised, as they can't be overridden*. This may be a limiting factor when using existing code as it has been common practice not to declare methods as virtual by default.

13.5.5. Concise proxy definitions

Especially when defining transactional proxies, if you do not make use of the transaction namespace, you may end up with many similar proxy definitions. The use of parent and child object definitions, along with inner object definitions, can result in much cleaner and more concise proxy definitions.

First a parent, template, object definition is created for the proxy:

```

<object id="txProxyTemplate" abstract="true"
  type="Spring.Transaction.Interceptor.TransactionProxyFactoryObject, Spring.Data">

```

```
<property name="PlatformTransactionManager" ref="adoTransactionManager"/>
<property name="TransactionAttributes">
  <name-values>
    <add key="*" value="PROPAGATION_REQUIRED"/>
  </name-values>
</property>
</object>
```

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child object definition, which wraps the target of the proxy as an inner object definition, since the target will never be used on its own anyway.

```
<object name="testObjectManager" parent="txProxyTemplate">
  <property name="Target">
    <object type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
      <property name="TestObjectDao" ref="testObjectDao"/>
    </object>
  </property>
</object>
```

It is of course possible to override properties from the parent template, such as in this case, the transaction propagation settings:

```
<object name="testObjectManager" parent="txProxyTemplate">
  <property name="Target">
    <object type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
      <property name="TestObjectDao" ref="testObjectDao"/>
    </object>
  </property>
  <property name="TransactionAttributes">
    <name-values>
      <add key="Save*" value="PROPAGATION_REQUIRED"/>
      <add key="Delete*" value="PROPAGATION_REQUIRED"/>
      <add key="Find*" value="PROPAGATION_REQUIRED,readonly"/>
    </name-values>
  </property>
</object>
```

Note that in the example above, we have explicitly marked the parent object definition as abstract by using the abstract attribute, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple object factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton object) that if you have a (parent) object definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the abstract attribute to true, otherwise the application context will actually try to pre-instantiate it.

13.6. Proxying mechanisms

Spring creates AOP proxies built at runtime through the use of the TypeBuilder API.

Two types of proxies can be created, composition based or inheritance based. If the target object implements at least one interface then a composition based proxy will be created, otherwise an inheritance based proxy will be created.

The composition based proxy is implemented by creating a type that implements all the interfaces specified on the target object. The actual class name of this dynamic type is 'GUID' like. A private field holds the target object and the dynamic type implementation will first execute any advice before or after making the target object method call on the target object.

The inheritance based mechanism creates a dynamic type where that inherits from the target type. This lets you downcast to the target type if needed. Please note that in both cases a target method implementation that calls

other methods on the target object will not be advised. To force inheritance based proxies you should either set the `ProxyTargetType` to true property of a `ProxyFactory` or set the XML namespace element `proxy-target-type = true` when using an AOP schema based configuration.



Note

An important alternative approach to inheritance based proxies is discussed in the next section.

In .NET 2.0 you can define the assembly level attribute, `InternalsVisibleTo`, to allow access of internal interfaces/classes to specified 'friend' assemblies. If you need to create an AOP proxy on an internal class/interface add the following code, `[assembly: InternalsVisibleTo("Spring.Proxy")]` and `[assembly: InternalsVisibleTo("Spring.DynamicReflection")]` to your `AssemblyInfo` file.

13.6.1. InheritanceBasedAopConfigurer

There is an important limitation in the inheritance based proxy as described above, all methods that manipulate the state of the object should be declared as virtual. Otherwise some method invocations get directed to the private 'target' field member and others to the base class. Winform object are an example of case where this approach does not apply. To address this limitation, a new post-processing mechanism was introduced in version 1.2 that creates a proxy type without the private 'target' field. Interception advice is added directly in the method body before invoking the base class method.

To use this new inheritance based proxy described in the note above, declare an instance of the `InheritanceBasedAopConfigurer`, and `IObjectFactoryPostProcessor`, in your configuration file. Here is an example.

```
<object type="Spring.Aop.Framework.AutoProxy.InheritanceBasedAopConfigurer, Spring.Aop">
  <property name="ObjectNames">
    <list>
      <value>Form*</value>
      <value>Control*</value>
    </list>
  </property>
  <property name="InterceptorNames">
    <list>
      <value>debugInterceptor</value>
    </list>
  </property>
</object>

<object id="debugInterceptor" type="AopPlay.DebugInterceptor, AopPlay"/>
```

This configuration style is similar to the autoproxy by name approach described here and is particularly appropriate when you want to apply advice to WinForm classes.

13.7. Creating AOP Proxies Programmatically with the ProxyFactory

It's easy to create AOP proxies Programmatically using Spring.NET. This enables you to use Spring.NET AOP without dependency on Spring.NET IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.AddAdvice(myMethodInterceptor);
```

```
factory.AddAdvisor(myAdvisor);
IBusinessInterface tb = (IBusinessInterface) factory.GetProxy();
```

The first step is to construct an object of type `Spring.Aop.Framework.ProxyFactory`. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add interceptors or advisors, and manipulate them for the life of the `ProxyFactory`.

There are also convenience methods on `ProxyFactory` (inherited from `AdvisedSupport`) allowing you to add other advice types such as before and throws advice. `AdvisedSupport` is the superclass of both `ProxyFactory` and `ProxyFactoryObject`.



Note

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from .NET code with AOP, as in general.

13.8. Manipulating Advised Objects

However you create AOP proxies, you can manipulate them using the `Spring.Aop.Framework.IAdvised` interface. Any AOP proxy can be cast to this interface, whatever other interfaces it implements. This interface includes the following methods and properties:

```
public interface IAdvised
{
    IAdvisor[] Advisors { get; }

    IIntroductionAdvisor[] Introductions { get; }

    void AddInterceptor(IInterceptor interceptor);

    void AddInterceptor(int pos, IInterceptor interceptor);

    void AddAdvisor(IAdvisor advisor);

    void AddAdvisor(int pos, IAdvisor advisor);

    void AddIntroduction(IIntroductionAdvisor advisor);

    void AddIntroduction(int pos, IIntroductionAdvisor advisor);

    int IndexOf(IAdvisor advisor);

    int IndexOf(IIntroductionAdvisor advisor);

    bool RemoveAdvisor(IAdvisor advisor);

    void RemoveAdvisor(int index);

    bool RemoveInterceptor(IInterceptor interceptor);

    bool RemoveIntroduction(IIntroductionAdvisor advisor);

    void RemoveIntroduction(int index);

    void ReplaceIntroduction(int index, IIntroductionAdvisor advisor);

    bool ReplaceAdvisor(IAdvisor a, IAdvisor b);
}
```

The `Advisors` property will return an `IAdvisor` for every advisor, interceptor or other advice type that has been added to the factory. If you added an `IAdvisor`, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring.NET will have wrapped this in an advisor with a `IPointcut` that always returns `true`. Thus if you added an `IMethodInterceptor`, the advisor returned for this

index will be a `DefaultPointcutAdvisor` returning your `IMethodInterceptor` and an `IPointcut` that matches all types and methods.

The `AddAdvisor()` methods can be used to add any `IAdvisor`. Usually this will be the generic `DefaultPointcutAdvisor`, which can be used with any advice or pointcut (but not for introduction).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

It's questionable whether it's advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a `Frozen` flag, in which case the `IAdvised.IsFrozen` property will return `true`, and any attempts to modify advice through addition or removal will result in an `AopConfigException`. The ability to freeze the state of an advised object is useful in some cases: For example, to prevent calling code removing a security interceptor.

13.9. Using the "autoproxy" facility

So far we've considered explicit creation of AOP proxies using a `ProxyFactoryObject` or similar factory objects. For applications that would like create many AOP proxies, say across all the classes in a service layer, this approach can lead to a lengthy configuration file. To simplify the creation of many AOP proxies Spring provides "autoproxy" capabilities that will automatically proxy object definitions based on higher level criteria that will group together multiple objects as candidates to be proxied.

This functionality is built on Spring "object post-processor" infrastructure, which enables modification of any object definition as the container loads. Refer to Section 5.9.1, "Customizing objects with `IObjectPostProcessors`" for general information on object post-processors.

In this model, you set up some special object definitions in your XML object definition file configuring the auto proxy infrastructure. This allows you just to declare the targets eligible for autoproxying: you don't need to use `ProxyFactoryObject`.

- Using an autoproxy creator that refers to specific objects in the current context.
- A special case of autoproxy creation that deserves to be considered separately; autoproxy creation driven by source-level attributes.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an unadvised object. Calling `GetObject("MyBusinessObject1")` on an `ApplicationContext` will return an AOP proxy, not the target business object. The "inline object" idiom shown earlier in Section 13.5.3, "Proxying Interfaces" also offers this benefit.)

13.9.1. Autoproxy object definitions

The namespace `Spring.Aop.Framework.AutoProxy` provides generic autoproxy infrastructure, should you choose to write your own autoproxy implementations, as well as several out-of-the-box implementations. Two

implementations are provided, `ObjectNameAutoProxyCreator` and `DefaultAdvisorAutoProxyCreator`. These are discussed in the following sections.

13.9.1.1. ObjectNameAutoProxyCreator

The `ObjectNameAutoProxyCreator` automatically creates AOP proxies for object with names matching literal values or wildcards. The pattern matching expressions supported are of the form `"*name"`, `"name*"`, and `"*name*"` and exact name matching, i.e. `"name"`. The following simple classes are used to demonstrate this autoproxy functionality.

```
public enum Language
{
    English = 1,
    Portuguese = 2,
    Italian = 3
}

public interface IHelloWorldSpeaker
{
    void SayHello();
}

public class HelloWorldSpeaker : IHelloWorldSpeaker
{
    private Language language;

    public Language Language
    {
        set { language = value; }
        get { return language; }
    }

    public void SayHello()
    {
        switch (language)
        {
            case Language.English:
                Console.WriteLine("Hello World!");
                break;
            case Language.Portuguese:
                Console.WriteLine("Oi Mundo!");
                break;
            case Language.Italian:
                Console.WriteLine("Ciao Mondo!");
                break;
        }
    }
}

public class DebugInterceptor : IMethodInterceptor
{
    public object Invoke(IMethodInvocation invocation)
    {
        Console.WriteLine("Before: " + invocation.Method.ToString());
        object rval = invocation.Proceed();
        Console.WriteLine("After: " + invocation.Method.ToString());
        return rval;
    }
}
```

The following XML is used to automatically create an AOP proxy and apply a Debug interceptor to object definitions whose names match `"English*"` and `"PortugueseSpeaker"`.

```
<object id="ProxyCreator" type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator, Spring.Aop">
  <property name="ObjectNames">
    <list>
```

```

        <value>English*</value>
        <value>PortugeseSpeaker</value>
    </list>
</property>
<property name="InterceptorNames">
    <list>
        <value>debugInterceptor</value>
    </list>
</property>
</object>

<object id="debugInterceptor" type="AopPlay.DebugInterceptor, AopPlay"/>

<object id="EnglishSpeakerOne" type="AopPlay.HelloWorldSpeaker, AopPlay">
    <property name="Language" value="English"/>
</object>

<object id="EnglishSpeakerTwo" type="AopPlay.HelloWorldSpeaker, AopPlay">
    <property name="Language" value="English"/>
</object>

<object id="PortugeseSpeaker" type="AopPlay.HelloWorldSpeaker, AopPlay">
    <property name="Language" value="Portuguese"/>
</object>

<object id="ItalianSpeakerOne" type="AopPlay.HelloWorldSpeaker, AopPlay">
    <property name="Language" value="Italian"/>
</object>

```

As with `ProxyFactoryObject`, there is an `InterceptorNames` property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

The same advice will be applied to all matching objects. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different objects.

Running the following simple program demonstrates the application of the AOP interceptor.

```

IApplicationContext ctx = ContextRegistry.GetContext();
IDictionary speakerDictionary = ctx.GetObjectsOfType(typeof(IHelloWorldSpeaker));
foreach (DictionaryEntry entry in speakerDictionary)
{
    string name = (string)entry.Key;
    IHelloWorldSpeaker worldSpeaker = (IHelloWorldSpeaker)entry.Value;
    Console.WriteLine(name + " says; ");
    worldSpeaker.SayHello();
}

```

The output is shown below

```

ItalianSpeakerOne says; Ciao Mondo!
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After: Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After: Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After: Void SayHello()

```

13.9.1.2. DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is `DefaultAdvisorAutoProxyCreator`. This will automatically apply eligible advisors in the current application context, without the need to include specific object names in the autoproxy advisor's object definition. It offers the same merit of consistent configuration and avoidance of duplication as `ObjectNameAutoProxyCreator`.

Using this mechanism involves:

- Specifying a `DefaultAdvisorAutoProxyCreator` object definition
- Specifying any number of Advisors in the same or related contexts. Note that these *must* be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate object definitions.

The `DefaultAdvisorAutoProxyCreator` will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each object defined in the application context.

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied.

The `DefaultAdvisorAutoProxyCreator` is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily--for example, tracing or performance monitoring aspects--with minimal change to configuration.

The following example demonstrates the use of `DefaultAdvisorAutoProxyCreator`. Expanding on the previous example code used to demonstrate `ObjectNameAutoProxyCreator` we will add a new class, `SpeakerDao`, that acts as a Data Access Object to find and store `ISpeakerDao` objects.

```
public interface ISpeakerDao
{
    IList FindAll();

    IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker);
}

public class SpeakerDao : ISpeakerDao
{
    public System.Collections.IList FindAll()
    {
        Console.WriteLine("Finding speakers...");
        // just a demo...fake the retrieval.
        Thread.Sleep(10000);
        HelloWorldSpeaker speaker = new HelloWorldSpeaker();
        speaker.Language = Language.Portuguese;

        IList list = new ArrayList();
        list.Add(speaker);
        return list;
    }

    public IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker)
    {
        Console.WriteLine("Saving speaker...");
        // just a demo...not really saving...
        return speaker;
    }
}
```

The XML configuration specifies two Advisors, that is, the combination of advice (the behavior to add) and a pointcut (where the behavior should be applied). A `RegularExpressionMethodPointcutAdvisor` is used as a convenience to specify the pointcut as a regular expression that matches methods names. Other pointcuts of your own creation could be used, in which case a `DefaultPointcutAdvisor` would be used to define the Advisor. The object definitions for these advisors, advice, and `SpeakerDao` object are shown below

```
<object id="SpeechAdvisor" type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor, Spring.Aop">
```

```

    <property name="advice" ref="debugInterceptor"/>
    <property name="patterns">
        <list>
            <value>.*Say.*</value>
        </list>
    </property>

</object>

<object id="AdoAdvisor" type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor, Spring.Aop">

    <property name="advice" ref="timingInterceptor"/>
    <property name="patterns">
        <list>
            <value>.*Find.*</value>
        </list>
    </property>

</object>

// Advice
<object id="debugInterceptor" type="AopPlay.DebugInterceptor, AopPlay"/>

<object id="timingInterceptor" type="AopPlay.TimingInterceptor, AopPlay"/>

// Speaker DAO Object - has 'FindAll' Method.
<object id="speakerDao" type="AopPlay.SpeakerDao, AopPlay"/>

// HelloWorldSpeaker objects as previously listed.

```

Adding an instance of DefaultAdvisorAutoProxyCreator to the configuration file

```

<object id="ProxyCreator" type="Spring.Aop.Framework.AutoProxy.DefaultAdvisorAutoProxyCreator, Spring.Aop"/>

```

will apply the debug interceptor on all objects in the context that have a method that contains the text "Say" and apply the timing interceptor on objects in the context that have a method that contains the text "Find". Running the following code demonstrates this behavior. Note that the "Save" method of SpeakerDao does not have any advice applied to it.

```

IApplicationContext ctx = ContextRegistry.GetContext();
IDictionary speakerDictionary = ctx.GetObjectsOfType(typeof(IHelloWorldSpeaker));
foreach (DictionaryEntry entry in speakerDictionary)
{
    string name = (string)entry.Key;
    IHelloWorldSpeaker worldSpeaker = (IHelloWorldSpeaker)entry.Value;
    Console.WriteLine(name + " says; Before: Void SayHello()");
    worldSpeaker.SayHello();
}
ISpeakerDao dao = (ISpeakerDao)ctx.GetObject("speakerDao");
IList speakerList = dao.FindAll();
IHelloWorldSpeaker speaker = dao.Save(new HelloWorldSpeaker());

```

This produces the following output

```

ItalianSpeakerOne says; Before: Void SayHello()
Ciao Mondo!
After: Void SayHello()
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After: Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After: Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After: Void SayHello()
Finding speakers...
Elapsed time = 00:00:10.0154745

```

```
Saving speaker...
```

The `DefaultAdvisorAutoProxyCreator` offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, `AdvisorAutoProxyCreators` in the same factory) and ordering. Advisors can implement the `Spring.Core.IOrdered` interface to ensure correct ordering if this is an issue. The default is unordered.

13.9.1.3. PointcutFilteringAutoProxyCreator

An `AutoProxyCreator` that identified objects to proxy by matching a specified `IPointcut`.

13.9.1.4. TypeNameAutoProxyCreator

An `AutoProxyCreator` that identifies objects to proxy by matching their `Type.FullName` against a list of patterns.

13.9.1.5. AttributeAutoProxyCreator

An `AutoProxyCreator`, that identifies objects to be proxied by checking any `System.Attribute` defined on a given type and that types interfaces.

13.9.1.6. AbstractFilteringAutoProxyCreator

The base class for `AutoProxyCreator` implementations that mark objects eligible for proxying based on arbitrary criteria.

13.9.1.7. AbstractAutoProxyCreator

This is the superclass of `DefaultAdvisorAutoProxyCreator`. You can create your own autoproxy creators by subclassing this class, in the unlikely event that advisor definitions offer insufficient customization to the behavior of the framework `DefaultAdvisorAutoProxyCreator`.

13.9.2. Using attribute-driven auto-proxying

A particularly important type of autoproxyming is driven by attributes. The programming model is similar to using Enterprise Services with `ServicedComponents`.

In this case, you use the `DefaultAdvisorAutoProxyCreator`, in combination with Advisors that understand attributes. The Advisor pointcut is identified by the presence of .NET attribute in the source code and it is configured via the data and/or methods of the attribute. This is a powerful alternative to identifying the advisor pointcut and advice configuration through traditional property configuration, either programmatic or through XML based configuration.

Several of the aspect provided with Spring use attribute driven autoproxyming. The most prominent example is Transaction support.

13.10. Using AOP Namespace

The AOP namespace allows you to define an advisor, i.e pointcut + 1 piece of advice, in a more declarative manner. Under the covers the `DefaultAdvisorAutoProxyCreator` is being used. Here is an example,

```
<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.net/aop">

  <aop:config>
```



```

<aop:advisor id="getDescriptionAdvisor" pointcut-ref="getDescriptionCalls" advice-ref="getDescriptionCounter"/>

</aop:config>

<object id="getDescriptionCalls"
  type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut, Spring.Aop">
  <property name="patterns">
    <list>
      <value>.*GetDescription.*</value>
    </list>
  </property>
</object>

<object id="getDescriptionCounter" type="Spring.Aop.Framework.CountingBeforeAdvice, Spring.Aop.Tests"/>

<object name="testObject" type="Spring.Objects.TestObject, Spring.Core.Tests"/>

</objects>

```

In this example, the TestObject, which implements the interface ITestObject, is having AOP advice applied to it. The method GetDescription() is specified as a regular expression pointcut. The aop:config tag and subsequent child tag, aop:advisor, brings together the pointcut with the advice.

In order to have Spring.NET recognise the aop namespace, you need to declare the namespace parser in the main Spring.NET configuration section. For convenience this is shown below. Please refer to the section titled context configuration for more extensive information..

```

<configuration>

  <configSections>
    <sectionGroup name="spring">

      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />

      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>

    </sectionGroup>
  </configSections>

  <spring>

    <parsers>
      <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
    </parsers>

    <context>
      <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
      ...
    </objects>

  </spring>

</configuration>

```

13.11. Using TargetSources

Spring.NET offers the concept of a *TargetSource*, expressed in the Spring.Aop.ITargetSource interface. This interface is responsible for returning the "target object" implementing the joinpoint. The TargetSource implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring.NET AOP don't normally need to work directly with `TargetSources`, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling `TargetSource` can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a `TargetSource`, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let's look at the standard target sources provided with Spring.NET, and how you can use them.

When using a custom target source, your target will usually need to be a prototype rather than a singleton object definition. This allows Spring.NET to create a new target instance when required.

13.11.1. Hot swappable target sources

The `org.Spring.NETframework.aop.target.HotSwappableTargetSource` exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source's target takes effect immediately. The `HotSwappableTargetSource` is thread safe.

You can change the target via the `swap()` method on `HotSwappableTargetSource` as follows:

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) objectFactory.GetObject("swapper");
object oldTarget = swapper.swap(newTarget);
```

The XML definitions required look as follows:

```
<object id="initialTarget" type="MyCompany.OldTarget, MyCompany">
</object>

<object id="swapper"
    type="Spring.Aop.Target.HotSwappableTargetSource, Spring.Aop">
    <constructor-arg><ref local="initialTarget" /></constructor-arg>
</object>

<object id="swappable"
    type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop"
>
    <property name="targetSource">
        <ref local="swapper" />
    </property>
</object>
```

The above `swap()` call changes the target of the swappable object. Clients who hold a reference to that object will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn't add any advice--and it's not necessary to add advice to use a `TargetSource`--of course any `TargetSource` can be used in conjunction with arbitrary advice.

13.11.2. Pooling target sources

Using a pooling target source provides a programming model in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring.NET pooling and pooling in .NET Enterprise Services pooling is that Spring.NET pooling can be applied to any PONO. (Plain old .NET object). As with Spring.NET in general, this service can be applied in a non-invasive way.

Spring.NET provides out-of-the-box support using a pooling implementation based on Jakarta Commons Pool 1.1, which provides a fairly efficient pooling implementation. It's also possible to subclass `Spring.Aop.Target.AbstractPoolingTargetSource` to support any other pooling API.

Sample configuration is shown below:

```
<object id="businessObjectTarget" type="MyCompany.MyBusinessObject, MyCompany" singleton="false">
  ... properties omitted
</object>

<object id="poolTargetSource" type="Spring.Aop.Target.SimplePoolTargetSource, Spring.Aop">
  <property name="targetObjectName" value="businessObjectTarget"/>
  <property name="maxSize" value="25"/>
</object>

<object id="businessObject" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
</object>
```

Note that the target object--"businessObjectTarget" in the example--*must* be a prototype. This allows the PoolingTargetSource implementation to create new instances of the target to grow the pool as necessary. See the SDK documentation for AbstractPoolingTargetSource and the concrete subclass you wish to use for information about its properties: maxSize is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn't necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don't set the interceptorNames property at all.

It's possible to configure Spring.NET so as to be able to cast any pooled object to the Spring.Aop.Target.PoolingConfig interface, which exposes information about the configuration and current size of the pool through an introduction. You'll need to define an advisor like this:

```
<object id="poolConfigAdvisor"
  type="Spring.Object.Factory.Config.MethodInvokingFactoryObject, Spring.Aop">
  <property name="target" ref="poolTargetSource" />
  <property name="targetMethod" value="getPoolingConfigMixin" />
</object>
```

This advisor is obtained by calling a convenience method on the AbstractPoolingTargetSource class, hence the use of MethodInvokingFactoryObject. This advisor's name ('poolConfigAdvisor' here) must be in the list of interceptor names in the ProxyFactoryObject exposing the pooled object.

The cast will look as follows:

```
PoolingConfig conf = (PoolingConfig) objectFactory.GetObject("businessObject");
Console.WriteLine("Max pool size is " + conf.getMaxSize());
```

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally threadsafe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using autoproxying. It's possible to set the TargetSources used by any autoproxy creator.

13.11.3. Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object may not be high, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn't use this approach without very good reason.

To do this, you could modify the poolTargetSource definition shown above as follows. (the name of the definition has also been changed, for clarity.)

```
<object id="prototypeTargetSource"
      type="Spring.Aop.Target.PrototypeTargetSource, Spring.Aop">
  <property name="targetObjectName" value="businessObject" />
</object>
```

There is only one property: the name of the target object. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target object must be a prototype object definition, the singleton property of the target should be set to false.

13.11.4. ThreadLocal target sources

ThreadLocal target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a ThreadLocal provides a facility to transparently store resource alongside a thread. Setting up a ThreadLocalTargetSource is pretty much the same as was explained for the other types of target source:

```
<object id="threadlocalTargetSource"
      type="Spring.Aop.Target.ThreadLocalTargetSource, Spring.Aop">
  <property name="targetObjectName" value="businessObject" />
</object>
```

13.12. Defining new Advice types

Spring.NET AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to interception around, before, throws, and after returning advice, which are supported out of the box.

The `Spring.Aop.Framework.Adapter` package is an SPI (Service Provider Interface) package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement the `AopAlliance.Aop.IAdvice` tag interface.

Please refer to the `Spring.Aop.Framework.Adapter` namespace documentation for further information.

13.13. Further reading and resources

The Spring.NET team recommends the excellent *AspectJ in Action* by Ramnivas Laddad (Manning, 2003) for an introduction to AOP.

If you are interested in more advanced capabilities of Spring.NET AOP, take a look at the test suite as it illustrates advanced features not discussed in this document.

Chapter 14. Aspect Library

14.1. Introduction

Spring provides several aspects in the distribution. The most popular of which is transactional advice, located in the `Spring.Data` module. However, the aspects that are documented in this section are those contained within the `Spring.Aop` module itself. The aspects in within `Spring.Aop.dll` are Caching, Exception Handling, Logging, Retry, and Parameter Validation. Other traditional advice types such as validation, security, and thread management, will be included in a future release.

14.2. Caching

Caching the return value of a method or the value of a method parameter is a common approach to increase application performance. Application performance is increased with effective use of caching since layers in the application that are closer to the user can return information within their own layer as compared to making more expensive calls to retrieve that information from a lower, and more slow, layer such as a database or a web service. Caching also can help in terms of application scalability, which is generally the more important concern.

The caching support in Spring.NET consists of base cache interfaces that can be used to specify a specific storage implementation of the cache and also an aspect that determines where to apply the caching functionality and its configuration.

The base cache interface that any cache implementation should implement is `Spring.Caching.ICache` located in `Spring.Core`. Two implementations are provided, `Spring.Caching.AspNetCache` located in `Spring.Web` which stores cache entries within an ASP.NET cache and a simple implementation, `Spring.Caching.NonExpiringCache` that stores cache entries in memory and never expires these entries. Custom implementations based on 3rd party implementations, such as Oracle Coherence, or memcached, can be used by implementing the `ICache` interface.

The cache aspect is `Spring.Aspects.Cache.CacheAspect` located in `Spring.Aop`. It consists of three pieces of functionality, the ability to cache return values, method parameters, and explicit eviction of an item from the cache. The aspect currently relies on using attributes to specify the pointcut as well as the behavior, much like the transactional aspect. Future versions will allow for external configuration of the behavior so you can apply caching to a code base without needing to use attributes in the code.

The following attributes are available

- `CacheResult` - used to cache the return value
- `CacheResultItems` - used when returning a collection as a return value
- `CacheParameter` - used to cache a method parameter
- `InvalidateCache` - used to indicate one or more cache items should be invalidated.

Each `CacheResult`, `CacheResultItems`, and `CacheParameter` attributes define the following properties.

- `CacheName` - the name of the cache implementation to use
- `Key` - a string representing a Spring Expression Language (SpEL) expression used as the key in the cache.

- `Condition` - a SpEL expression that should be evaluated in order to determine whether the item should be cached.
- `TimeToLive` - The amount of time an object should remain in the cache (in seconds).

The `InvalidateCache` attribute has properties for the `CacheName`, the `Key` as well as the `Condition`, with the same meanings as listed previously.

Each `ICache` implementation will have properties that are specific to a caching technology. In the case of `AspNetCache`, the two important properties to configure are:

- `SlidingExpiration` - If this property value is set to true, every time the marked object is accessed it's `TimeToLive` value is reset to its original value
- `Priority` - the cache item priority controlling how likely an object is to be removed from an associated cache when the cache is being purged.
- `TimeToLive` - The amount of time an object should remain in the cache (in seconds).

The values of the `Priority` enumeration are

- `Low` - low likelihood of deletion when cache is purged.
- `Normal` - default priority for deletion when cache is purged.
- `High` - high likelihood of deletion when cache is purged.
- `NotRemovable` - cache item not deleted when cache is purged.

An important element of the applying these attributes is the use of the expression language that allows for calling context information to drive the caching actions. Here is an example taken from the Spring Air sample application of the `AirportDao` implementation that implements an interface with the method `GetAirport(long id)`.

```
[CacheResult("AspNetCache", "'Airport.Id=' + #id", TimeToLive = "0:1:0")]
public Airport GetAirport(long id)
{
    // implementation not shown...
}
```

The first parameter is the cache name. The second string parameter is the cache key and is a string expression that incorporates the argument passed into the method, the `id`. The method parameter names are exposed as variables to the key expression. If you do not specify a key, then all the parameter values will be used to cache the returned value. The expression may also call out to other objects in the Spring container allowing for a more complex key algorithm to be encapsulated. The end result is that the `Airport` object is cached by `id` for 60 seconds in a cache named `AspNetCache`. The `TimeToLive` property could also have been specified on the configuration of the `AspNetCache` object.

The configuration to enable the caching aspect is shown below

```
<object id="CacheAspect" type="Spring.Aspects.Cache.CacheAspect, Spring.Aop"/>

<object id="AspNetCache" type="Spring.Caching.AspNetCache, Spring.Web">
  <property name="SlidingExpiration" value="true"/>
  <property name="Priority" value="Low"/>
  <property name="TimeToLive" value="00:02:00"/>
</object>
```

```

<!-- Apply aspects to DAOs -->
<object type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator, Spring.Aop">
  <property name="ObjectNames">
    <list>
      <value>*Dao</value>
    </list>
  </property>
  <property name="InterceptorNames">
    <list>
      <value>CacheAspect</value>
    </list>
  </property>
</object>

```

in this example an `ObjectNameAutoProxyCreator` was used to apply the cache aspect to objects that have `Dao` in their name. The `AspNetCache` setting for `TimeToLive` will override the `TimeToLive` value set at the method level via the attribute.

14.3. Exception Handling

In some cases existing code can be easily adopted to a simple error handling strategy that can perform one of the following actions

- translations - either wrap the thrown exception inside a new one or replace it with a new exception type (no inner exception is set).
- return value - the exception is ignored and a return value for the method is provided instead
- swallow - the exception is ignored.
- execute - Execute an arbitrary Spring Expression Language (SpEL expression)

The applicability of general exception handling advice depends greatly on how tangled the code is regarding access to local variables that may form part of the exception. Once you get familiar with the feature set of Spring declarative exception handling advice you should evaluate where it may be effectively applied in your code base. It is worth noting that you can still chain together multiple pieces of exception handling advice allowing you to mix the declarative approach shown in this section with the traditional inheritance based approach, i.e. implementing `IThrowsAdvice` or `IMethodInterceptor`.

Declarative exception handling is expressed in the form of a mini-language relevant to the domain at hand, exception handling. This could be referred to as a Domain Specific Language (DSL). Here is a simple example, which should hopefully be self explanatory.

```

<object name="exceptionHandlingAdvice" type="Spring.Aspects.Exceptions.ExceptionHandlerAdvice, Spring.Aop">
  <property name="exceptionHandlers">
    <list>
      <value>on exception name ArithmeticException wrap System.InvalidOperationException</value>
    </list>
  </property>
</object>

```

What this is instructing the advice to do is the following bit of code when an `ArithmeticException` is thrown, throw new `System.InvalidOperationException("Wrapped ArithmeticException", e)`, where `e` is the original `ArithmeticException`. The default message, "Wrapped ArithmeticException" is automatically appended. You may however specify the message used in the newly thrown exception as shown below

```
on exception name ArithmeticException wrap System.InvalidOperationException 'My Message'
```

Similarly, if you would rather replace the exception, that is do not nest one inside the other, you can use the following syntax

```
on exception name ArithmeticException replace System.InvalidOperationException

or

on exception name ArithmeticException replace System.InvalidOperationException 'My Message'
```

Both wrap and replace are special cases of the more general translate action. An example of a translate expression is shown below

```
on exception name ArithmeticException translate new System.InvalidOperationException('My Message, Method Name ' + #method.Name
```

What we see here after the translate keyword is text that will be passed into Spring's expression language (SpEL) for evaluation. Refer to the chapter on the expression language for more details. One important feature of the expression evaluation is the availability of variables relating to the calling context when the exception was thrown. These are

- method - the MethodInfo object corresponding to the method that threw the exception
- args - the argument array to the method that threw the exception, signature is object[]
- target - the AOP target object instance.
- e - the thrown exception

You can invoke methods on these variables, prefixed by a '#' in the expression. This gives you the flexibility to call special purpose constructors that can have any piece of information accessible via the above variables, or even other external data through the use of SpEL's ability to reference objects within the Spring container.

You may also choose to 'swallow' the exception or to return a specific return value, for example

```
on exception name ArithmeticException swallow

or

on exception name ArithmeticException return 12
```

You may also simply log the exception

```
on exception name ArithmeticException,ArgumentException log 'My Message, Method Name ' + #method.Name
```

Here we see that a comma delimited list of exception names can be specified.

The logging is performed using the Commons.Logging library that provides an abstraction over the underlying logging implementation. Logging is currently at the debug level with a logger name of "LogExceptionHandler". The ability to specify these values will be a future enhancement and likely via a syntax resembling a constructor for the action, i.e. log(Debug,"LoggerName").

Multiple exception handling statements can be specified within the list shown above. The processing flow is on exception, the name of the exception listed in the statement is compared to the thrown exception to see if there is a match. A comma separated list of exceptions can be used to group together the same action taken for different exception names. If the action to take is logging, then the logging action is performed and the search for other matching exception names continues. For all other actions, namely translate, wrap, replace, swallow, return, once an exception handler is matched, those in the chain are no longer evaluated. Note, do not confuse

this handler chain with the general advice AOP advice chain. For translate, wrap, and replace actions a SpEL expression is created and used to instantiate a new exception (in addition to any other processing that may occur when evaluating the expression) which is then thrown.

The exception handling DSL also supports the ability to provide a SpEL boolean expression to determine if the advice will apply instead of just filtering by the expression name. For example, the following is the equivalent to the first example based on exception names but compares the specific type of the exception thrown

```
on exception (#e is T(System.ArithmeticException)) wrap System.InvalidOperationException
```

The syntax use is 'on exception (SpEL boolean expression)' and inside the expression you have access to the variables of the calling context listed before, i.e. method, args, target, and e. This can be useful to implement a small amount of conditional logic, such as checking for a specific error number in an exception, i.e. (#e is T(System.Data.SqlException) && #e.Errors[0].Number in {156,170,207,208}), to catch and translate bad grammar codes in a SqlException.

While the examples given above are toy examples, they could just as easily be changed to convert your application specific exceptions. If you find yourself pushing the limits of using SpEL expressions, you will likely be better off creating your own custom aspect class instead of a scripting approach.

You can also configure the each of the Handlers individually based on the action keyword. For example, to configure the logging properties on the LogExceptionHandler.

```
<object name="logExceptionHandler" type="Spring.Aspects.Exceptions.LogExceptionHandler, Spring.Aop">
  <property name="LogName" value="Cms.Session.ExceptionHandler" />
  <property name="LogLevel" value="Debug"/>
  <property name="LogMessageOnly" value="true"/>
</object>

<object name="exceptionHandlingAdvice" type="Spring.Aspects.Exceptions.ExceptionHandlerAdvice, Spring.Aop">
  <property name="ExceptionHandlerDictionary">
    <dictionary>
      <entry key="log" ref="logExceptionHandler"/>
    </dictionary>
  </property>

  <property name="ExceptionHandlers">
    <list>
      <value>on exception name ArithmeticException,ArgumentException log 'My Message, Method Name ' + #method.Name</value>
    </list>
  </property>
</object>
```

You can also configure ExceptionHandlerAdvice to use an instance of IExceptionHandler by specifying it as an entry in the ExceptionHandlers list. This gives you complete control over all properties of the handler but you must set ConstraintExpressionText and ActionExpressionText which are normally parsed for you from the string. To use the case of configuring the LogExceptionHandler, this approach also lets you specify advanced logging functionality, but at a cost of some additional complexity. For example setting the logging level and pass the exception into the logging subsystem

```
<object name="exceptionHandlingAdvice" type="Spring.Aspects.Exceptions.ExceptionHandlerAdvice, Spring.Aop">
  <property name="exceptionHandlers">
    <list>
      <object type="Spring.Aspects.Exceptions.LogExceptionHandler">
        <property name="LogName" value="Cms.Session.ExceptionHandler" />
        <property name="ConstraintExpressionText" value="#e is T(System.Threading.ThreadAbortException)" />
        <property name="ActionExpressionText" value="#log.Fatal('Request Timeout occurred', #e)" />
      </object>
    </list>
  </property>
</object>
```

The configuration of the logger name, level, and whether or not to pass the thrown exception as the second argument to the log method will be supported in the DSL style in a future release.

14.3.1. Language Reference

The general syntax of the language is

```
on exception name [ExceptionName1,ExceptionName2,...] [action] [SpEL expression]
```

or

```
on exception (SpEL boolean expression) [action] [SpEL expression]
```

The exception names are required as well as the action. The valid actions are

- log
- translate
- wrap
- replace
- return
- swallow
- execute

The form of the expression depends on the action. For logging, the entire string is taken as the SpEL expression to log. Translate expects an exception to be returned from evaluation the SpEL expression. Wrap and replace are shorthand for the translate action. For wrap and replace you specify the exception name and the message to pass into the standard exception constructors (string, exception) and (string). The exception name can be a partial or fully qualified name. Spring will attempt to resolve the typename across all referenced assemblies. You may also register type aliases for use with SpEL in the standard manner with Spring.NET and those will be accessible from within the exception handling expression.

14.4. Logging

The logging advice lets you log the information on method entry, exit and thrown exception (if any). The implementation is based on the logging library, [Common.Logging](#), that provides portability across different logging libraries. There are a number of configuration options available, listed below

- LogUniqueIdentifier
- LogExecutionTime
- LogMethodArguments
- LogReturnValue
- Separator
- LogLevel

You declare the logging advice in IoC container with the following XML fragment. Alternatively, you can use the class `SimpleLoggingAdvice` programmatically.

```
<object name="loggingAdvice" type="Spring.Aspects.Logging.SimpleLoggingAdvice, Spring.Aop">
  <property name="LogUniqueIdentifier" value="true"/>
  <property name="LogExecutionTime" value="true"/>
  <property name="LogMethodArguments" value="true"/>
  <property name="LogReturnValue" value="true"/>

  <property name="Separator" value=";"/>
  <property name="LogLevel" value="Info"/>

  <property name="HideProxyTypeNames" value="true"/>
  <property name="UseDynamicLogger" value="true"/>
</object>
```

The default values for `LogUniqueIdentifier`, `LogExecutionTime`, `LogMethodArguments` and `LogReturnValue` are false. The default separator value is `;` and the default log level is `Common.Logging's LogLevel.Trace`.

You can set the name of the logger with the property `LoggerName`, for example `"DataAccessLayer"` for a logging advice that would be applied across the all the classes in the data access layer. That works well when using a 'category' style of logging. If you do not set the `LoggerName` property, then the type name of the logging advice is used as the logging name. Another approach to logging is to log based on the type of the object being called, the target type. Since often this is a proxy class with a relatively meaningless name, the property `HideProxyTypeNames` can be set to true to show the true target type and not the proxy type. The `UseDynamicLogger` property determines which `ILog` instance should be used to write log messages for a particular method invocation: a dynamic one for the Type getting called, or a static one for the Type of the trace interceptor. The default is to use a static logger.

To further extend the functionality of the `SimpleLoggingAdvice` you can subclass `SimpleLoggingAdvice` and override the methods

- `string GetEntryMessage(IMethodInvocation invocation, string idString)`
- `string GetExceptionMessage(IMethodInvocation invocation, Exception e, TimeSpan executionTimeSpan, string idString)`
- `string GetExitMessage(IMethodInvocation invocation, object returnValue, TimeSpan executionTimeSpan, string idString)`

The default implementation to calculate a unique identifier is to use a GUID. You can alter this behavior by overriding the method `string CreateUniqueIdentifier()`. The `SimpleLoggingAdvice` class inherits from `AbstractLoggingAdvice`, which has the abstract method `object InvokeUnderLog(IMethodInvocation invocation, ILog log)` and you can also override the method `ILog GetLoggerForInvocation(IMethodInvocation invocation)` to customize the logger instance used for logging. Refer to the SDK documentation for more details on subclassing `AbstractLoggingAdvice`.

As an example of the Logging advice's output, adding the advice to the method

```
public int Bark(string message, int[] luckyNumbers)
{
    return 4;
}
```

And calling `Bark("hello", new int[]{1, 2, 3})`, results in the following output

```
Entering Bark, 5d2bad47-62cd-435b-8de7-91f12b7f433e, message=hello; luckyNumbers=System.Int32[]
```

```
Exiting Bark, 5d2bad47-62cd-435b-8de7-91f12b7f433e, 30453.125 ms, return=4
```

The method parameters values are obtained using the `ToString()` method. If you would like to have an alternate implementation, say to view some values in an array, override the method `string GetMethodArgumentAsString(IMethodInvocation invocation)`.

14.5. Retry

When making a distributed call it is often a common requirement to be able to retry the method invocation if there was an exception. Typically the exception will be due to a communication issue that is intermittent and retrying over a period of time will likely result in a successful invocation. When applying retry advice it is important to know if making two calls to the remote service will cause side effects. Generally speaking, the method being invoked should be [idempotent](#), that is, it is safe to call multiple times.

The retry advice is specified using a little language, i.e a DSL. A simple example is shown below

```
on exception name ArithmeticException retry 3x delay 1s
```

The meaning is: when an exception that has 'ArithmeticException' in its type name is thrown, retry the invocation up to 3 times and delay for 1 second between each retry event.

You can also provide a SpEL (Spring Expression Language) expression that calculates the time interval to sleep between each retry event. The syntax for this is shown below

```
on exception name ArithmeticException retry 3x rate (1*#n + 0.5)
```

As with the exception handling advice, you may also specify a boolean SpEL that must evaluate to true in order for the advice to apply. For example

```
on exception (#e is T(System.ArithmeticException)) retry 3x delay 1s

on exception (#e is T(System.ArithmeticException)) retry 3x rate (1*#n + 0.5)
```

The time specified after the delay keyword is converted to a `TimeSpan` object using Spring's `TimeSpanConverter`. This supports setting the time as an integer + time unit. Time units are (d, h, m, s, ms) representing (days, hours, minutes, seconds, and milliseconds). For example; 1d = 1day, 5h = 5 hours etc. You can not specify a string such as '1d 5h'. The value that is calculated from the expression after the rate keyword is interpreted as a number of seconds. The power of using SpEL for the rate expression is that you can easily specify some exponential retry rate (a bigger delay for each retry attempt) or call out to a custom function developed for this purpose.

When using a SpEL expression for the filter condition or for the rate expression, the following variable are available

- `method` - the `MethodInfo` object corresponding to the method that threw the exception
- `args` - the argument array to the method that threw the exception, signature is `object[]`
- `target` - the AOP target object instance.
- `e` - the thrown exception

You declare the advice in IoC container with the following XML fragment. Alternatively, you can use the `RetryAdvice` class programmatically.

```
<object name="exceptionHandlingAdvice" type="Spring.Aspects.RetryAdvice, Spring.Aop">
```

```
<property name="retryExpression" value="on exception name ArithmeticException retry 3x delay 1s"/>
</object>
```

14.5.1. Language Reference

The general syntax of the language is

```
on exception name [ExceptionName1,ExceptionName2,...] retry [number of times]x [delay|rate]
[delay time|SpEL rate expression]
```

or

```
on exception (SpEL boolean expression) retry [number of times]x [delay|rate] [delay time|
SpELrate expression]
```

14.6. Transactions

The transaction aspect is more fully described in the section on transaction management.

14.7. Parameter Validation

Spring provides a UI-agnostic validation framework in which you can declare validation rules, both programmatically and declaratively, and have those rules evaluated against an arbitrary .NET object. Spring provides additional support for the rendering of validation errors within Spring's ASP.NET framework. (See the section on ASP.NET usage tips for more information.) However, validation is not confined to the UI tier. It is a common task that occurs across most, if not all, applications layers. Validation that is performed in the UI layer is often repeated in the service layer, in order to be proactive in case non UI-based clients invoke the service layer. Validation rules completely different from those used in the UI layer may also be used on the server side.

To address some of the common needs for validation on the server side, Spring provides parameter validation advice so that applies Spring's validation rules to the method parameters. The class `ParameterValidationAdvice` is used in conjunction with the `Validated` attribute to specify which validation rules are applied to method parameters. For example, to apply parameter validation to the method `SuggestFlights` in the `BookingAgent` class used in the `SpringAir` sample application, you would apply the `Validated` attribute to the method parameters as shown below.

```
public FlightSuggestions SuggestFlights( [Validated("tripValidator")] Trip trip)
{
    // unmodified implementation goes here
}
```

The `Validated` attribute takes a string name that specifies the name of the validation rule, i.e. the name of the `IValidator` object in the Spring application context. The `Validated` attribute is located in the namespace `Spring.Validation` of the `Spring.Core` assembly.

The configuration of the advice is to simply define the an instance of the `ParameterValidationAdvice` class and apply the advice, for example based on object names using an `ObjectNameAutoProxyCreator`, as shown below,

```
<object id="validationAdvice" type="Spring.Aspects.Validation.ParameterValidationAdvice, Spring.Aop"/>

<object type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator, Spring.Aop">
  <property name="ObjectNames">
    <list>
      <value>bookingAgent</value>
    </list>
  </property>
```

```
<property name="InterceptorNames">
  <list>
    <value>validationAdvice</value>
  </list>
</property>
</object>
```

When the advised method is invoked first the validation of each method parameter is performed. If all validation succeeds, then the method body is executed. If validation fails an exception of the type `ValidationException` is thrown and you can retrieve errors information from its property `ValidationErrors`. See the SDK documentation for details.

Chapter 15. Common Logging

15.1. Introduction

Spring uses a simple logging abstraction in order to provide a layer of indirection between logging calls made by Spring and the specific logging library used in your application (log4net, EntLib logging, NLog). The library is available for .NET 1.0, 1.1, and 2.0 with both debug and strongly signed assemblies. Since this need is not specific to Spring, the logging library was moved out of the Spring project and into a more general open source project called [Common Infrastructure Libraries for .NET](#). The logging abstraction within the project is known as Common.Logging. Note that it is not the intention of this library to be a replacement for the many fine logging libraries that are out there. The API is incredibly minimal and will very likely stay that way. Please note that this library is intended only for use where the paramount requirement is portability and you will generally be better served by using a specific logging implementation so that you can leverage its advanced features and extended APIs to your advantage.

You can find online documentation on how to configure Common.Logging is available in [HTML](#) , [PDF](#), and [HTML Help](#) formats.

Chapter 16. Testing

16.1. Introduction

The Spring team considers developer testing to be an absolutely integral part of enterprise software development. A thorough treatment of testing in the enterprise is beyond the scope of this chapter; rather, the focus here is on the value add that the adoption of the IoC principle can bring to unit testing; and on the benefits that the Spring Framework provides in integration testing.

16.2. Unit testing

One of the main benefits of Dependency Injection is that your code is much less likely to have any hidden dependencies on the runtime environment or other configuration subsystems. This allows for unit tests to be written in a manner such that the object under test can be simply instantiated with the `new` operator and have its dependences set in the unit test code. You can use mock objects (in conjunction with many other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations around Spring you will find that the resulting clean layering and componentization of your codebase will naturally facilitate *easier* unit testing. For example, you will be able to test service layer objects by stubbing or mocking DAO interfaces, without any need to access persistent data while running unit tests.

True unit tests typically will run extremely quickly, as there is no runtime infrastructure to set up, i.e., database, ORM tool, or whatever. Thus emphasizing true unit tests as part of your development methodology will boost your productivity. The upshot of this is that you do not need this section of the testing chapter to help you write effective *unit* tests for your IoC-based applications.

16.3. Integration testing

However, it is also important to be able to perform some integration testing enabling you to test things such as:

- The correct wiring of your Spring IoC container contexts.
- Data access using ADO.NET or an ORM tool. This would include such things such as the correctness of SQL statements / or NHibernate XML mapping files.

The Spring Framework provides support for integration testing when using NUnit and Microsoft's Testing framework 'MSTest'. The NUnit classes are located in the assembly `Spring.Testing.NUnit.dll` and the MSTest is located in `Spring.Testing.Microsoft.dll`.



Note

The `Spring.Testing.NUnit.dll` library is compiled against NUnit 2.5.1. Note that test runners integrated inside VS.NET may or may not support this version. At the time of this writing Reshaper 4.5.0 did not properly support NUnit 2.5.1. To use Resharper with NUnit 2.5.1 you need to download 4.5.1 RC2 or later.

These namespaces provides NUnit and MSTest superclasses for integration testing using a Spring container.

These superclasses provide the following functionality:

- Spring IoC container caching between test case execution.

- The pretty-much-transparent Dependency Injection of test fixture instances (this is nice).
- Transaction management appropriate to integration testing (this is even nicer).
- A number of Spring-specific inherited instance variables that are really useful when integration testing.

16.3.1. Context management and caching

The `Spring.Testing.NUnit` and `Spring.Testing.Microsoft` namespace provides support for consistent loading of Spring contexts, and caching of loaded contexts. Similarly `Spring.TestingSupport` for the caching of loaded contexts is important, because if you are working on a large project, startup time may become an issue - not because of the overhead of Spring itself, but because the objects instantiated by the Spring container will themselves take time to instantiate. For example, a project with 50-100 NHibernate mapping files might take 10-20 seconds to load the mapping files, and incurring that cost before running every single test case in every single test fixture will lead to slower overall test runs that could reduce productivity.

To address this issue, the `AbstractDependencyInjectionSpringContextTests` has an protected property that subclasses must implement to provide the location of context definition files:

```
protected abstract string[] ConfigLocations { get; }
```

Implementations of this method must provide an array containing the `IResource` locations of XML configuration metadata used to configure the application. This will be the same, or nearly the same, as the list of configuration locations specified in `App.config/Web.config` or other deployment configuration.

By default, once loaded, the configuration file set will be reused for each test case. Thus the setup cost will be incurred only once (per test fixture), and subsequent test execution will be much faster. In the unlikely case that a test may 'dirty' the config location, requiring reloading - for example, by changing an object definition or the state of an application object - you can call the `SetDirty()` method on `AbstractDependencyInjectionSpringContextTests` to cause the test fixture to reload the configurations and rebuild the application context before executing the next test case.

16.3.2. Dependency Injection of test fixtures

When `AbstractDependencyInjectionSpringContextTests` (and subclasses) load your application context, they can optionally configure instances of your test classes by Setter Injection. All you need to do is to define instance variables and the corresponding setters. `AbstractDependencyInjectionSpringContextTests` will automatically locate the corresponding object in the set of configuration files specified in the `ConfigLocations` property.

Consider the scenario where we have a class, `HibernateTitleDao`, that performs data access logic for say, the `Title` domain object. We want to write integration tests that test all of the following areas:

- The Spring configuration; basically, is everything related to the configuration of the `HibernateTitleDao` object correct and present?
- The Hibernate mapping file configuration; is everything mapped correctly and are the correct lazy-loading settings in place?
- The logic of the `HibernateTitleDao`; does the configured instance of this class perform as anticipated?

Let's look at the NUnit test class itself (we will look at the configuration immediately afterwards).

```
/// Using NUnit

[TestFixture]
public class HibernateTitleDaoTests : AbstractDependencyInjectionSpringContextTests {
```

```
// this instance will be (automatically) dependency injected
private HibernateTitleDao titleDao;

// a setter method to enable DI of the 'titleDao' instance variable
public HibernateTitleDao HibernateTitleDao {
    set { titleDao = value; }
}

[Test]
public void LoadTitle() {
    Title title = this.titleDao.LoadTitle(10);
    Assert.IsNotNull(title);
}

// specifies the Spring configuration to load for this test fixture
protected override string[] ConfigLocations {
    return new String[] { "assembly://MyAssembly/MyNamespace/daos.xml" };
}
}
```

The file referenced by the ConfigLocations method ('classpath:com/foo/daos.xml') looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net">

    <!-- this object will be injected into the HibernateTitleDaoTests class -->
    <object id="titleDao" type="Spring.Samples.HibernateTitleDao, Spring.Samples">
        <property name="sessionFactory" ref="sessionFactory"/>
    </object>

    <object id="sessionFactory" type="Spring.Data.NHibernate.LocalSessionFactoryObject, Spring.Data.NHibernate">
        <!-- dependencies elided for clarity -->
    </object>

</objects>
```

The `AbstractDependencyInjectionSpringContextTests` classes uses *autowire by type*. Thus if you have multiple object definitions of the same type, you cannot rely on this approach for those particular object. In that case, you can use the inherited `applicationContext` instance variable, and explicit lookup using (for example) an explicit call to `applicationContext.GetObject("titleDao")`.

Using `AbstractDependencyInjectionSpringContextTests` with `MSTest` is very similar.

```
/// Using Microsoft's Testing Framework

[TestClass]
public class HibernateTitleDaoTests : AbstractDependencyInjectionSpringContextTests {

    // this instance will be (automatically) dependency injected
    private HibernateTitleDao titleDao;

    // a setter method to enable DI of the 'titleDao' instance variable
    public HibernateTitleDao HibernateTitleDao {
        set { titleDao = value; }
    }

    [Test]
    public void LoadTitle() {
        Title title = this.titleDao.LoadTitle(10);
        Assert.IsNotNull(title);
    }

    // specifies the Spring configuration to load for this test fixture
    protected override string[] ConfigLocations {
        return new String[] { "assembly://MyAssembly/MyNamespace/daos.xml" };
    }
}
```

If you don't want dependency injection applied to your test cases, simply don't declare any set properties. Alternatively, you can extend the `AbstractSpringContextTests` - the root of the class hierarchy in the `Spring.Testing.NUnit` and `Spring.Testing.Microsoft` namespaces. It merely contains convenience methods to load Spring contexts, and performs no Dependency Injection of the test fixture.

16.3.2.1. Field level injection

If, for whatever reason, you don't fancy having setter properties in your test fixtures, Spring can (in this one case) inject dependencies into protected fields. Find below a reworking of the previous example to use field level injection (the Spring XML configuration does not need to change, merely the test fixture).

```
[TestFixture]
public class HibernateTitleDaoTests : AbstractDependencyInjectionSpringContextTests{

    public HibernateTitleDaoTests() {
        // switch on field level injection
        PopulateProtectedVariables = true;
    }

    // this instance will be (automatically) dependency injected
    protected HibernateTitleDao titleDao;

    [TestMethod]
    public void LoadTitle() {
        Title title = this.titleDao.LoadTitle(10);
        Assert.IsNotNull(title);
    }

    // specifies the Spring configuration to load for this test fixture
    protected override string[] ConfigLocations {
        return new String[] { "assembly://MyAssembly/MyNamespace/daos.xml" };
    }
}
```

In the case of field injection, there is no autowiring going on: the name of your protected instances variable(s) are used as the lookup object name in the configured Spring container.

16.3.3. Transaction management

One common issue in tests that access a real database is their effect on the state of the persistence store. Even when you're using a development database, changes to the state may affect future tests. Also, many operations - such as inserting to or modifying persistent data - cannot be done (or verified) outside a transaction.

The `AbstractTransactionalDbProviderSpringContextTests` superclass (and subclasses) exist to meet this need. By default, they create and roll back a transaction for each test. You simply write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they will behave correctly, according to their transactional semantics.

`AbstractTransactionalSpringContextTests` depends on a `IPlatformTransactionManager` object being defined in the application context. The name doesn't matter, due to the use of `autowire` by type.

Typically you will extend the subclass, `AbstractTransactionalDbProviderSpringContextTests`. This also requires that a `DbProvider` object definition - again, with any name - be present in the configurations. It creates an `AdoTemplate` instance variable that is useful for convenient querying, and provides handy methods to delete the contents of selected tables (remember that the transaction will roll back by default, so this is safe to do).

If you want a transaction to commit - unusual, but occasionally useful when you want a particular test to populate the database - you can call the `SetComplete()` method inherited from `AbstractTransactionalSpringContextTests`. This will cause the transaction to commit instead of roll back.

There is also convenient ability to end a transaction before the test case ends, through calling the `EndTransaction()` method. This will roll back the transaction by default, and commit it only if `SetComplete()` had previously been called. This functionality is useful if you want to test the behavior of 'disconnected' data objects, such as Hibernate-mapped objects that will be used in a web or remoting tier outside a transaction. Often, lazy loading errors are discovered only through UI testing; if you call `EndTransaction()` you can ensure correct operation of the UI through your NUnit test suite.

16.3.4. Convenience variables

When you extend the `AbstractTransactionalDbProviderSpringContextTests` class you will have access to the following protected instance variables:

- `applicationContext` (a `IConfigurableApplicationContext`): inherited from the `AbstractDependencyInjectionSpringContextTests` superclass. Use this to perform explicit object lookup, or test the state of the context as a whole.
- `adoTemplate`: inherited from `AbstractTransactionalDbProviderSpringContextTests`. Useful for querying to confirm state. For example, you might query before and after testing application code that creates an object and persists it using an ORM tool, to verify that the data appears in the database. (Spring will ensure that the query runs in the scope of the same transaction.) You will need to tell your ORM tool to 'flush' its changes for this to work correctly, for example using the `Flush()` method on NHibernate's `ISession` interface.

Often you will provide an application-wide superclass for integration tests that provides further useful instance variables used in many tests.

Part II. Middle Tier Data Access

This part of the reference documentation is concerned with the middle tier, and specifically the data access responsibilities of said tier.

Spring's comprehensive transaction management support is covered in some detail, followed by thorough coverage of the various middle tier data access frameworks and technologies that the Spring Framework integrates with.

- Chapter 17, *Transaction management*
- Chapter 18, *DAO support*
- Chapter 19, *DbProvider*
- Chapter 20, *Data access using ADO.NET*
- Chapter 21, *Object Relational Mapping (ORM) data access*

Chapter 17. Transaction management

17.1. Introduction

Spring.NET provides a consistent abstraction for transaction management that provides the following benefits

- Provides a consistent programming model across different transaction APIs such as ADO.NET, Enterprise Services, System.Transactions, and NHibernate.
- Support for declarative transaction management with any of the above data access technologies
- Provides a simple API for programmatic transaction management
- Integrates with Spring's high level persistence integration APIs such as AdoTemplate.

This chapter is divided up into a number of sections, each detailing one of the value-adds or technologies of the Spring Framework's transaction support. The chapter closes with some discussion of best practices surrounding transaction management.

- The first section, entitled Motivations describes why one would want to use the Spring Framework's transaction abstraction as opposed to using System.Transactions or a specific data access technology transaction API.
- The second section, entitled Key Abstractions outline the core classes as well as how to configure them.
- The third section, entitled Declarative transaction management, covers support for declarative transaction management.
- The fourth section, entitled Programmatic transaction management, covers support for programmatic transaction management.

17.2. Motivations

The data access technology landscape is a broad one, within the .NET BCL there are three APIs for performing transaction management, namely ADO.NET, Enterprise Services, and System.Transactions. Other data access technologies such as object relational mappers and result-set mapping libraries are also gaining in popularity and each come with their own APIs for transaction management. As such, code is often directly tied to a particular transaction API which means you must make an up-front decision which API to use in your application. Furthermore, if the need arises to change your approach, it quite often will not be a simple refactoring. Using Spring's transaction API you can keep the same API across different data access technologies. Changing the underlying transaction implementation that is used is a simple matter of configuration or a centralized programmatic change as compared to a major overhauling.

Hand in hand with the variety of options available is the establishment generally agreed upon best practices for data access. Martin Fowler's book, Patterns of Enterprise Application Architecture, is an excellent source of approaches to data access that have been successful in the real world. One approach that is quite common is to introduce a data access layer into your architecture. The data access layer is concerned not only with providing some portability between different data access technologies and databases but its scope is strictly related to data access. A simple data access layer would be not much more than data access objects (DAOs) with 'Create/Retrieve/Update/Delete' (CRUD) methods devoid of any business logic. Business logic resides in another application layer, the business service layer, in which business logic will call one or more DAOs to fulfill a higher level end-user function.

In order to perform this end-user function with all-or-nothing transactional semantics, the transaction context is controlled by the business service layer (or other 'higher' layers). In such a common scenario, an important implementation detail is how to make the DAO objects aware of the 'outer' transaction started in another layer. A simplistic implementation of a DAO would perform its own connection and transaction management, but this would not allow grouping of DAO operations with the same transaction as the DAO is doing its own transaction/resource management. As such there needs to be a means to transfer the connection/transaction pair managed in the business service layer to the DAOs. There are a variety of ways to do this, the most invasive being the explicitly pass a connection/transaction object as method arguments to your DAOs. Another way is to store the connection/transaction pair in thread local storage. In either case, if you are using ADO.NET you must invent some infrastructure code to perform this task.

But wait, doesn't Enterprise Services solve this problem - and what about the functionality in the System.Transactions namespace? The answer is yes...and no. Enterprise Services lets you use the 'raw' ADO.NET API within a transaction context such that multiple DAO operations are grouped within the same transaction. The downside to Enterprise Services is that it always uses distributed (global) transactions via the Microsoft Distributed Transaction Coordinator (MS-DTC). For most applications this is overkill just to get this functionality as global transactions are significantly less performant than local ADO.NET transactions.

There are similar issues with using the 'using TransactionScope' construct within the new System.Transactions namespace. The goal with TransactionScope is to define a, well - transaction scope - within a using statement. Plain ADO.NET code within that using block will then be a local ADO.NET based transaction if only a single transactional resource is accessed. However, the 'magic' of System.Transactions (and the database) is that local transactions will be promoted to distributed transactions when a second transaction resource is detected. The name that this goes by is Promotable Single Phase Enlistment (PSPE). However, there is a big caveat - opening up a second IDbConnection object to the same database with the same database string will trigger promotion from local to global transactions. As such, if your DAOs are performing their own connection management you will end up being bumped up to a distributed transaction. In order to avoid this situation for the common case of an application using a single database, you must pass around a connection object to your DAOs. It is also worth to note that many database providers (Oracle for sure) do not yet support PSPE and as such will always use a distributed transaction even if there is only a single database.

Last but not least is the ability to use declarative transaction management. Not many topics in database transaction-land give developers as much 'bang-for-the-buck' as declarative transactions since the noisy tedious bits of transactional API code in your application are pushed to the edges, usually in the form of class/method attributes. Only Enterprise Services offers this feature in the BCL. Spring fills the gap - it provides declarative transaction management if you are using local ADO.NET or System.Transactions (the most popular) or other data access technologies. Enterprise Services is not without its small warts as well, such as the need to separate your query/retrieve operations from your create/update/delete operations if you want to use different isolation levels since declarative transaction metadata can only be applied at the class level. Nevertheless, all in all, Enterprise Services, in particular with the new 'Services Without Components' implementation for XP SP2/Server 2003, and hosted within the same process as your application code is as good as it gets out of the .NET box. Despite these positive points, it hasn't gained a significant mindshare in the development community.

Spring's transaction support aims to relieve these 'pain-points' using the data access technologies within the BCL - and for other third party data access technologies as well. It provides declarative transaction management with a configurable means to obtain transaction option metadata - out of the box attributes and XML within Spring's IoC configuration file are supported.

Finally, Spring's transaction support lets you mix data access technologies within a single transaction - for example ADO.NET and NHibernate operations.

With this long winded touchy/feely motivational section behind us, lets move on to see the code.

17.3. Key Abstractions

The key to the Spring transaction management abstraction is the notion of a *transaction strategy*. A transaction strategy is defined by the `Spring.Transaction.IPlatformTransactionManager` interface, shown below:

```
public interface IPlatformTransactionManager {

    ITransactionStatus GetTransaction( ITransactionDefinition definition );

    void Commit( ITransactionStatus transactionStatus );

    void Rollback( ITransactionStatus transactionStatus );

}
```

This is primarily a 'SPI' (Service Provider Interface), although it can be used Programatically. Note that in keeping with the Spring Framework's philosophy, `IPlatformTransactionManager` is an interface, and can thus be easily mocked or stubbed as necessary. `IPlatformTransactionManager` implementations are defined like any other object in the IoC container. The following implementations are provided

- `AdoPlatformTransactionManager` - local ADO.NET based transactions
- `ServiceDomainPlatformTransactionManager` - distributed transaction manager from Enterprise Services
- `TxScopePlatformTransactionManager` - local/distributed transaction manager from System.Transactions.
- `HibernatePlatformTransactionManager` - local transaction manager for use with NHibernate or mixed ADO.NET/NHibernate data access operations.

Under the covers, the following API calls are made. For the `AdoPlatformTransactionManager`, `Transaction.Begin()`, `Commit()`, `Rollback()`. `ServiceDomainPlatformTransactionManager` uses the 'Services without Components' update so that your objects do not need to inherit from `ServicedComponent` or directly call the Enterprise Services API `ServiceDomain.Enter()`, `Leave()`, `ContextUtil.SetAbort()`. `TxScopePlatformTransactionManager` calls; `new TransactionScope()`, `.Complete()`, `Dispose()`, `Transaction.Current.Rollback()`. Configuration properties for each transaction manager are specific to the data access technology used. Refer to the API docs for comprehensive information but the examples should give you a good basis for getting started. The `HibernatePlatformTransactionManager` is described more in the following section .

The `GetTransaction(...)` method returns a `ITransactionStatus` object, depending on a `ITransactionDefinition` parameters. The returned `ITransactionStatus` might represent a new or existing transaction (if there was a matching transaction in the current call stack - with the implication being that a `ITransactionStatus` is associated with a logical thread of execution.

The `ITransactionDefinition` interface specified

- **Isolation:** the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- **Propagation:** normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction.

- Timeout: how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).
- Read-only status: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using NHibernate).

These settings reflect standard transactional concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts because understanding such core concepts is essential to using the Spring Framework or indeed any other transaction management solution.

The `ITransactionStatus` interface provides a simple way for transactional code to control transaction execution and query transaction status.

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `IPlatformTransactionManager` implementation is absolutely essential. In good Spring fashion, this important definition typically is made using via Dependency Injection.

`IPlatformTransactionManager` implementations normally require knowledge of the environment in which they work, ADO.NET, NHibernate, etc. The following example shows how a standard ADO.NET based `IPlatformTransactionManager` can be defined.

We must define a Spring `IDbProvider` and then use Spring's `AdoPlatformTransactionManager`, giving it a reference to the `IDbProvider`. For more information on the `IDbProvider` abstraction refer to the next chapter.

```
<objects xmlns='http://www.springframework.net'
         xmlns:db="http://www.springframework.net/database">

    <db:provider id="DbProvider"
                provider="SqlServer-1.1"
                connectionString="Data Source=(local);Database=Spring;User ID=springqa;Password=springqa;Trusted_Connection=..." />

    <object id="TransactionManager"
           type="Spring.Data.Core.AdoPlatformTransactionManager, Spring.Data">
        <property name="DbProvider" ref="DbProvider"/>
    </object>

    . . . other object definitions . . .

</objects>
```

We can also use a transaction manager based on `System.Transactions` just as easily, as shown in the following example

```
<object id="TransactionManager"
       type="Spring.Data.Core.TxScopeTransactionManager, Spring.Data">
</object>
```

Similarly for the `HibernateTransactionManager` as shown in the section on ORM transaction management.

Note that in all these cases, application code will not need to change at all since, dependency injection is a perfect companion to using the strategy pattern. We can now change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

17.4. Resource synchronization with transactions

How does application code participate with the resources (i.e. Connection/Transactions/Sessions) that are created/reused/cleaned up via the different transaction managers? There are two approaches - a high-level and a low-level approach

17.4.1. High-level approach

The preferred approach is to use Spring's high level persistence integration APIs. These do not replace native APIs, but internally handle resource creation/reuse, cleanup, and optional transaction synchronization (i.e. event notification) of the resources and exception mapping so that user data access code doesn't have to worry about these concerns at all, but can concentrate purely on non-boilerplate persistence logic. Generally, the same inversion of control approach is used for all persistence APIs. In this approach the API has a callback method or delegate that presents the user code with the relevant resource ready to use - i.e. a `DbCommand` with its `Connection` and `Transaction` properties set based on the transaction option metadata. These classes go by the naming scheme 'template', examples of which are `AdoTemplate` and `HibernateTemplate`. Convenient 'one-liner' helper methods in these template classes build upon the core callback/IoC design by providing specific implementations of the callback interface.

17.4.2. Low-level approach

A utility class can be used to directly obtain a connection/transaction pair that is aware of the transactional calling context and returns a pair suitable for that context. The class `ConnectionUtils` contains the static method `ConnectionTxPair GetConnectionTxPair(IDbProvider provider)` which serves this purpose.

```
public class LowLevelIntegration
{
    // Spring's IDbProvider abstraction
    private IDbProvider dbProvider;

    public IDbProvider dbProvider
    {
        set { dbProvider = value; }
    }

    public void DoWork() {
        ConnectionTxPair connTxPair = ConnectionUtils.GetConnectionTxPair(dbProvider);
        //Use some data access library that allows you to pass in the transaction

        DbWrapper dbWrapper = new DbWrapper();
        string cmdText = ... // some command text
        dbWrapper.ExecuteNonQuery(cmdText, connTxPair.Transaction);
    }
}
```

17.5. Declarative transaction management

Most Spring users choose declarative transaction management. It is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

Spring's declarative transaction management is made possible with Spring's aspect-oriented programming (AOP), although, as the transactional aspects code comes with Spring and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

The approach is to specify transaction behavior (or lack of it) down to the individual method level. It is also possible to mark a transaction for rollback by calling the `SetRollbackOnly()` method within a transaction context if necessary. Some of the highlights of Spring's declarative transaction management are:

- Declarative Transaction management works in any environment. It can work with ADO.NET, System.Transactions, NHibernate etc, with configuration changes only.
- Enables declarative transaction management to be applied to any class, not merely special classes such as those that inherit from `ServiceComponent` or other infrastructure related base classes.

- Declarative rollback rules. Rollback rules can be control declaratively and allow for only specified exceptions thrown within a transactional context to trigger a rollback
- Spring gives you an opportunity to customize transactional behavior, using AOP. For example if you want to insert custom behavior in the case of a transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice.
- Spring does not support propagation of transaction context across remote calls.

The concept of rollback rules is important: they enable us to specify which exceptions should cause automatic roll back. We specify this declaratively, in configuration, not in code. So, although you can still call `SetRollbackOnly()` on the `ITransactionStatus` object to roll the current transaction back, most often you can specify a rule that `MyApplicationException` must always result in rollback. This has the significant advantage that business objects do not depend on the transaction infrastructure. For example, they typically don't need to import any Spring transaction APIs or other Spring APIs. However, to rollback the transaction programmatically when using declarative transaction management, use the utility method

```
TransactionInterceptor.CurrentTransactionStatus.SetRollbackOnly();
```



Note

Prior to Spring.NET 1.2 RC1 the API call would be `TransactionInterceptor.CurrentTransactionStatus.RollbackOnly = true;`

17.5.1. Understanding Spring's declarative transaction implementation

It is not sufficient to tell you simply to annotate your classes with the `[Transaction]` attribute, add the line (`<tx:attribute-driven/>`) to your configuration, and then expect you to understand how it all works. This section explains the inner workings of the Spring Framework's declarative transaction infrastructure in the event of transaction-related issues.

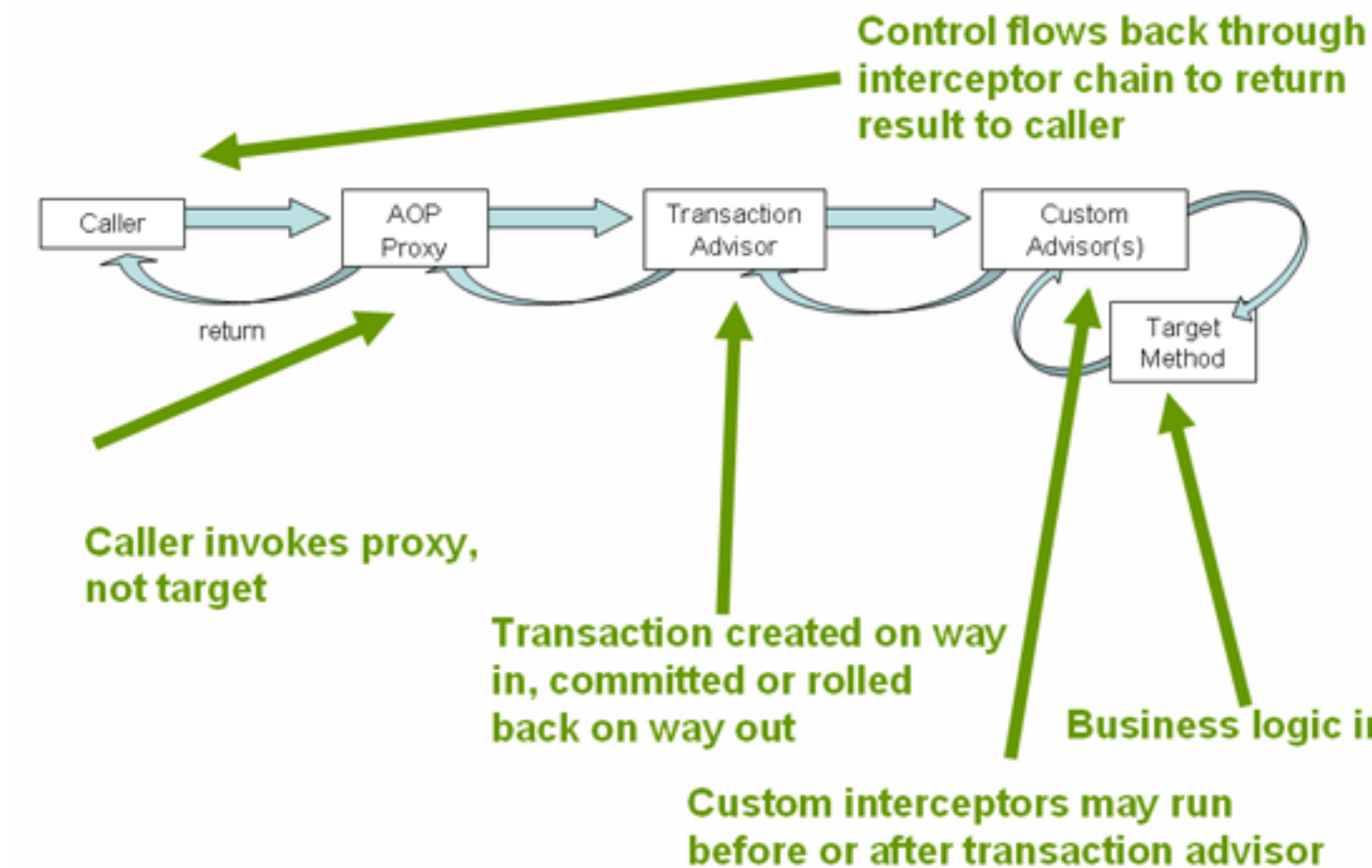
The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled via AOP proxies, and that the transactional advice is driven by metadata (currently XML- or attribute-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a [TransactionInterceptor](#) in conjunction with an appropriate [IPlatformTransactionManager](#) implementation to drive transactions around method invocations.



Note

Spring AOP is covered in Chapter 13, *Aspect Oriented Programming with Spring.NET*

Conceptually, calling a method on a transactional proxy looks like this.



17.5.2. Example of declarative transaction implementation

Consider the following interface. The intent is to convey the concepts to you so you can concentrate on the transaction usage and not have to worry about domain specific details.



Note

A QuickStart application for declarative transaction management is included in the Spring.NET distribution and is described here.

The `ITestObjectManager` is a poor-mans business service layer - the implementation of which will make two DAO calls. Clearly this example is overly simplistic from the service layer perspective as there isn't any business logic at all!. The 'service' interface is shown below.

```

public interface ITestObjectManager
{
    void SaveTwoTestObjects(TestObject to1, TestObject to2);

    void DeleteTwoTestObjects(string name1, string name2);
}

```

The implementation of `ITestObjectManager` is shown below

```

public class TestObjectManager : ITestObjectManager
{
    // Fields/Properties omitted

    [Transaction]
    public void SaveTwoTestObjects(TestObject to1, TestObject to2)
    {
    }
}

```

```

{
    TestObjectDao.Create(to1.Name, to1.Age);
    TestObjectDao.Create(to2.Name, to1.Age);
}

[Transaction]
public void DeleteTwoTestObjects(string name1, string name2)
{
    TestObjectDao.Delete(name1);
    TestObjectDao.Delete(name2);
}
}

```

Note the Transaction attribute on the methods. Other options such as isolation level can also be specified but in this example the default settings are used. However, please note that the mere presence of the Transaction attribute is not enough to actually turn on the transactional behavior - the Transaction attribute is simply metadata that can be consumed by something that is Transaction attribute-aware and that can use the said metadata to configure the appropriate objects with transactional behavior.

The TestObjectDao property has basic create update delete and find method for the 'domain' object TestObject. TestObject in turn has simple properties like name and age.

```

public interface ITestObjectDao
{
    void Create(string name, int age);
    void Update(TestObject to);
    void Delete(string name);
    TestObject FindByName(string name);
    IList FindAll();
}

```

The Create and Delete method implementation is shown below. Note that this uses the AdoTemplate class discussed in the following chapter. Refer to Section 17.4, “Resource synchronization with transactions” for information on the interaction between Spring's high level persistence integration APIs and transaction management features.

```

public class TestObjectDao : AdoDaoSupport, ITestObjectDao
{
    public void Create(string name, int age)
    {
        AdoTemplate.ExecuteNonQuery(CommandType.Text,
            String.Format("insert into TestObjects(Age, Name) VALUES ({0}, '{1}']",
                age, name));
    }

    public void Delete(string name)
    {
        AdoTemplate.ExecuteNonQuery(CommandType.Text,
            String.Format("delete from TestObjects where Name = '{0}'",
                name));
    }
}

```

The TestObjectManager is configured with the DAO objects by standard dependency injection techniques. The client code, which in this case directly asks the Spring IoC container for an instance of ITestObjectManager, will receive a transaction proxy with transaction options based on the attribute metadata. Note that typically the ITestObjectManager would be set on yet another higher level object via dependency injection, for example a web service.

The client calling code is shown below

```

IApplicationContext ctx =
    new XmlApplicationContext("assembly://Spring.Data.Integration.Tests/Spring.Data/autoDeclarativeServices.xml");

```

```

ITestObjectManager mgr = ctx["testObjectManager"] as ITestObjectManager;

TestObject to1 = new TestObject();
to1.Name = "Jack";
to1.Age = 7;

TestObject to2 = new TestObject();
to2.Name = "Jill";
to2.Age = 8;

mgr.SaveTwoTestObjects(to1, to2);

mgr.DeleteTwoTestObjects("Jack", "Jill");

```

The configuration of the object definitions of the DAO and manager classes is shown below.

```

<objects xmlns='http://www.springframework.net'
        xmlns:db='http://www.springframework.net/database'>

    <db:provider id="DbProvider"
        provider="SqlServer-1.1"
        connectionString="Data Source=(local);Database=Spring;User ID=springqa;Password=springqa;Trusted_Connection=
        yes;" />

    <object id="transactionManager"
        type="Spring.Data.Core.AdoPlatformTransactionManager, Spring.Data">

        <property name="DbProvider" ref="DbProvider"/>
    </object>

    <object id="adoTemplate" type="Spring.Data.AdoTemplate, Spring.Data">
        <property name="DbProvider" ref="DbProvider"/>
    </object>

    <object id="testObjectDao" type="Spring.Data.TestObjectDao, Spring.Data.Integration.Tests">
        <property name="AdoTemplate" ref="adoTemplate"/>
    </object>

    <!-- The object that performs multiple data access operations -->
    <object id="testObjectManager" type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
        <property name="TestObjectDao" ref="testObjectDao"/>
    </object>

</objects>

```

This is standard Spring configuration and as such provides you with the flexibility to parameterize your connection string and to easily switch implementations of your DAO objects.

The following section shows how to configure the declarative transactions using Spring's transaction namespace.

17.5.3. Declarative transactions using the transaction namespace

Spring provides a custom XML schema to simplify the configuration of declarative transaction management. If you would like to perform attribute driven transaction management you first need to register the custom namespace parser for the transaction namespace. This can be done in the application configuration file as shown below

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>

    <configSections>

        <sectionGroup name="spring">
            <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core" />

            <!-- other spring config sections like context, typeAliases, etc not shown for brevity -->
        </sectionGroup>
    </configSections>

```

```

</sectionGroup>
</configSections>

<spring>

  <parsers>
    <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
    <parser type="Spring.Transaction.Config.TxNamespaceParser, Spring.Data" />
    <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
  </parsers>

</spring>

</configSections>

```

Instead of using the XML configuration listed at the end of the previous section (declarativeServices.xml) you can use the following. Note that the schemaLocation in the objects element is needed only if you have not installed Spring's schema into the proper VS.NET 2005 location. See the chapter on VS.NET integration for more details.

```

<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.net/tx"
  xmlns:db="http://www.springframework.net/database"
  xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/schema/objects/spring-objects.xsd
  http://www.springframework.net/schema/tx http://www.springframework.net/schema/tx/spring-tx-1.1.xsd"
  http://www.springframework.net/schema/db http://www.springframework.net/schema/db/spring-database.xsd">

  <db:provider id="DbProvider"
    provider="SqlServer-1.1"
    connectionString="Data Source=(local);Database=Spring;User ID=springqa;Password=springqa;Trusted_Connection=

  <object id="transactionManager"
    type="Spring.Data.Core.AdoPlatformTransactionManager, Spring.Data">

    <property name="DbProvider" ref="DbProvider"/>
  </object>

  <object id="adoTemplate" type="Spring.Data.AdoTemplate, Spring.Data">
    <property name="DbProvider" ref="DbProvider"/>
  </object>

  <object id="testObjectDao" type="Spring.Data.TestObjectDao, Spring.Data.Integration.Tests">
    <property name="AdoTemplate" ref="adoTemplate"/>
  </object>

  <!-- The object that performs multiple data access operations -->
  <object id="testObjectManager"
    type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
    <property name="TestObjectDao" ref="testObjectDao"/>
  </object>

  <tx:attribute-driven transaction-manager="transactionManager"/>

</objects>

```



Tip

You can actually omit the 'transaction-manager' attribute in the <tx:attribute-driven/> tag if the object name of the IPlatformTransactionManager that you want to wire in has the name 'transactionManager'. If the PlatformTransactionManager object that you want to dependency inject has any other name, then you have to be explicit and use the 'transaction-manager' attribute as in the example above.

The various optional elements of the <tx:attribute-driven/> tag are summarised in the following table

Table 17.1. `<tx:annotation-driven/>` settings

Attribute	Required?	Default	Description
transaction-manager	No	transactionManager	The name of transaction manager to use. Only required if the name of the transaction manager is not <code>transactionManager</code> , as in the example above.
proxy-target-type	No		Controls what type of transactional proxies are created for classes annotated with the <code>[Transaction]</code> attribute. If "proxy-target-type" attribute is set to "true", then class-based proxies will be created (proxy inherits from target class, however calls are still delegated to target object via composition. This allows for casting to base class. If "proxy-target-type" is "false" or if the attribute is omitted, then a pure composition based proxy is created and you can only cast the proxy to implemented interfaces. (See the section entitled Section 13.6, “Proxying mechanisms” for a detailed examination of the different proxy types.)
order	No		Defines the order of the transaction advice that will be applied to objects annotated with <code>[Transaction]</code> . More on the rules related to ordering of AOP advice can be found in the AOP chapter (see section Section 13.3.2.5, “Advice Ordering”). Note that not specifying any ordering

Attribute	Required?	Default	Description
			will leave the decision as to what order advice is run in to the AOP subsystem.



Note

The "proxy-target-type" attribute on the `<tx:attribute-driven/>` element controls what type of transactional proxies are created for classes annotated with the `Transaction` attribute. If "proxy-target-type" attribute is set to "true", then inheritance-based proxies will be created. If "proxy-target-type" is "false" or if the attribute is omitted, then composition based proxies will be created. (See the section entitled Section 13.6, “Proxying mechanisms” for a detailed examination of the different proxy types.)

You can also define the transactional semantics you want to apply through the use of a `<tx:advice>` definition. This lets you define the transaction metadata such as propagation and isolation level as well as the methods for which that metadata applies external to the code unlike the case of using the transaction attribute. The `<tx:advice>` definition creates an instance of a `ITransactionAttributeSource` during parsing time. Switching to use `<tx:advice>` instead of `<tx:attribute-driven/>` in the example would look like the following

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="Save*" />
    <tx:method name="Delete*" />
  </tx:attributes>
</tx:advice>
```

This says that all methods that start with `Save` and `Delete` would have associated with them the default settings of transaction metadata. These default values are listed below..

Here is an example using other elements of the `<tx:method/>` definition

```
<!-- the transactional advice (i.e. what 'happens'; see the <aop:advisor/> object below) -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="Get*" read-only="true" />
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
```

The `<tx:advice/>` definition reads as “... all methods on starting with 'Get' are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics”. The 'transaction-manager' attribute of the `<tx:advice/>` tag is set to the name of the `PlatformTransactionManager` object that is going to actually drive the transactions (in this case the 'transactionManager' object).

You can also use the AOP namespace `<aop:advisor>` element to tie together a pointcut and the above defined advice as shown below.

```
<object id="serviceOperation" type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut, Spring.Aop">
  <property name="pattern" value="Spring.TxQuickStart.Services.*" />
</object>

<aop:config>

  <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice" />

</aop:config>
```

```
</aop:config>
```

This is assuming that the service layer class, `TestObjectManager`, in the namespace `Spring.TxQuickStart.Services`. The `<aop:config>` definition ensures that the transactional advice defined by the `'txAdvice'` object actually executes at the appropriate points in the program. First we define a pointcut that matches any operation defined on classes in the `Spring.TxQuickStart.Services` (you can be more selective in your regular expression). Then we associate the pointcut with the `'txAdvice'` using an advisor. In the example, the result indicates that at the execution of a `'SaveTwoTestObjects'` and `'DeleteTwoTestObject'`, the advice defined by `'txAdvice'` will be run.

The various transactional settings that can be specified using the `<tx:advice/>` tag. The default `<tx:advice/>` settings are listed below and are the same as when you use the Transaction attribute.

- The propagation setting is `TransactionPropagation.Required`
- The isolation level is `IsolationLevel.ReadCommitted`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
- `EnterpriseServicesInteropOption` (.NET 2.0 only with `TxScopeTransactionManager`) - options between transaction created with `System.Transactions` and transactions created through COM+
- Any exception will trigger rollback.

These default settings can be changed; the various attributes of the `<tx:method/>` tags that are nested within `<tx:advice/>` and `<tx:attributes/>` tags are summarized below:

Table 17.2. `<tx:method/>` settings

Attribute	Required?	Default	Description
name	Yes		The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, <code>'Get*'</code> , <code>'Handle*'</code> , <code>'On*Event'</code> , and so forth.
propagation	No	Required	The transaction propagation behavior
isolation	No	ReadCommitted	The transaction isolation level
timeout	No	-1	The transaction timeout value (in seconds)

Attribute	Required?	Default	Description
read-only	No	false	Is this transaction read-only?
EnterpriseServicesInteropOption	No	None	Interoperability options with COM+ transactions. (.NET 2.0 and TxScopeTransactionManager only)
rollback-for	No		The Exception(s) that will trigger rollback; comma-delimited. For example, 'MyProduct.MyBusinessException, V'
no-rollback-for	No		The Exception(s) that will <i>not</i> trigger rollback; comma-delimited. For example, 'MyProduct.MyBusinessException, V'

17.5.4. Transaction attribute settings

The Transaction attribute is metadata that specifies that a class or method must have transactional semantics. The default Transaction attribute settings are

- The propagation setting is `TransactionPropagation.Required`
- The isolation level is `IsolationLevel.ReadCommitted`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
- `EnterpriseServicesInteropOption` (.NET 2.0 only with `TxScopeTransactionManager`) - options between transaction created with `System.Transactions` and transactions created through COM+
- Any exception will trigger rollback.

The default settings can, of course, be changed; the various properties of the Transaction attribute are summarised in the following table

Table 17.3. Transaction attribute properties

Property	Type	Description
TransactionPropagation	enumeration, <code>Spring.Transaction.TransactionPropagation</code>	optional propagation setting. Required, Supports, Mandatory, RequiresNew, NotSupported, Never, Nested
Isolation	<code>System.Data.IsolationLevel</code>	optional isolation level

ReadOnly	boolean	read/write vs. read-only transaction
EnterpriseServicesInteropOption	enumeration System.Transactions.EnterpriseServicesInteropOption	Options for interoperability with COM and TxScopeTransactionManager only (.NET 2.0 and later only)
Timeout	int (in seconds granularity)	the transaction timeout
RollbackFor	an array of <code>Type</code> objects	an optional array of exception classes that must cause rollback
NoRollbackFor	an array of <code>Type</code> objects	an optional array of exception classes that must not cause rollback

Note that setting the `TransactionPropagation` to `Nested` will throw a `NestedTransactionNotSupportedException` in a case where an actual nested transaction occurs, i.e. not in the case of applying the `Nested` propagation but in fact no nested calls are made. This will be fixed for the Spring 1.2 release for `SqlServer` and `Oracle` which support nested transactions. Also note, that changing of isolation levels on a per-method basis is also scheduled for the Spring 1.2 release since it requires detailed command text metadata for each dbprovider. Please check the forums for news on when this feature will be introduced into the nightly builds.

If you specify an exception type for 'NoRollbackFor' the action taken is to commit the work that has been done in the database up to the point where the exception occurred. The exception is still propagated out to the calling code.

The `ReadOnly` boolean is a hint to the data access technology to enable read-only optimizations. This currently has no effect in Spring's ADO.NET framework. If you would like to enable read-only optimizations in ADO.NET this is generally done via the 'Mode=Read' or 'Mode=Read-Only' options in the connection string. Check your database provider for more information. In the case of `NHibernate` the flush mode is set to `Never` when a new `Session` is created for the transaction.

Throwing exceptions to indicate failure and assuming success is an easier and less invasive programming model than performing the same task Programatically - `ContextUtil.MyTransactionVote` or `TransactionScope.Complete`. The rollback options are a means to influence the outcome of the transaction based on the exception type which adds an extra degree of flexibility.

Having any exception trigger a rollback has similar behavior as applying the `AutoComplete` attribute available when using .NET Enterprise Services. The difference with `AutoComplete` is that using `AutoComplete` is also coupled to the lifetime of the `ServiceComponent` since it sets `ContextUtil.DeactivateOnReturn` to `true`. For a stateless DAO layer this is not an issue but it could be in other scenarios. Spring's transactional aspect does not affect the lifetime of your object.

17.5.5. Declarative Transactions using AutoProxy

if you choose not to use the transaction namespace for declarative transaction management then you can use 'lower level' object definitions to configure declarative transactions. The use of Spring's autoproxy functionality defines criteria to select a collection of objects to create a transactional AOP proxy. There are two `AutoProxy` classes that you can use, `ObjectNameAutoProxyCreator` and `DefaultAdvisorAutoProxyCreator`. If you are using the new transaction namespace support you do not need to configure these objects as a `DefaultAdvisorAutoProxyCreator` is created 'under the covers' while parsing the transaction namespace elements

17.5.5.1. Creating transactional proxies with `ObjectNameAutoProxyCreator`

The `ObjectNameAutoProxyCreator` is useful when you would like to create transactional proxies for many objects. The definitions for the `TransactionInterceptor` and associated attributes is done once. When you add new objects to your configuration file that need to be proxies you only need to add them to the list of object referenced in the `ObjectNameAutoProxyCreator`. Here is an example showing its use. Look in the section that use `ProxyFactoryObject` for the declaration of the `transactionInterceptor`.

```
<object name="autoProxyCreator"
        type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator, Spring.Aop">

    <property name="InterceptorNames" value="transactionInterceptor"/>
    <property name="ObjectNames">
        <list>
            <idref local="testObjectManager"/>
        </list>
    </property>
</object>
```

17.5.5.2. Creating transactional proxies with `DefaultAdvisorAutoProxyCreator`

This is not longer a common way to configure declarative transactions but is discussed in the "Classic Spring" appendix here.

17.6. Programmatic transaction management

Spring provides two means of programmatic transaction management:

- Using the `TransactionTemplate`
- Using a `IPlatformTransactionManager` implementation directly

These are located in the `Spring.Transaction.Support` namespace. If you are going to use programmatic transaction management, the Spring team generally recommends the first approach (i.e. Using the `TransactionTemplate`)

17.6.1. Using the `TransactionTemplate`

The `TransactionTemplate` adopts the same approach as other Spring templates such as `AdoTemplate` and `HibernateTemplate`. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do. Granted that the using construct of `System.Transaction` alleviates much of this. One key difference with the approach taken with the `TransactionTemplate` is that a commit is assumed - throwing an exception triggers a rollback instead of using the `TransactionScope` API to commit or rollback. This also allows for the use of rollback rules, that is a commit can still occur for exceptions of certain types.



Note

As you will immediately see in the examples that follow, using the `TransactionTemplate` absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you will have to make yourself.

Application code that must execute in a transaction context looks like this. You, as an application developer, will write a `ITransactionCallback` implementation (typically expressed as an anonymous delegate) that will contain

all of the code that you need to have execute in the context of a transaction. You will then pass an instance of your custom `ITransactionCallback` to the `Execute(..)` method exposed on the `TransactionTemplate`. Note that the `ITransactionCallback` can be used to return a value:

```
public class SimpleService : IService
{
    private TransactionTemplate transactionTemplate;

    public SimpleService(IPlatformTransactionManager transactionManager)
    {
        AssertUtils.ArgumentNotNull(transactionManager, "transactionManager");
        transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public object SomeServiceMethod()
    {
        return tt.Execute(delegate {
            UpdateOperation(userId);
            return ResultOfUpdateOperation2();
        });
    }
}
```

This code example is specific to .NET 2.0 since it uses anonymous delegates, which provides a particularly elegant means to invoke a callback function as local variables can be referred to inside the delegate, i.e. `userId`. In this case the `ITransactionStatus` was not exposed in the delegate (delegate can infer the signature to use), but one could also obtain a reference to the `ITransactionStatus` instance and set the `RollbackOnly` property to trigger a rollback - or alternatively throw an exception. This is shown below

```
tt.Execute(delegate(ITransactionStatus status)
{
    try {
        UpdateOperation1();
        UpdateOperation2();
    } catch (SomeBusinessException ex) {
        status.RollbackOnly = true;
    }
    return null;
});
```

If you are using .NET 1.1 then you should provide a normal delegate reference or an instance of a class that implements the `ITransactionCallback` interface. This is shown below

```
tt.Execute(new TransactionRollbackTxCallback(amount));

public class TransactionRollbackTxCallback : ITransactionCallback
{
    private decimal amount;

    public TransactionRollbackTxCallback(decimal amount)
    {
        this.amount = amount
    }

    public object DoInTransaction(ITransactionStatus status)
    {
        adoTemplate.ExecuteNonQuery(CommandType.Text, "insert into dbo.Debits (DebitAmount) VALUES (@amount)", "amount"
        // decide you need to rollback...
        status.RollbackOnly = true;
        return null;
    }
}
```

Application classes wishing to use the `TransactionTemplate` must have access to a `IPlatformTransactionManager` (which will typically be supplied to the class via dependency injection). It is easy to unit test such classes with a mock or stub `IPlatformTransactionManager`.

17.6.1.1. Specifying transaction settings

Transaction settings such as the propagation mode, the isolation level, the timeout, and so forth can be set on the `TransactionTemplate` either programmatically or in configuration. `TransactionTemplate` instances by default have the default transactional settings. Find below an example of programmatically customizing the transactional settings for a specific `TransactionTemplate`.

```
public class SimpleService : IService
{
    private TransactionTemplate transactionTemplate;

    public SimpleService(IPlatformTransactionManager transactionManager)
    {
        AssertUtils.ArgumentNotNull(transactionManager, "transactionManager");
        transactionTemplate = new TransactionTemplate(transactionManager);

        // the transaction settings can be set here explicitly if so desired

        transactionTemplate.TransactionIsolationLevel = IsolationLevel.ReadUncommitted;
        transactionTemplate.TransactionTimeout = 30;

        // and so forth...
    }

    . . .
}
```

Find below an example of defining a `TransactionTemplate` with some custom transactional settings, using Spring XML configuration. The 'sharedTransactionTemplate' can then be injected into as many services as are required.

```
<object id="sharedTransactionTemplate"
    type="Spring.Transaction.Support.TransactionTemplate, Sprng.Data">
    <property name="TransactionIsolationLevel" value="IsolationLevel.ReadUncommitted"/>
    <property name="TransactionTimeout" value="30"/>
</object>
```

Finally, instances of the `TransactionTemplate` class are threadsafe, in that instances do not maintain any conversational state. `TransactionTemplate` instances do however maintain configuration state, so while a number of classes may choose to share a single instance of a `TransactionTemplate`, if a class needed to use a `TransactionTemplate` with different settings (for example, a different isolation level), then two distinct `TransactionTemplate` instances would need to be created and used.

17.6.2. Using the PlatformTransactionManager

You can also use the `PlatformTransactionManager` directly to manage your transaction. Simply pass the implementation of the `PlatformTransactionManager` you're using to your object via a object reference through standard Dependency Injection techniques. Then, using the `TransactionDefinition` and `ITransactionStatus` objects, you can initiate transactions, rollback and commit.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.PropagationBehavior = TransactionPropagation.Required;

ITransactionStatus status = transactionManager.GetTransaction(def);

try
{
    // execute your business logic here
} catch (Exception e)
{
    transactionManager.Rollback(status);
    throw;
}
```

```
transactionManager.Commit(status);
```

Note that a corresponding 'using TransactionManagerScope' class can be modeled to get similar API usage to System.Transactions TransactionScope.

17.7. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the TransactionTemplate may be a good approach. On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure in Spring.

17.8. Transaction lifecycle and status information

You can query the status of the current Spring managed transaction with the class TransactionSynchronizationManager. Typical application code should not need to rely on using this class but in some cases it is convenient to receive events around the lifecycle of the transaction, i.e. before committing, after committing. TransactionSynchronizationManager provides a method to register a callback object that is informed on all significant stages in the transaction lifecycle. Note that you can register for lifecycle call back information for any of the transaction managers you use, be it NHibernate or local ADO.NET transactions.

The method to register a callback with the TransactionSynchronizationManager is

```
public static void RegisterSynchronization( ITransactionSynchronization synchronization )
```

Please refer to the SDK docs for information on other methods in this class.

The ITransactionSynchronization interface is

```
public interface ITransactionSynchronization
{
    // Typically used by Spring resource management code
    void Suspend();
    void Resume();

    // Transaction lifecycle callback methods
    // Typically used by Spring resource management code but maybe useful in certain cases to application code
    void BeforeCommit( bool readOnly );
    void AfterCommit();
    void BeforeCompletion();
    void AfterCompletion( TransactionSynchronizationStatus status );
}
```

The TransactionSynchronizationStatus is an enum with the values Committed, Rolledback, and Unknown.

Chapter 18. DAO support

18.1. Introduction

Spring promotes the use of data access interfaces in your application architecture. These interfaces encapsulate the storage and retrieval of data and objects specific to your business domain without reference to a specific persistence API. Within a layered architecture, the service layer is typically responsible for coordinating responses to a particular business request and it delegates any persistence related activities to objects that implement these data access interfaces. These objects are commonly referred to as DAOs (Data Access Objects) and the architectural layer as a DAL (Data Access Layer).

The benefits of using DAOs in your application are increased portability across persistence technology and ease of testing. Testing is more easily facilitated because a mock or stub implementation of the data access interface can be easily created in a NUnit test so that service layer functionality can be tested without any dependency on the database. This is beneficial because tests that rely on the database are usually hard to set up and tear down and also are impractical for testing exceptional behavior.

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like ADO.NET and NHibernate in a standardized way. Spring provides two central pieces of functionality to meet this goal. The first is providing a common exception hierarchy across providers and the second is providing base DAOs classes that raise the level of abstraction when performing common ADO.NET operations. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

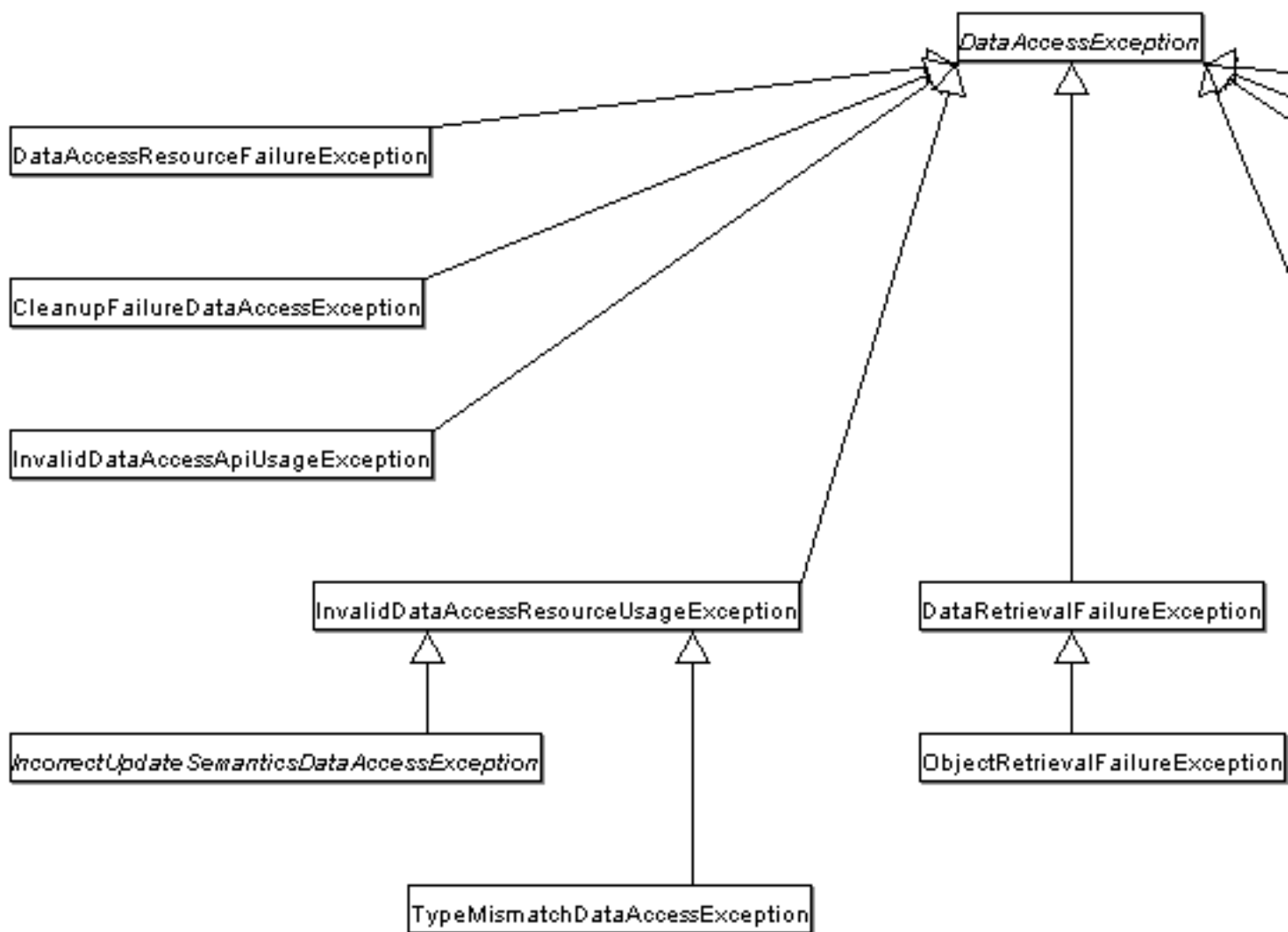
18.2. Consistent exception hierarchy

Database exceptions in the ADO.NET API are not consistent across providers. The .NET 1.1 BCL did not provide a common base class for ADO.NET exceptions. As such you were required to handle exceptions specific to each provider such as `System.Data.SqlClient.SqlException` or `System.Data.OracleClient.OracleException`. The .NET 2.0 BCL improved in this regard by introducing a common base class for exceptions, `System.Data.Common.DbException`. However the common `DbException` is not very portable either as it provides a vendor specific error code as the underlying piece of information as to what went wrong. This error code is different across providers for the same conceptual error, such as a violation of data integrity or providing bad SQL grammar.

To promote writing portable and descriptive exception handling code Spring provides a convenient translation from technology specific exceptions like `System.Data.SqlClient.SqlException` or `System.Data.OracleClient.OracleException` to its own exception hierarchy with the `Spring.Dao.DataAccessException` as the root exception. These exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong.

In addition to exceptions from ADO.NET providers, Spring can also wrap NHibernate-specific exceptions.. This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without boilerplate using `try` or `catch` and `throw` blocks, and exception declarations. As mentioned above, ADO.NET exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with ADO.NET within a consistent programming model. The above holds true for the various template-based versions of the ORM access framework.

The exception hierarchy that Spring uses is outlined in the following image:



(Please note that the class hierarchy detailed in the above image shows only a subset of the whole, rich, `DataAccessException` hierarchy.)

The exception translation functionality is in the namespace `Spring.Data.Support` and is based on the interface `IAdoExceptionTranslator` shown below.

```

public interface IAdoExceptionTranslator
{
    DataAccessException Translate( string task, string sql, Exception exception );
}

```

The arguments to the translator are a task string providing a description of the task being attempted, the SQL query or update that caused the problem, and the 'raw' exception thrown by the ADO.NET data provider. The additional task and SQL arguments allow for very readable and clear error messages to be created when an exception occurs.

A default implementation, `ErrorCodeExceptionTranslator`, is provided that uses the error codes defined for each data provider in the file `dbproviders.xml`. Refer to this file, an embedded resource in the `Spring.Data` assembly, for the exact mappings of error codes to Spring `DataAccessExceptions`.

A common need is to modify the error codes that are map onto the exception hierarchy. There are several ways to accomplish this task.

One approach is to override the error codes that are defined in `assembly://Spring.Data/Spring.Data.Common/dbproviders.xml`. By default, the `DbProviderFactory` will look for additional metadata for the IoC container it uses internally to define and manage the `DbProviders` in a file named `dbProviders.xml` located in the root

runtime directory. (You can change this location, see the documentation on `DbProvider` for more information.) This is a standard Spring application context so all features, such as `ObjectFactoryPostProcessors` are available and will be automatically applied. Defining a `PropertyOverrideConfigurer` in this additional configuration file will allow for you to override specific property values defined in the embedded resource file. As an example, the additional `dbProviders.xml` file shown below will add the error code 2601 to the list of error codes that map to a `DataIntegrityViolationException`.

```
<objects xmlns='http://www.springframework.net'>

  <alias name='SqlServer-2.0' alias='SqlServer2005' />

  <object name="appConfigPropertyOverride" type="Spring.Objects.Factory.Config.PropertyOverrideConfigurer, Spring.Core">
    <property name="Properties">
      <name-values>
        <add key="SqlServer2005.DbMetadata.ErrorCodes.DataIntegrityViolationCodes"
          value="544,2601,2627,8114,8115" />
      </name-values>
    </property>
  </object>

</objects>
```

The reason to define the alias is that `PropertyOverrideConfigurer` assumes a period (.) as the separator to pick out the object name but the names of the objects in `dbProviders.xml` have periods in them (i.e. `SqlServer-2.0` or `System.Data.SqlClient`). Creating an alias that has no periods in the name is a workaround.

Another way to customize the mappings of error codes to exceptions is to subclass `ErrorCodeExceptionTranslator` and override the method, `DataAdapterException.TranslateException(string task, string sql, string errorCode, Exception exception)`. This will be called before referencing the metadata to perform exception translation. The vendor specific error code provided as a method argument has already been parsed out of the raw ADO.NET exception. If you create your own specific subclass, then you should set the property `ExceptionTranslator` on `AdoTemplate` and `HibernateTemplate/HibernateTransactionManager` to refer to your custom implementation (unless you are using autowiring).

The third way is to write an implementation of `IAdoExceptionTranslator` and set the property `FallbackTranslator` on `ErrorCodeExceptionTranslator`. In this case you are responsible for parsing out the vendor specific error code from the raw ADO.NET exception. As with the case of subclassing `ErrorCodeExceptionTranslator`, you will need to refer to this custom exception translator when using `AdoTemplate` or `HibernateTemplate/HibernateTransactionManager`.

The ordering of the exception translation processing is as follows. The method `TranslateException` is called first, then the standard exception translation logic, then the `FallbackTranslator`.

Note that you can use this API directly in your own Spring independent data layer. If you are using Spring's ADO.NET abstraction class, `AdoTemplate`, or `HibernateTemplate`, the converted exceptions will be thrown automatically. Somewhere in between these two cases is using Spring's declarative transaction management features in .NET 2.0 with the raw ADO.NET APIs and using `IAdoExceptionTranslator` in your exception handling layer (which might be implemented in AOP using Spring's exception translation aspect).

Some of the more common data access exceptions are described here. Please refer to the API documentation for more details.

Table 18.1. Common DataAccessExceptions

Exception	Description
<code>BadSqlGrammarException</code>	Exception thrown when SQL specified is invalid.

Exception	Description
<code>DataIntegrityViolationException</code>	Exception thrown when an attempt to insert or update data results in violation of an integrity constraint. For example, inserting a duplicate key.
<code>PermissionDeniedDataAccessException</code>	Exception thrown when the underlying resource denied a permission to access a specific element, such as a specific database table.
<code>DataAccessResourceFailureException</code>	Exception thrown when a resource fails completely, for example, if we can't connect to a database.
<code>ConcurrencyFailureException</code>	Exception thrown when a concurrency error occurs. <code>OptimisticLockingFailureException</code> and <code>PessimisticLockingFailureException</code> are subclasses. This is a useful exception to catch and to retry the transaction again. See Spring's Retry Aspect for an AOP based solution.
<code>OptimisticLockingFailureException</code>	Exception thrown when there an optimistic locking failure occurs. The subclass <code>ObjectOptimisticLockingFailureException</code> can be used to examine the Type and the ID of the object that failed the optimistic locking.
<code>PessimisticLockingFailure</code>	Exception thrown when a pessimistic locking failure occurs. Subclasses of this exception are <code>CannotAcquireLockException</code> , <code>CannotSerializeTransactionException</code> , and <code>DeadlockLoserDataAccessException</code> .
<code>CannotAcquireLockException</code>	Exception thrown when a lock can not be acquired, for example during an update, i.e a select for update
<code>CannotSerializeTransactionException</code>	Exception thrown when a transaction can not be serialized.

18.3. Consistent abstract classes for DAO support

To make it easier to work with a variety of data access technologies such as ADO.NET, NHibernate, and iBatis.NET in a consistent way, Spring provides a set of abstract DAO classes that one can extend. These abstract classes have methods for providing the data source and any other configuration settings that are specific to the technology one is currently using.

DAO support classes:

- `AdoDaoSupport` - super class for ADO.NET data access objects. Requires a `DbProvider` to be provided; in turn, this class provides a `AdoTemplate` instance initialized from the supplied `DbProvider` to subclasses. See the documentation for `AdoTemplate` for more information.
- `HibernateDaoSupport` - super class for NHibernate data access objects. Requires a `ISessionFactory` to be provided; in turn, this class provides a `HibernateTemplate` instance initialized from the supplied

`SessionFactory` to subclasses. Can alternatively be initialized directly via a `HibernateTemplate`, to reuse the latter's settings like `SessionFactory`, flush mode, exception translator, etc. This is contained in a download separate from the main Spring.NET distribution.

Chapter 19. DbProvider

19.1. Introduction

Spring provides a generic factory for creating ADO.NET API artifacts such as `IDbConnection` and `IDbCommand`. The factory API is very similar to the one introduced in .NET 2.0 but adds extra metadata needed by Spring to support features provided by its DAO/ADO.NET framework such as error code translation to a DAO exception hierarchy. The factory itself is configured by using a standard Spring XML based configuration file though it is unlikely you will need to modify those settings yourself, you only need be concerned with using the factory. Out of the box several popular databases are supported and an extension mechanism is available for defining new database providers or modifying existing ones. A custom database namespace for configuration aids in making terse XML based declarations of Spring's database objects you wish to use.

The downside of Spring's factory as compared to the one in .NET 2.0 is that the types returned are lower level interfaces and not the abstract base classes in `System.Data.Common`. However, there are still 'holes' in the current .NET 2.0 provider classes that are 'plugged' with Spring's provider implementation. One of the most prominent is the that the top level `DbException` exposes the `HRESULT` of the remote procedure call, which is not what you are commonly looking for when things go wrong. As such Spring's provider factory exposes the vendor sql error code and also maps that error code onto a consistent data access exception hierarchy. This makes writing portable exception handlers much easier. In addition, the `DbParameter` class doesn't provide the most common convenient methods you would expect as when using say the `SqlServer` provider. If you need to access the BCL provider abstraction, you still can through Spring's provider class. Furthermore, a small wrapper around the standard BCL provider abstraction allows for integration with Spring's transaction management facilities, allowing you to create a `DbCommand` with its connection and transaction properties already set based on the transaction calling context.

19.2. IDbProvider and DbProviderFactory

The `IDbProvider` API is shown below and should look familiar to anyone using .NET 2.0 data providers. Note that Spring's `DbProvider` abstraction can be used on .NET 1.1 in addition to .NET 2.0

```
public interface IDbProvider
{
    IDbCommand CreateCommand();

    object CreateCommandBuilder();

    IDbConnection CreateConnection();

    IDbDataAdapter CreateDataAdapter();

    IDbDataParameter CreateParameter();

    string CreateParameterName(string name);

    string CreateParameterNameForCollection(string name);

    IDbMetadata DbMetadata
    {
        get;
    }

    string ConnectionString
    {
        set;
        get;
    }
}
```

```

string ExtractError(Exception e);

bool IsDataAccessException(Exception e);
}

```

ExtractError is used to return an error string for translation into a DAO exception. On .NET 1.1 the method IsDataAccessException is used to determine if the thrown exception is related to data access since in .NET 1.1 there isn't a common base class for database exceptions. CreateParameterName is used to create the string for parameters used in a CommandText object while CreateParameterNameForCollection is used to create the string for a IDataParameter.ParameterName, typically contained inside a IDataParameterCollection.

The class DbProviderFactory creates IDbProvider instances given a provider name. The connection string property will be used to set the IDbConnection returned by the factory if present. The provider names, and corresponding database, currently configured are listed below.

- SqlServer-1.1 - Microsoft SQL Server, provider V1.0.5000.0 in framework .NET V1.1
- SqlServer-2.0 (aliased to System.Data.SqlClient) - Microsoft SQL Server, provider V2.0.0.0 in framework .NET V2.0
- SqlServerCe-3.1 - Microsoft SQL Server Compact Edition, provider V9.0.242.0
- SqlServerCe-3.5.1 (aliased to System.Data.SqlServerCe) - Microsoft SQL Server Compact Edition, provider V3.5.1.0
- OleDb-1.1 - OleDb, provider V1.0.5000.0 in framework .NET V1.1
- OleDb-2.0 (aliased to System.Data.OleDb) - OleDb, provider V2.0.0.0 in framework .NET V2.0
- OracleClient-2.0 (aliased to System.Data.OracleClient) - Oracle, Microsoft provider V2.0.0.0
- OracleODP-2.0 (aliased to System.DataAccess.Client) - Oracle, Oracle provider V2.102.2.20 (Oracle 10g)
- OracleODP-11-2.0 - Oracle, Oracle provider V2.111.7.20 (Oracle 11g)
- MySql - MySQL, MySQL provider 1.0.10.1
- MySql-1.0.9 - MySQL, MySQL provider 1.0.9
- MySql-5.0 - MySQL, MySQL provider 5.0.7.0
- MySql-5.0.8.1 - MySQL, MySQL provider 5.0.8.1
- MySql-5.1 - MySQL, MySQL provider 5.1.2.2
- MySql-5.1.4 - MySQL, MySQL provider 5.1.2.2
- MySql-5.2.3 - MySQL, MySQL provider 5.2.3.0
- MySql-6.1.3 (aliased to MySql.Data.MySqlClient) - MySQL, MySQL provider 6.1.3.0
- Npgsql-1.0 - Postgresql provider 1.0.0.0 (and 1.0.0.1 - were build with same version info)
- Npgsql-2.0-beta1 - Postgresql provider 1.98.1.0 beta 1

- Npgsql-2.0 - Postgresql provider 2.0.0.0
- DB2-9.0.0-1.1 - IBM DB2 Data Provider 9.0.0 for .NET Framework 1.1
- DB2-9.0.0-2.0 (aliased to IBM.Data.DB2) - IBM DB2 Data Provider 9.0.0 for .NET Framework 2.0
- DB2-9.1.0-1.1 - IBM DB2 Data Provider 9.1.0 for .NET Framework 1.1
- DB2-9.1.0.2 (aliased to IBM.Data.DB2.9.1.0) - IBM DB2 Data Provider 9.1.0 for .NET Framework 2.0
- iDB2-10.0.0.0 - IBM iSeries DB2 Data Provider 10.0.0.0 for .NET Framework 2.0
- SQLite-1.0.43 - SQLite provider 1.0.43 for .NET Framework 2.0
- SQLite-1.0.44 - SQLite provider 1.0.44 for .NET Framework 2.0
- SQLite-1.0.47 - SQLite provider 1.0.47 for .NET Framework 2.0
- SQLite-1.0.56 - SQLite provider 1.0.56 for .NET Framework 2.0
- SQLite-1.0.65 (aliased to System.Data.SQLite) - SQLite provider 1.0.65 for .NET Framework 2.0



Note

The default parameter prefix used in SQLite is : and not @, please write your SQL accordingly or define a provider definition for SQLite.

- Firebird-2.1 (aliased to Firebird-2.1) - Firebird Server, provider V2.1.0.0 in framework .NET V2.0
- SybaseAse-12 - Sybase ASE provider for ASE 12.x
- SybaseAse-15 - Sybase ASE provider for ASE 15.x
- SybaseAse-AdoNet2 - Sybase ADO.NET 2.0 provider for ASE 12.x and 15.x
- Odbc-1.1 - ODBC provider V1.0.5000.0 in framework .NET V1.1
- Odbc-2.0 - ODBC provider V2.0.0.0 in framework .NET V2
- Cache-2.0.0.1 (aliased to InterSystems.Data.CacheClient) - Cache provider Version 2.0.0.1 in framework .NET V2



Note

If your exact version of the database provider is not listed, you can pick the general provider name, i.e. `MySql.Data.MySqlClient`, and then perform an assembly redirect in `App.config`. This will often be sufficient to upgrade to newer versions. As shown below

```
<runtime>

  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

    <dependentAssembly>
      <assemblyIdentity name="Npgsql"
        publicKeyToken="5d8b90d52f46fda7"
        culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-65535.65535.65535.65535"
        newVersion="2.0.0.0" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```



```

    </dependentAssembly>

    </assemblyBinding>

</runtime>

```

An example using DbProviderFactory is shown below

```
IDbProvider dbProvider = DbProviderFactory.GetDbProvider("System.Data.SqlClient");
```

The default definitions of the providers are contained in the assembly resource `assembly://Spring.Data/Spring.Data.Common/dbproviders.xml`. Future additions to round out the database coverage are forthcoming. The current crude mechanism to add additional providers, or to apply any standard Spring `IApplicationContext` functionality, such as applying AOP advice, is to set the public static property `DBPROVIDER_ADDITIONAL_RESOURCE_NAME` in `DbProviderFactory` to a Spring resource location. The default value is `file://dbProviders.xml`. (That isn't a typo, there is a difference in case with the name of the embedded resource). This crude mechanism will eventually be replaced with one based on a custom configuration section in `App.config/Web.config`.

It may happen that the version number of an assembly you have downloaded is different than the one listed above. If it is a point release, i.e. the API hasn't changed in anyway that is material to your application, you should add an assembly redirect of the form shown below.

```

<dependentAssembly>
  <assemblyIdentity name="MySQL.Data"
    publicKeyToken="c5687fc88969c44d"
    culture="neutral"/>
  <bindingRedirect oldVersion="0.0.0.0-65535.65535.65535.65535"
    newVersion="1.0.10.1"/>
</dependentAssembly>

```

This redirects any reference to an older version of the assembly `MySQL.Data` to the version `1.0.10.1`.

19.3. XML based configuration

Creating a DbProvider in Spring's XML configuration file is shown below in the typical case of using it to specify the DbProvider property on an `AdoTemplate`.

```

<objects xmlns='http://www.springframework.net'
  xmlns:db="http://www.springframework.net/database">

  <db:provider id="DbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=(local);Database=Spring;User ID=springqa;Password=springqa;Trusted_Connection=False"/>

  <object id="adoTemplate" type="Spring.Data.Core.AdoTemplate, Spring.Data">
    <property name="DbProvider" ref="DbProvider"/>
  </object>

</objects>

```

A custom namespace should be registered in the main application configuration file to use this syntax. This configuration, only for the parsers, is shown below. Additional section handlers are needed to specify the rest of the Spring configuration locations as described in previous chapters.

```

<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>

```

```

<spring>
  <parsers>
    <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
  </parsers>
</spring>

</configuration>

```

19.4. Connection String management

There are a few options available to help manage your connection strings.

The first option is to leverage the Spring property replacement functionality, as described in Section 5.9.2.1, “Example: The PropertyPlaceholderConfigurer”. This lets you insert variable names as placeholders for values in a Spring configuration file. In the following example specific parts of a connection string have been parameterized but you can also use a variable to set the entire connection string.

An example of such a setting is shown below

```

<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name='context' type='Spring.Context.Support.ContextHandler, Spring.Core' />
    </sectionGroup>

    <section name="databaseSettings" type="System.Configuration.NameValueSectionHandler, System, Version=1.0.5000.0, Culture=
  </configSections>

  <spring>
    <context>
      <resource uri="Aspects.xml" />
      <resource uri="Services.xml" />
      <resource uri="Dao.xml" />
    </context>
  </spring>

  <!-- These properties are referenced in Dao.xml -->
  <databaseSettings>
    <add key="db.datasource" value="(local)" />
    <add key="db.user" value="springqa" />
    <add key="db.password" value="springqa" />
    <add key="db.database" value="Northwind" />
  </databaseSettings>

</configuration>

```

Where Dao.xml has a connection string as shown below

```

<objects xmlns='http://www.springframework.net'
  xmlns:db="http://www.springframework.net/database">

  <db:provider id="DbProvider"
    provider="System.Data.SqlClient"
    connectionString="${db.datasource};Database=${db.database};User ID=${db.user};Password=${db.password};Trusted_Connection=
  </db:provider>

  <object id="adoTemplate" type="Spring.Data.Core.AdoTemplate, Spring.Data">
    <property name="DbProvider" ref="DbProvider"/>
  </object>

  <!-- configuration of what values to substitute for ${ } variables listed above -->
  <object name="appConfigPropertyHolder"
    type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">
    <property name="configSections" value="DatabaseConfiguration"/>
  </object>

</objects>

```

Please refer to the Section Section 5.9.2.1, “Example: The PropertyPlaceholderConfigurer” for more information.

19.5. Additional IDbProvider implementations

Spring provides some convenient implementations of the IDbProvider interface that add additional behavior on top of the standard implementation.



Note

These provider implementations do not take into account usage with NHibernate. NHibernate scopes a SessionFactory, where second level caching is managed, to each connection. This forum thread [<http://forum.springframework.net/showthread.php?t=4462>], contains an implementation of the class LocalDelegatingSessionFactoryObject that will create multiple SessionFactories for each database connection.

19.5.1. UserCredentialsDbProvider

This UserCredentialsDbProvider will allow you to change the username and password of a database connection at runtime. The API contains the properties Username and Password which are used as the default strings representing the user and password in the connection string. You can then change the value of these properties in the connection string by calling the method SetCredentialsForCurrentThread and fall back to the default values by calling the method RemoveCredentialsFromCurrentThread. You call the SetCredentialsForCurrentThread method at runtime, before any data access occurs, to determine which database user should be used for the current user-case. Which user to select is up to you. You may retrieve the user information from an HTTP session for example. Example configuration and usage is shown below

```
<object id="DbProvider" type="Spring.Data.Common.UserCredentialsDbProvider, Spring.Data">
  <property name="TargetDbProvider" ref="targetDbProvider"/>
  <property name="Username" value="User ID=defaultName"/>
  <property name="Password" value="Password=defaultPass"/>
</object>

<db:provider id="targetDbProvider" provider="SqlServer-2.0"
  connectionString="Data Source=MARKT60\SQL2005;Database=Spring;Trusted_Connection=False"/>
```

If you use dependency injection to configure a class with a property of the type IDbProvider, you will need to downcast to the subtype or you can change your class to have a property of the type UserCredentialsDbProvider instead of IDbProvider.

```
userCredentialsDbProvider.SetCredentialsForCurrentThread("User ID=springqa", "Password=springqa");
```

UserCredentialsDbProvider's has a base class, DelegatingDbProvider, and is intended for you to use in your own implementations that delegate calls to a target IDbProvider instance. This class is meant to be subclassed with subclasses overriding only those methods, such as CreateConnection(), that should not simply delegate to the target IDbProvider.

19.5.2. MultiDelegatingDbProvider

There are use-cases in which there will need to be a runtime selection of the database to connect to among many possible candidates. This is often the case where the same schema is installed in separate databases for different clients. The MultiDelegatingDbProvider implements the IDbProvider interface and provides an abstraction to the multiple databases and can be used in DAO layer such that the DAO layer is unaware of the switching between databases. MultiDelegatingDbProvider does its job by looking into thread local storage under the key dbProviderName. This storage location stores the name of the dbProvider that is to be used for processing the

request. `MultiDelegatingDbProvider` is configured using the dictionary property `TargetDbProviders`. The key of this dictionary contains the name of a `dbProvider` and its value is a `dbProvider` object. (You can also provide this dictionary as a constructor argument.)

During request processing, once you have determined which target `dbProvider` should be use, in this example `database1ProviderName`, you should execute the following code is you are using Spring 1.2 M1 or later

```
// Spring 1.2 M1 or later
LogicalThreadContext.SetData(MultiDelegatingDbProvider.CURRENT_DBPROVIDER_SLOTNAME, "database1ProviderName")
```

and the following ocde if you are using earlier versions

```
// Prior to Spring 1.2 M1
LogicalThreadContext.SetData("dbProviderName", "database1ProviderName")
```

and then call the data access layer.

Here is a sample configuration to build up an object definition for `MultiDelegatingDbProvider`.

```
<db:provider id="CreditAndDebitsDbProvider"
  provider="System.Data.SqlClient"
  connectionString="Data Source=MARKT60\SQL2005;Initial Catalog=CreditsAndDebits;User ID=springqa; Password=springqa"/>

<db:provider id="CreditDbProvider"
  provider="System.Data.SqlClient"
  connectionString="Data Source=MARKT60\SQL2005;Initial Catalog=Credits;User ID=springqa; Password=springqa"/>

<object id="dbProviderDictionary" type="Spring.Collections.SynchronizedHashtable, Spring.Core">
  <property name="['DbProvider1']" ref="CreditAndDebitsDbProvider"/>
  <property name="['DbProvider2']" ref="CreditDbProvider"/>
</object>

<object id="DbProvider" type="Spring.Data.MultiDelegatingDbProvider, Spring.Data">
  <property name="TargetDbProviders" ref="dbProviderDictionary"/>
</object>
```

Chapter 20. Data access using ADO.NET

20.1. Introduction

Spring provides an abstraction for data access via ADO.NET that provides the following benefits and features

- Consistent and comprehensive database provider interfaces for both .NET 1.1 and 2.0
- Integration with Spring's transaction management features.
- Template style use of DbCommand that removes the need to write typical ADO.NET boiler-plate code.
- 'One-liner' implementations for the most common database usage patterns lets you focus on the 'meat' of your ADO.NET code.
- Easy database parameter creation/management
- Provider independent exceptions with database error codes and higher level DAO exception hierarchy.
- Centralized resource management for connections, commands, data readers, etc.
- Simple DataReader to Object mapping framework.

This chapter is divided up into a number of sections that describe the major areas of functionality within Spring's ADO.NET support.

- Motivations - describes why one should consider using Spring's ADO.NET features as compared to using 'raw' ADO.NET API.
- Provider Abstraction - a quick overview of Spring's provider abstraction.
- Approaches to ADO.NET Data Access - Discusses the two styles of Spring's ADO.NET data access classes - template and object based.
- Introduction to AdoTemplate - Introduction to the design and core methods of the central class in Spring's ADO.NET support.
- Exception Translation - Describes the features of Spring's data access exceptions
- Parameter Management - Convenience classes and methods for easy parameter management.
- Custom IDataReader implementations - Strategy for providing custom implementations of IDataReader. This can be used to centralized and transparently map DBNull values to CLR types when accessing an IDataReader or to provide extended mapping functionality in sub-interfaces.
- Basic data access operations - Usage of AdoTemplate for IDbCommand 'ExecuteScalar' and 'ExecuteNonQuery' functionality
- Queries and Lightweight Object Mapping - Using AdoTemplate to map result sets into objects
- DataSet and DataTable operations - Using AdoTemplate with DataSets and DataTables
- Modeling ADO.NET operations as .NET objects - An object-oriented approach to data access operations.

20.2. Motivations

There are a variety of motivations to create a higher level ADO.NET persistence API.

Encapsulation of common 'boiler plate' tasks when coding directly against the ADO.NET API. For example here is a list of the tasks typically required to be coded for processing a result set query. Note that the code needed when using Spring's ADO.NET framework is in italics.

1. Define connection parameters
2. Open the connection
3. *Specify the command type and text*
4. Prepare and execute the statement
5. Set up the loop to iterate through the results (if any)
6. *Do the work for each iteration*
7. Process any exception
8. Display or rollback on warnings
9. Handle transactions
10. Close the connection

Spring takes care of the low-level tasks and lets you focus on specifying the SQL and doing the real work of extracting data. This standard boiler plate pattern is encapsulated in a class, `AdoTemplate`. The name 'Template' is used because if you look at the typical code workflow for the above listing, you would essentially like to 'template' it, that is stick in the code that is doing the real work in the midst of the resource, transaction, exception management.

Another very important motivation is to provide an easy means to group multiple ADO.NET operations within a single transaction while at the same time adhering to a DAO style design in which transactions are initiated outside the DAOs, typically in a business service layer. Using the 'raw' ADO.NET API to implement this design often results in explicitly passing around of a Transaction/Connection pair to DAO objects. This infrastructure task distracts from the main database task at hand and is frequently done in an ad-hoc manner. Integrating with Spring's transaction management features provides an elegant means to achieve this common design goal. There are many other benefits to integration with Spring's transaction management features, see Chapter 17, *Transaction management* for more information.

Provider Independent Code: In .NET 1.1 writing provider independent code was difficult for a variety of reasons. The most prominent was the lack of a central factory for creating interface based references to the core ADO.NET classes such as `IDbConnection`, `IDbCommand`, `DbParameter` etc. In addition, the APIs exposed by many of these interfaces were minimal or incomplete - making for tedious code that would otherwise be more easily developed with provider specific subclasses. Lastly, there was no common base class for data access exceptions across the providers. .NET 2.0 made many changes for the better in that regard across all these areas of concern - and Spring only plugs smaller holes in that regard to help in the portability of your data access code.

Resource Management: The 'using' block is the heart of elegant resource management in .NET from the API perspective. However, despite its elegance, writing 2-3 nested using statements for each data access method also

starts to be tedious, which introduces the risk of forgetting to do the right thing *all the time* in terms of both direct coding and 'cut-n-paste' errors. Spring centralizes this resource management in one spot so you never forget or make a mistake and rely on it always being done correctly.

Parameter management: Frequently much of data access code is related to creating appropriate parameters. To alleviate this boiler plate code Spring provides a parameter 'builder' class that allows for succinct creation of parameter collections. Also, for the case of stored procedures, parameters can be derived from the database itself which reduces parameter creation code to just one line.

Frequently result set data is converted into objects. Spring provides a simple framework to organize that mapping task and allows you to reuse mapping artifacts across your application.

Exceptions: The standard course of action when an exception is thrown from ADO.NET code is to look up the error code and then re-run the application to set a break point where the exception occurred so as to see what the command text and data values were that caused the exception. Spring provides exceptions translation from these error codes (across database vendors) to a Data Access Object exception hierarchy. This allows you to quickly understand the category of the error that occurred and also the 'bad' data which lead to the exception.

Warnings: A common means to extract warning from the database, and to optionally treat those warnings as a reason to rollback is not directly supported with the new System.Data.Common API

Portability: Where possible, increase the portability of code across database provider in the higher level API. The need adding of a parameter prefix, i.e. @ for SqlServer or ':' for oracle is one such example of an area where a higher level API can offer some help in making your code more portable.

Note that Spring's ADO.NET framework is just 'slightly' above the raw API. It does not try to compete with other higher level persistence abstractions such as result set mappers (iBATIS.NET) or other ORM tools (NHibernate). (Apologies if your favorite is left out of that short list). As always, pick and choose the appropriate level of abstraction for the task at hand. As a side note, Spring does offer integration with higher level persistence abstractions (currently NHibernate) providing such features as integration with Spring's transaction management features as well as mixing orm/ado.net operations within the same transaction.

20.3. Provider Abstraction

Before you get started executing queries against the database you need to connect to it. Chapter 19, *DbProvider* covers this topic in detail so we only discuss the basic idea of how to interact with the database in this section. One important ingredient that increases the portability of writing ADO.NET applications is to refer to the base ADO.NET interfaces, such as *IDbCommand* or *IDbParameter* in your code. However, In the .NET 1.1 BCL the only means to obtain references to instances of these interfaces is to directly instantiate the classes, i.e. for *SqlServer* this would be

```
IDbCommand command = new SqlCommand();
```

One of the classic creational patterns in the GoF Design Patterns book addresses this situation directly, the Abstract Factory pattern. This approach was applied in the .NET BCL with the introduction of the *DbProviderFactory* class which contains various factory methods that create the various objects used in ADO.NET programming. In addition, .NET 2.0 introduced new abstract base classes that all ADO.NET providers must inherit from. These base classes provide more core functionality and uniformity across the various providers as compared to the original ADO.NET interfaces.

Spring's database provider abstraction has a similar API to that of .ADO.NET 2.0's *DbProviderFactory*. The central interface is *IDbProvider* and it has factory methods that are analogous to those in the *DbProviderFactory*

class except that they return references to the base ADO.NET interfaces. Note that in keeping with the Spring Framework's philosophy, `IDbProvider` is an interface, and can thus be easily mocked or stubbed as necessary. Another key element of this interface is the `ConnectionString` property that specifies the specific runtime information necessary to connect to the provider. The interface also has a `IDbMetadata` property that contains minimal database metadata information needed to support the functionality in rest of the Spring ADO.NET framework. It is unlikely you will need to use the `DatabaseMetadata` class directly in your application.

For more information on configuring a Spring database provider refer to Chapter 19, *DbProvider*

20.3.1. Creating an instance of `IDbProvider`

Each database vendor is associated with a particular implementation of the `IDbProvider` interfaces. A variety of implementations are provided with Spring such as `SqlServer`, `Oracle` and `MySQL`. Refer to the documentation on Spring's `DbProvider` for creating a configuration for database that is not yet provided. The programmatic way to create an `IDbProvider` is shown below

```
IDbProvider dbProvider = DbProviderFactory.GetDbProvider("System.Data.SqlClient");
```

Please refer to the Chapter 19, *DbProvider* for information on how to create a `IDbProvider` in Spring's XML configuration file.

20.4. Namespaces

The ADO.NET framework consists of a few namespaces, namely `Spring.Data`, `Spring.Data.Generic`, `Spring.Data.Common`, `Spring.Data.Support`, and `Spring.Data.Object`.

The `Spring.Data` namespace contains the majority of the classes and interfaces you will deal with on a day to day basis.

The `Spring.Data.Generic` namespaces add generic versions of some classes and interfaces and you will also likely deal with this on a day to day basis if you are using .NET 2.0

The `Spring.Data.Common` namespaces contains Spring's `DbProvider` abstraction in addition to utility classes for parameter creation.

The `Spring.Data.Object` namespaces contains classes that represent RDBMS queries, updates, and stored procedures as thread safe, reusable objects.

Finally the `Spring.Data.Support` namespace is where you find the `IAdoExceptionTransactor` translation functionality and some utility classes.

20.5. Approaches to Data Access

Spring provides two styles to interact with ADO.NET. The first is a 'template' based approach in which you create an single instance of `AdoTemplate` to be used by all your DAO implementations. Your DAO methods are frequently implemented as a single method call on the template class as described in detail in the following section. The other approach a more object-oriented manner that models database operations as objects. For example, one can encapsulate the functionality of a database query via an `AdoQuery` class and a create/update/delete operation as a `AdoNonQuery` class. Stored procedures are also modelled in this manner via the class `StoredProcedure`. To use these classes you inherit from them and define the details of the operation in the constructor and implement an abstract method. This reads very cleanly when looking at DAO method implementation as you can generally see all the details of what is going on.

Generally speaking, experience has shown that the `AdoTemplate` approach reads very cleanly when looking at DAO method implementation as you can generally see all the details of what is going on as compared to the object based approach. The object based approach however, offers some advantages when calling stored procedures since it acts as a cache of derived stored procedure arguments and can be invoked passing a variable length argument list to the 'execute' method. As always, take a look at both approaches and use the approach that provides you with the most benefit for a particular situation.

20.6. Introduction to AdoTemplate

The class `AdoTemplate` is at the heart of Spring's ADO.NET support. It is based on an Inversion of Control (i.e. callback) design with the central method 'Execute' handing you a `IDbCommand` instance that has its `Connection` and `Transaction` properties set based on the transaction context of the calling code. All resource management is handled by the framework, you only need to focus on dealing with the `IDbCommand` object. The other methods in this class build upon this central 'Execute' method to provide you a quick means to execute common data access scenarios.

There are two implementations of `AdoTemplate`. The one that uses Generics and is in the namespace `Spring.Data.Generic` and the other non-generic version in `Spring.Data`. In either case you create an instance of an `AdoTemplate` by passing it a `IDbProvider` instance as shown below

```
AdoTemplate adoTemplate = new AdoTemplate(dbProvider);
```

`AdoTemplate` is a thread-safe class and as such a single instance can be used for all data access operations in your applications DAOs. `AdoTemplate` implements an `IAdoOperations` interface. Although the `IAdoOperations` interface is more commonly used for testing scenarios you may prefer to code against it instead of the direct class instance.

If you are using the generic version of `AdoTemplate` you can access the non-generic version via the property `ClassicAdoTemplate`.

The following two sections show basic usage of the `AdoTemplate` 'Execute' API for .NET 1.1 and 2.0.

20.6.1. Execute Callback

The `Execute` method and its associated callback function/inteface is the basic method upon which all the other methods in `AdoTemplate` delegate their work. If you can not find a suitable 'one-liner' method in `AdoTemplate` for your purpose you can always fall back to the `Execute` method to perform any database operation while benefiting from ADO.NET resource management and transaction enlistment. This is commonly the case when you are using special provider specific features, such as XML or BLOB support.

20.6.2. Execute Callback in .NET 2.0

In this example a simple query against the 'Northwind' database is done to determine the number of customers who have a particular postal code.

```
public int FindCountWithPostalCode(string postalCode)
{
    return adoTemplate.Execute<int>(delegate(DbCommand command)
    {
        command.CommandText =
            "select count(*) from Customers where PostalCode = @PostalCode";

        DbParameter p = command.CreateParameter();
```

```

        p.ParameterName = "@PostalCode";
        p.Value = postalCode;
        command.Parameters.Add(p);

        return (int)command.ExecuteScalar();

    });
}

```

The `DbCommand` that is passed into the anonymous delegate is already has its `Connection` property set to the corresponding value of the `dbProvider` instance used to create the template. Furthermore, the `Transaction` property of the `DbCommand` is set based on the transactional calling context of the code as based on the use of Spring's transaction management features. Also note the feature of anonymous delegates to access the variable 'postalCode' which is defined 'outside' the anonymous delegate implementation. The use of anonymous delegates is a powerful approach since it allows you to write compact data access code. If you find that your callback implementation is getting very long, it may improve code clarity to use an interface based version of the callback function, i.e. an `ICommandCallback` shown below.

As you can see, only the most relevant portions of the data access task at hand need to be coded. (Note that in this simple example you would be better off using `AdoTemplate`'s `ExecuteScalar` method directly. This method is described in the following sections). As mentioned before, the typical usage scenario for the `Execute` callback would involve downcasting the passed in `DbCommand` object to access specific provider API features.

There is also an interface based version of the execute method. The signatures for the delegate and interface are shown below

```

public delegate T CommandDelegate<T>(DbCommand command);

public interface ICommandCallback
{
    T DoInCommand<T>(DbCommand command);
}

```

While the delegate version offers the most compact syntax, the interface version allows for reuse. The corresponding method signatures on `Spring.Data.Generic.AdoTemplate` are shown below

```

public class AdoTemplate : AdoAccessor, IAdoOperations
{
    ...

    T Execute<T>(ICommandCallback action);

    T Execute<T>(CommandDelegate<T> del);

    ...
}

```

While it is common for .NET 2.0 ADO.NET provider implementations to inherit from the base class `System.Data.Common.DbCommand`, that is not a requirement. To accommodate the few that don't, which as of this writing are the latest Oracle (ODP) provider, Postgres, and DB2 for iSeries, two additional execute methods are provided. The only difference is the use of callback and delegate implementations that have `IDbCommand` and not `DbCommand` as callback arguments. The following listing shows these methods on `AdoTemplate`.

```

public class AdoTemplate : AdoAccessor, IAdoOperations
{
    ...

    T Execute<T>(IDbCommandCallback action);

    T Execute<T>(IDbCommandDelegate<T> del);
}

```

```
...
}
```

where the signatures for the delegate and interface are shown below

```
public delegate T IDbCommandDelegate<T>(IDbCommand command);

public interface IDbCommandCallback<T>
{
    T DoInCommand(IDbCommand command);
}
```

Internally the `AdoTemplate` implementation delegates to implementations of `IDbCommandCallback` so that the 'lowest common denominator' API is used to have maximum portability. If you accidentally call `Execute<T>(ICommandCallback action)` and the command does not inherit from `DbCommand`, an `InvalidDataAccessApiUsageException` will be thrown.

Depending on how portable you would like your code to be, you can choose among the two callback styles. The one based on `DbCommand` has the advantage of access to the more user friendly `DbParameter` class as compared to `IDbParameter` obtained from `IDbCommand`.

20.6.3. Execute Callback in .NET 1.1

>

`AdoTemplate` differs from its .NET 2.0 generic counterpart in that it exposes the interface `IDbCommand` in its 'Execute' callback methods and delegate as compared to the abstract base class `DbProvider`. Also, since anonymous delegates are not available in .NET 1.1, the typical usage pattern requires you to create a explicitly delegate and/or class that implements the `ICommandCallback` interface. Example code to query In .NET 1.1 the 'Northwind' database is done to determine the number of customers who have a particular postal code is shown below.

```
public virtual int FindCountWithPostalCode(string postalCode)
{
    return (int) AdoTemplate.Execute(new PostalCodeCommandCallback(postalCode));
}
```

and the callback implementation is

```
private class PostalCodeCommandCallback : ICommandCallback
{
    private string cmdText = "select count(*) from Customer where PostalCode = @PostalCode";

    private string postalCode;

    public PostalCodeCommandCallback(string postalCode)
    {
        this.postalCode = postalCode;
    }

    public object DoInCommand(IDbCommand command)
    {
        command.CommandText = cmdText;

        IDbDataParameter p = command.CreateParameter();
        p.ParameterName = "@PostalCode";
        p.Value = postalCode;
        command.Parameters.Add(p);

        return command.ExecuteScalar();
    }
}
```

```
}
```

Note that in this example, one could more easily use `AdoTemplate`'s `ExecuteScalar` method.

The `Execute` method has interface and delegate overloads. The signatures for the delegate and interface are shown below

```
public delegate object CommandDelegate(IDbCommand command);

public interface ICommandCallback
{
    object DoInCommand(IDbCommand command);
}
```

The corresponding method signatures on `Spring.Data.AdTemplate` are shown below

```
public class AdoTemplate : AdoAccessor, IAdoOperations
{
    ...

    object Execute(CommandDelegate del);

    object Execute(ICommandCallback action);

    ...
}
```

Note that you have to cast to the appropriate object type returned from the `execute` method.

20.6.4. Quick Guide to AdoTemplate Methods

There are many methods in `AdoTemplate` so it is easy to feel a bit overwhelmed when taking a look at the SDK documentation. However, after a while you will hopefully find the class 'easy to navigate' with intellisense. Here is a quick categorization of the method names and their associated data access operation. Each method is overloaded to handle common cases of passing in parameter values.

The generic 'catch-all' method

- `Execute` - Allows you to perform any data access operation on a standard `DbCommand` object. The connection and transaction properties of the `DbCommand` are already set based on the transactional calling context. There is also an overloaded method that operates on a standard `IDbCommand` object. This is for those providers that do not inherit from the base class `DbCommand`.

The following methods mirror those on the `DbCommand` object.

- `ExecuteNonQuery` - Executes the 'NonQuery' method on a `DbCommand`, applying provided parameters and returning the number of rows affected.
- `ExecuteScalar` - Executes the 'Scalar' method on a `DbCommand`, applying provided parameters, and returning the first column of the first row in the result set.

Mapping result sets to objects

- `QueryWithResultSetExtractor` - Execute a query mapping a result set to an object with an implementation of the `IResultSetExtractor` interface.
- `QueryWithResultSetExtractorDelegate` - Same as `QueryWithResultSetExtractor` but using a `ResultSetExtractorDelegate` to perform result set mapping.

- `QueryWithRowCallback` - Execute a query calling an implementation of `IRowCallback` for each row in the result set.
- `QueryWithRowCallbackDelegate` - Same as `QueryWithRowCallback` but calling a `RowCallbackDelegate` for each row.
- `QueryWithRowMapper` - Execute a query mapping a result set on a row by row basis with an implementation of the `IRowMapper` interface.
- `QueryWithRowMapperDelegate` - Same as `QueryWithRowMapper` but using a `RowMapperDelegate` to perform result set row to object mapping.

Mapping result set to a single object

- `QueryForObject` - Execute a query mapping the result set to an object using a `IRowMapper`. Exception is thrown if the query does not return exactly one object.

Query with a callback to create the `DbCommand` object. These are generally used by the framework itself to support other functionality, such as in the `Spring.Data.Objects` namespace.

- `QueryWithCommandCreator` - Execute a query with a callback to `IDbCommandCreator` to create a `IDbCommand` object and using either a `IRowMapper` or `IResultSetExtractor` to map the result set to an object. One variation lets multiple result set 'processors' be specified to act on multiple result sets and return output parameters.

DataTable and DataSet operations

- `DataTableCreate` - Create and Fill DataTables
- `DataTableCreateWithParameters` - Create and Fill DataTables using a parameter collection.
- `DataTableFill` - Fill a pre-existing DataTable.
- `DataTableFillWithParameters` - Fill a pre-existing DataTable using parameter collection.
- `DataTableUpdate` - Update the database using the provided DataTable, insert, update, delete SQL.
- `DataTableUpdateWithCommandBuilder` - Update the database using the provided DataTable, select SQL, and parameters.
- `DataSetCreate` - Create and Fill DataSets
- `DataSetCreateWithParameters` - Create and Fill DataTables using a parameter collection.
- `DataSetFill` - Fill a pre-existing DataSet
- `DataSetFillWithParameters` - Fill a pre-existing DataTable using parameter collection.
- `DataSetUpdate` - Update the database using the provided DataSet, insert, update, delete SQL.
- `DataSetUpdateWithCommandBuilder` - Update the database using the provided DataSet, select SQL, and parameters..



Note

These methods are not currently in the generic version of `AdoTemplate` but accessible through the property `ClassicAdoTemplate`.

Parameter Creation utility methods

- `DeriveParameters` - Derive the parameter collection for stored procedures.

In turn each method typically has four overloads, one with no parameters and three for providing parameters. Aside from the `DataTable/DataSet` operations, the three parameter overloads are of the form shown below

- `MethodName(CommandType cmdType, string cmdText, CallbackInterfaceOrDelegate, parameter setting arguments)`

The `CallbackInterfaceOrDelegate` is one of the three types listed previously. The parameters setting arguments are of the form

- `MethodName(... string parameterName, Enum dbType, int size, object parameterValue)`
- `MethodName(... IDbParameters parameters)`
- `MethodName(... ICommandSetter commandSetter)`

The first overload is a convenience method when you only have one parameter to set. The database enumeration is the base class 'Enum' allowing you to pass in any of the provider specific enumerations as well as the common `DbType` enumeration. This is a trade off of type-safety with provider portability. (Note generic version could be improved to provide type safety...).

The second overload contains a collection of parameters. The data type is Spring's `IDbParameters` collection class discussed in the following section.

The third overload is a callback interface allowing you to set the parameters (or other properties) of the `IDbCommand` passed to you by the framework directly.

If you are using .NET 2.0 the delegate versions of the methods are very useful since very compact definitions of database operations can be created that reference variables local to the DAO method. This removes some of the tedium in passing parameters around with interface based versions of the callback functions since they need to be passed into the constructor of the implementing class. The general guideline is to use the delegate when available for functionality that does not need to be shared across multiple DAO classes or methods and use interface based version to reuse the implementation in multiple places. The .NET 2.0 versions make use of generics where appropriate and therefore enhance type-safety.

20.6.5. Quick Guide to AdoTemplate Properties

`AdoTemplate` has the following properties that you can configure

- `LazyInit` - Indicates if the `IAdoExceptionTranslator` should be created on first encounter of an exception from the data provider or when `AdoTemplate` is created. Default is true, i.e. to lazily instantiate.
- `ExceptionTranslator` - Gets or sets the implementation of `IAdoExceptionTranslator` to use. If no custom translator is provided, a default `ErrorCodeExceptionTranslator` is used.
- `DbProvider` - Gets or sets the `IDbProvider` instance to use.
- `DataReaderWrapperType` - Gets or set the `System.Type` to use to create an instance of `IDataReaderWrapper` for the purpose of providing extended mapping functionality. Spring provides an implementation to use as the

basis for a mapping strategy that will map `DBNull` values to default values based on the standard `IDataReader` interface. See the section custom `IDataReader` implementations for more information.

- `CommandTimeout` - Gets or sets the command timeout for `IDbCommands` that this `AdoTemplate` executes. Default is 0, indicating to use the database provider's default.

20.7. Transaction Management

The `AdoTemplate` is used in conjunction with an implementation of a `IPlatformTransactionManager`, which is Spring's portable transaction management API. This section gives a brief overview of the transaction managers you can use with `AdoTemplate` and the details of how you can retrieve the connection/transaction ADO.NET objects that are bound to the thread when a transaction starts. Please refer to the section key abstractions in the chapter on transactions for more comprehensive introduction to transaction management.

To use local transactions, those with only one transactional resource (i.e. the database) you will typically use `AdoPlatformTransactionManager`. If you need to mix Hibernate and ADO.NET data access operations within the same local transaction you should use `HibernatePlatformTransactionManager` which is described more in the section on ORM transaction management.

While it is most common to use Spring's transaction management features to avoid the low level management of ADO.NET connection and transaction objects, you can retrieve the connection/transaction pair that was created at the start of a transaction and bound to the current thread. This may be useful for some integration with other data access APIs. This can be done using the utility class `ConnectionUtils` as shown below.

```
IDbProvider dbProvider = DbProviderFactory.GetDbProvider("System.Data.SqlClient");
ConnectionTxPair connectionTxPairToUse = ConnectionUtils.GetConnectionTxPair(dbProvider);

IDbCommand command = dbProvider.CreateCommand();
command.Connection = connectionTxPairToUse.Connection;
command.Transaction = connectionTxPairToUse.Transaction;
```

It is possible to provide a wrapper around the standard .NET provider interfaces such that you can use the plain ADO.NET API in conjunction with Spring's transaction management features.

If you are using `ServiceDomainPlatformTransactionManager` or `TxScopePlatformTransactionManager` then you can retrieve the currently executing transaction object via the standard .NET APIs.

20.8. Exception Translation

`AdoTemplate`'s methods throw exceptions within a Data Access Object (DAO) exception hierarchy described in Chapter 18, *DAO support*. In addition, the command text and error code of the exception are extracted and logged. This leads to easier to write provider independent exception handling layer since the exceptions thrown are not tied to a specific persistence technology. Additionally, for ADO.NET code the error messages logged provide information on the SQL and error code to better help diagnose the issue.

20.9. Parameter Management

A fair amount of the code in ADO.NET applications is related to the creation and population of parameters. The BCL parameter interfaces are very minimal and do not have many convenience methods found in provider implementations such as `SqlClient`. Even still, with `SqlClient`, there is a fair amount of verbosity to creating and populating a parameter collection. Spring provides two ways to make this mundane task easier and more portable across providers.

20.9.1. IDbParametersBuilder

Instead of creating a parameter on one line of code, then setting its type on another and size on another, a builder and parameter interface, `IDbParametersBuilder` and `IDbParameter` respectfully, are provided so that this declaration process can be condensed. The `IDbParameter` support chaining calls to its methods, in effect a simple language-constrained domain specific language, to be fancy about it. Here is an example of it in use.

```
IDbParametersBuilder builder = CreateDbParametersBuilder();
builder.Create().Name("Country").Type(DbType.String).Size(15).Value(country);
builder.Create().Name("City").Type(DbType.String).Size(15).Value(city);

// now get the IDbParameters collection for use in passing to AdoTemplate methods.

IDbParameters parameters = builder.GetParameters();
```

Please note that `IDbParameters` and `IDbParameter` are not part of the BCL, but part of the `Spring.Data.Common` namespace. The `IDbParameters` collection is a frequent argument to the overloaded methods of `AdoTemplate`.

The parameter prefix, i.e. '@' in Sql Server, is not required to be added to the parameter name. The `DbProvider` is aware of this metadata and `AdoTemplate` will add it automatically if required before execution.

An additional feature of the `IDbParametersBuilder` is to create a `Spring.FactoryObject` that creates `IDbParameters` for use in the XML configuration file of the IoC container. By leveraging Spring's expression evaluation language, the above lines of code can be taken as text from the XML configuration file and executed. As a result you can externalize your parameter definitions from your code. In combination with abstract object definitions and importing of configuration files your increase the chances of having one code base support multiple database providers just by a change in configuration files.

20.9.2. IDbParameters

This class is similar to the parameter collection class you find in provider specific implementations of `IDataParameterCollection`. It contains a variety of convenience methods to build up a collection of parameters.

Here is an abbreviated listing of the common convenience methods.

- `int Add(object parameterValue)`
- `void AddRange(Array values)`
- `IDbDataParameter AddWithValue(string name, object parameterValue)`
- `IDbDataParameter Add(string name, Enum parameterType)`
- `IDbDataParameter AddOut(string name, Enum parameterType)`
- `IDbDataParameter AddReturn(string name, Enum parameterType)`
- `void DeriveParameters(string storedProcedureName)`

Here a simple usage example

```
// inside method has local variable country and city...

IDbParameters parameters = CreateDbParameters();
```



```
parameters.AddWithValue("Country", country).DbType = DbType.String;
parameters.Add("City", DbType.String).Value = city;

// now pass on to AdoTemplate methods.
```

The parameter prefix, i.e. '@' in Sql Server, is not required to be added to the parameter name. The DbProvider is aware of this metadata and AdoTemplate will add it automatically if required before execution.

20.9.3. Parameter names in SQL text

While the use of `IDbParameters` or `IDbParametersBuilder` will remove the need for use to vendor specific parameter prefixes when creating a parameter collection, @User in Sql Server vs. :User in Oracle, you still need to specify the vendor specific parameter prefix in the SQL Text. Portable SQL in this regard is possible to implement, it is available as a feature in Spring Java. If you would like such a feature, please raise an issue [<http://jira.springsource.org/secure/CreateIssue!default.jspx?pid=10020>].

20.10. Custom IDataReader implementations

The passed in implementation of `IDataReader` can be customized. This lets you add a strategy for handling null values to the standard methods in the `IDataReader` interface or to provide sub-interface of `IDataReader` that contains extended functionality, for example support for default values. In callback code, i.e. `IRowMapper` and associated delegate, you would downcast to the sub-interface to perform processing.

Spring provides a class to map `DBNull` values to default values. When reading from a `IDataReader` there is often the need to map `DBNull` values to some default values, i.e. null or say a magic number such as -1. This is usually done via a ternary operator which decreases readability and also increases the likelihood of mistakes. Spring provides an `IDataReaderWrapper` interface (which inherits from the standard `IDataReader`) so that you can provide your own implementation of a `IDataReader` that will perform `DBNull` mapping for you in a consistent and non invasive manner to your result set reading code. A default implementation, `NullMappingDataReader` is provided which you can subclass to customize or simply implement the `IDataReaderWrapper` interface directly. This interface is shown below

```
public interface IDataReaderWrapper : IDataReader
{
    IDataReader WrappedReader
    {
        get;
        set;
    }
}
```

All of AdoTemplates callback interfaces/delegates that have an `IDataReader` as an argument are wrapped with a `IDataReaderWrapper` if the AdoTemplate has been configured with one via its `DataReaderWrapperType` property. Your implementation should support a zero-arg constructor.

Frequently you will use a common mapper for `DBNull` across your application so only one instance of AdoTemplate and `IDataReaderWrapper` is required. If you need to use multiple null mapping strategies you will need to create multiple instances of AdoTemplate and configure them appropriately in the DAO objects.

20.11. Basic data access operations

The 'ExecuteNonQuery' and 'ExecuteScalar' methods of AdoTemplate have the same functionality as the same named methods on the DbCommand object

20.11.1. ExecuteNonQuery

ExecuteNonQuery is used to perform create, update, and delete operations. It has four overloads listed below reflecting different ways to set the parameters.

An example of using this method is shown below

```
public void CreateCredit(float creditAmount)
{
    AdoTemplate.ExecuteNonQuery(CommandType.Text,
        String.Format("insert into Credits(creditAmount) VALUES ({0})",
            creditAmount));
}
```

20.11.2. ExecuteScalar

An example of using this method is shown below

```
int iCount = (int)adoTemplate.ExecuteScalar(CommandType.Text, "SELECT COUNT(*) FROM TestObjects");
```

20.12. Queries and Lightweight Object Mapping

A common ADO.NET development task is reading in a result set and converting it to a collection of domain objects. The family of QueryWith methods on AdoTemplate help in this task. The responsibility of performing the mapping is given to one of three callback interfaces/delegates that you are responsible for developing. These callback interfaces/delegates are:

- `IResultSetExtractor` / `ResultSetExtractorDelegate` - hands you a `IDataReader` object for you to iterate over and return a result object.
- `IRowCallback` / `RowCallbackDelegate` - hands you a `IDataReader` to process the current row. Returns void and as such is usually stateful in the case of `IRowCallback` implementations or uses a variable to collect a result that is available to an anonymous delegate.
- `IRowMapper` / `RowMapperDelegate` - hands you a `IDataReader` to process the current row and return an object corresponding to that row.

There are generic versions of the `IResultSetExtractor` and `IRowMapper` interfaces/delegates providing you with additional type-safety as compared to the object based method signatures used in the .NET 1.1 implementation.

As usual with callback APIs in Spring.Data, your implementations of these interfaces/delegates are only concerned with the core task at hand - mapping data - while the framework handles iteration of readers and resource management.

Each 'QueryWith' method has 4 overloads to handle common ways to bind parameters to the command text.

The following sections describe in more detail how to use Spring's lightweight object mapping framework.

20.12.1. ResultSetExtractor

The `ResultSetExtractor` gives you control to iterate over the `IDataReader` returned from the query. You are responsible for iterating through all the result sets and returning a corresponding result object. Implementations of `IResultSetExtractor` are typically stateless and therefore reusable as long as the implementation doesn't access stateful resources. The framework will close the `IDataReader` for you.

The interface and delegate signature for `ResultSetExtractors` is shown below for the generic version in the `Spring.Data.Generic` namespace

```
public interface IResultSetExtractor<T>
{
    T ExtractData(IDataReader reader);
}

public delegate T ResultSetExtractorDelegate<T>(IDataReader reader);
```

The definition for the non-generic version is shown below

```
public interface IResultSetExtractor
{
    object ExtractData(IDataReader reader);
}

public delegate object ResultSetExtractorDelegate(IDataReader reader);
```

Here is an example taken from the `Spring.Data.QuickStart`. It is a method in a DAO class that inherits from `AdoDaoSupport`, which has a convenience method '`CreateDbParametersBuilder()`'.

```
public virtual IList<string> GetCustomerNameByCountryAndCityWithParamsBuilder(string country, string city)
{
    IDbParametersBuilder builder = CreateDbParametersBuilder();
    builder.Create().Name("Country").Type(DbType.String).Size(15).Value(country);
    builder.Create().Name("City").Type(DbType.String).Size(15).Value(city);
    return AdoTemplate.QueryWithResultSetExtractor(CommandType.Text,
                                                    customerByCountryAndCityCommandText,
                                                    new CustomerNameResultSetExtractor<List<string>>(),
                                                    builder.GetParameters());
}
```

The implementation of the `ResultSetExtractor` is shown below.

```
internal class CustomerNameResultSetExtractor<T> : IResultSetExtractor<T> where T : IList<string>, new()
{
    public T ExtractData(IDataReader reader)
    {
        T customerList = new T();
        while (reader.Read())
        {
            string contactName = reader.GetString(0);
            customerList.Add(contactName);
        }
        return customerList;
    }
}
```

Internally the implementation of the `QueryWithRowCallback` and `QueryWithRowMapper` methods are specializations of the general `ResultSetExtractor`. For example, the `QueryWithRowMapper` implementation iterates through the result set, calling the callback method '`MapRow`' for each row and collecting the results in an `IList`. If you have a specific case that is not covered by the `QueryWithXXX` methods you can subclass `AdoTemplate` and follow the same implementation pattern to create a new `QueryWithXXX` method to suit your needs.

20.12.2. RowCallback

The `RowCallback` is usually a stateful object itself or populates another stateful object that is accessible to the calling code. Here is a sample take from the `Data QuickStart`

```

public class RowCallbackDao : AdoDaoSupport
{
    private string cmdText = "select ContactName, PostalCode from Customers";

    public virtual IDictionary<string, IList<string>> GetPostalCodeCustomerMapping()
    {
        PostalCodeRowCallback statefullCallback = new PostalCodeRowCallback();
        AdoTemplate.QueryWithRowCallback(CommandType.Text, cmdText,
                                         statefullCallback);

        // Do something with results in stateful callback...
        return statefullCallback.PostalCodeMultimap;
    }
}

```

The `PostalCodeRowCallback` builds up state which is then retrieved via the property `PostalCodeMultimap`. The Callback implementation is shown below

```

internal class PostalCodeRowCallback : IRowCallback
{
    private IDictionary<string, IList<string>> postalCodeMultimap =
        new Dictionary<string, IList<string>>();

    public IDictionary<string, IList<string>> PostalCodeMultimap
    {
        get { return postalCodeMultimap; }
    }

    public void ProcessRow(IDataReader reader)
    {
        string contactName = reader.GetString(0);
        string postalCode = reader.GetString(1);
        IList<string> contactNameList;
        if (postalCodeMultimap.ContainsKey(postalCode))
        {
            contactNameList = postalCodeMultimap[postalCode];
        }
        else
        {
            postalCodeMultimap.Add(postalCode, contactNameList = new List<string>());
        }
        contactNameList.Add(contactName);
    }
}

```

20.12.3. RowMapper

The `RowMapper` lets you focus on just the logic to map a row of your result set to an object. The creation of a `IList` to store the results and iterating through the `IDataReader` is handled by the framework. Here is a simple example taken from the Data QuickStart application

```

public class RowMapperDao : AdoDaoSupport
{
    private string cmdText = "select Address, City, CompanyName, ContactName, " +
        "ContactTitle, Country, Fax, CustomerID, Phone, PostalCode, " +
        "Region from Customers";

    public virtual IList<Customer> GetCustomers()
    {
        return AdoTemplate.QueryWithRowMapper<Customer>(CommandType.Text, cmdText,
                                                         new CustomerRowMapper<Customer>());
    }
}

```

where the implementation of the `RowMapper` is

```

public class CustomerRowMapper<T> : IRowMapper<T> where T : Customer, new()

```

```

{
    public T MapRow(IDataReader dataReader, int rowNum)
    {
        T customer = new T();
        customer.Address = dataReader.GetString(0);
        customer.City = dataReader.GetString(1);
        customer.CompanyName = dataReader.GetString(2);
        customer.ContactName = dataReader.GetString(3);
        customer.ContactTitle = dataReader.GetString(4);
        customer.Country = dataReader.GetString(5);
        customer.Fax = dataReader.GetString(6);
        customer.Id = dataReader.GetString(7);
        customer.Phone = dataReader.GetString(8);
        customer.PostalCode = dataReader.GetString(9);
        customer.Region = dataReader.GetString(10);
        return customer;
    }
}

```

You may also pass in a delegate, which is particularly convenient if the mapping logic is short and you need to access local variables within the mapping logic.

```

public virtual IList<Customer> GetCustomersWithDelegate()
{
    return AdoTemplate.QueryWithRowMapperDelegate<Customer>(CommandType.Text, cmdText,
        delegate(IDataReader dataReader, int rowNum)
        {
            Customer customer = new Customer();
            customer.Address = dataReader.GetString(0);
            customer.City = dataReader.GetString(1);
            customer.CompanyName = dataReader.GetString(2);
            customer.ContactName = dataReader.GetString(3);
            customer.ContactTitle = dataReader.GetString(4);
            customer.Country = dataReader.GetString(5);
            customer.Fax = dataReader.GetString(6);
            customer.Id = dataReader.GetString(7);
            customer.Phone = dataReader.GetString(8);
            customer.PostalCode = dataReader.GetString(9);
            customer.Region = dataReader.GetString(10);
            return customer;
        });
}

```

20.12.4. Query for a single object

The `QueryForObject` method is used when you expect there to be exactly one object returned from the mapping, otherwise a `Spring.Dao.IncorrectResultSizeDataAccessException` will be thrown. Here is some sample usage taken from the Data QuickStart.

```

public class QueryForObjectDao : AdoDaoSupport
{
    private string cmdText = "select Address, City, CompanyName, ContactName, " +
        "ContactTitle, Country, Fax, CustomerID, Phone, PostalCode, " +
        "Region from Customers where ContactName = @ContactName";

    public Customer GetCustomer(string contactName)
    {
        return AdoTemplate.QueryForObject(CommandType.Text, cmdText,
            new CustomerRowMapper<Customer>(),
            "ContactName", DbType.String, 30, contactName);
    }
}

```

20.12.5. Query using a CommandCreator

There is a family of overloaded methods that allows you to encapsulate and reuse a particular configuration of a `IDbCommand` object. These methods also allow for access to returned out parameters as well as a method that allows processing of multiple result sets. These methods are used internally to support the classes in the

`Spring.Data.Objects` namespace and you may find the API used in that namespace to be more convenient. The family of methods is listed below.

- `object QueryWithCommandCreator(IDbCommandCreator cc, IResultSetExtractor rse)`
- `void QueryWithCommandCreator(IDbCommandCreator cc, IRowCallback rowCallback)`
- `IList QueryWithCommandCreator(IDbCommandCreator cc, IRowMapper rowMapper)`

There is also the same methods with an additional collecting parameter to obtain any output parameters. These are

- `object QueryWithCommandCreator(IDbCommandCreator cc, IResultSetExtractor rse, IDictionary returnedParameters)`
- `void QueryWithCommandCreator(IDbCommandCreator cc, IRowCallback rowCallback, IDictionary returnedParameters)`
- `IList QueryWithCommandCreator(IDbCommandCreator cc, IRowMapper rowMapper, IDictionary returnedParameters)`

The `IDbCommandCreator` callback interface is shown below

```
public interface IDbCommandCreator
{
    IDbCommand CreateDbCommand();
}
```

The created `IDbCommand` object is used when performing the `QueryWithCommandCreator` method.

To process multiple result sets specify a list of named result set processors,(i.e. `IResultSetExtractor`, `IRowCallback`, or `IRowMapper`). This method is shown below

- `IDictionary QueryWithCommandCreator(IDbCommandCreator cc, IList namedResultSetProcessors)`

The list must contain objects of the type `Spring.Data.Support.NamedResultSetProcessor`. This is the class responsible for associating a name with a result set processor. The constructors are listed below.

```
public class NamedResultSetProcessor {

    public NamedResultSetProcessor(string name, IRowMapper rowMapper) { ... }

    public NamedResultSetProcessor(string name, IRowCallback rowcallback) { ... }

    public NamedResultSetProcessor(string name, IResultSetExtractor resultSetExtractor) { ... }

    . . .
}
```

The results of the `RowMapper` or `ResultSetExtractor` are retrieved by name from the dictionary that is returned. `RowCallbacks`, being stateless, only have the placeholder text, "ResultSet returned was processed by an `IRowCallback`" as a value for the name of the `RowCallback` used as a key. Output and InputOutput parameters can be retrieved by name. If this parameter name is null, then the index of the parameter prefixed with the letter 'P' is a key name, i.e P2, P3, etc.

The namespace `Spring.Data.Objects.Generic` contains generic versions of these methods. These are listed below

- `T QueryWithCommandCreator<T>(IDbCommandCreator cc, IResultSetExtractor<T> rse)`

- `IList<T> QueryWithCommandCreator<T>(IDbCommandCreator cc, IRowMapper<T> rowMapper)`

and overloads that have an additional collecting parameter to obtain any output parameters.

- `T QueryWithCommandCreator<T>(IDbCommandCreator cc, IResultSetExtractor<T> rse, IDictionary returnedParameters)`
- `IList<T> QueryWithCommandCreator<T>(IDbCommandCreator cc, IRowMapper<T> rowMapper, IDictionary returnedParameters)`

When processing multiple result sets you can specify up to two type safe result set processors.

- `IDictionary QueryWithCommandCreator<T>(IDbCommandCreator cc, IList namedResultSetProcessors)`
- `IDictionary QueryWithCommandCreator<T,U>(IDbCommandCreator cc, IList namedResultSetProcessors)`

The list of result set processors contains either objects of the type `Spring.Data.Generic.NamedResultSetProcessor<T>` or `Spring.Data.NamedResultSetProcessor`. The generic result set processors, `NamedResultSetProcessor<T>`, is used to process the first result set in the case of using `QueryWithCommandCreator<T>` and to process the first and second result set in the case of using `QueryWithCommandCreator<T,U>`. Additional `Spring.Data.NamedResultSetProcessors` that are listed can be used to process additional result sets. If you specify a `RowCallback` with `NamedResultSetProcessor<T>`, you still need to specify a type parameter (say string) because the `RowCallback` processor does not return any object. It is up to subclasses of `RowCallback` to collect state due to processing the result set which is later queried.

20.13. DataTable and DataSet

`AdoTemplate` contains several 'families' of methods to help remove boilerplate code and reduce common programming errors when using `DataTables` and `DataSets`. There are many methods in `AdoTemplate` so it is easy to feel a bit overwhelmed when taking a look at the SDK documentation. However, after a while you will hopefully find the class 'easy to navigate' with intellisense. Here is a quick categorization of the method names and their associated data access operation. Each method is overloaded to handle common cases of passing in parameter values.

The 'catch-all' `Execute` methods upon which other functionality is built up upon are shown below.

In `Spring.Data.Core.AdoTemplate`

- `object Execute(IDataAdapterCallback dataAdapterCallback)` - Execute ADO.NET operations on a `IDbDataAdapter` object using an interface based callback.

Where `IDataAdapterCallback` is defined as

```
public interface IDataAdapterCallback
{
    object DoInDataAdapter(IDbDataAdapter dataAdapter);
}
```

The passed in `IDbDataAdapter` will have its `SelectCommand` property created and set with its `Connection` and `Transaction` values based on the calling transaction context. The return value is the result of processing or null.

There are type-safe versions of this method in `Spring.Data.Generic.AdoTemplate`

- `T Execute<T>(IDataAdapterCallback<T> dataAdapterCallback)` - Execute ADO.NET operations on a `IDbDataAdapter` object using an interface based callback.
- `T Execute<T>(DataAdapterDelegate<T> del)` - Execute ADO.NET operations on a `IDbDataAdapter` object using an delegate based callback.

Where `IDbDataAdapterCallback<T>` and `DataAdapterDelegate<T>` are defined as

```
public interface IDbDataAdapterCallback<T>
{
    T DoInDataAdapter(IDbDataAdapter dataAdapter);
}

public delegate T DataAdapterDelegate<T>(IDbDataAdapter dataAdapter);
```

20.13.1. DataTables

`DataTable` operations are available on the class `Spring.Data.Core.AdoTemplate`. If you are using the generic version, `Spring.Data.Generic.AdoTemplate`, you can access these methods through the property `ClassicAdoTemplate`, which returns the non-generic version of `AdoTemplate`. `DataTable` operations available fall into the general family of methods with 3-5 overloads per method.

- `DataTableCreate` - Create and Fill `DataTable`s
- `DataTableCreateWithParameters` - Create and Fill `DataTable`s using a parameter collection.
- `DataTableFill` - Fill a pre-existing `DataTable`.
- `DataTableFillWithParameters` - Fill a pre-existing `DataTable` using a parameter collection.
- `DataTableUpdate` - Update the database using the provided `DataTable`, insert, update, delete SQL.
- `DataTableUpdateWithCommandBuilder` - Update the database using the provided `DataTable`, select SQL, and parameters.

20.13.2. DataSets

`DataSet` operations are available on the class `Spring.Data.Core.AdoTemplate`. If you are using the generic version, `Spring.Data.Generic.AdoTemplate`, you can access these methods through the property `ClassicAdoTemplate`, which returns the non-generic version of `AdoTemplate`. `DataSet` operations available fall into the following family of methods with 3-5 overloads per method.

- `DataSetCreate` - Create and Fill `DataSets`
- `DataSetCreateWithParameters` - Create and Fill `DataTable`s using a parameter collection.
- `DataSetFill` - Fill a pre-existing `DataSet`
- `DataSetFillWithParameters` - Fill a pre-existing `DataTable` using parameter collection.
- `DataSetUpdate` - Update the database using the provided `DataSet`, insert, update, delete SQL.
- `DataSetUpdateWithCommandBuilder` - Update the database using the provided `DataSet`, select SQL, and parameters.

The following code snippets demonstrate the basic functionality of these methods using the Northwind database. See the SDK documentation for more details on other overloaded methods.

```
public class DataSetDemo : AdoDaoSupport
{
    private string selectAll = @"select Address, City, CompanyName, ContactName, " +
        "ContactTitle, Country, Fax, CustomerID, Phone, PostalCode, " +
        "Region from Customers";

    public void DemoDataSetCreate()
    {
        DataSet customerDataSet = AdoTemplate.DataSetCreate(CommandType.Text, selectAll);

        // customerDataSet has a table named 'Table' with 91 rows

        customerDataSet = AdoTemplate.DataSetCreate(CommandType.Text, selectAll, new string[] { "Customers" });

        // customerDataSet has a table named 'Customers' with 91 rows
    }

    public void DemoDataSetCreateWithParameters()
    {
        string selectLike = @"select Address, City, CompanyName, ContactName, " +
            "ContactTitle, Country, Fax, CustomerID, Phone, PostalCode, " +
            "Region from Customers where ContactName like @ContactName";

        DbParameters dbParameters = CreateDbParameters();
        dbParameters.Add("ContactName", DbType.String).Value = "M%";
        DataSet customerLikeMDataSet = AdoTemplate.DataSetCreateWithParams(CommandType.Text, selectLike, dbParameters);

        // customerLikeMDataSet has a table named 'Table' with 12 rows
    }

    public void DemoDataSetFill()
    {
        DataSet dataSet = new DataSet();
        dataSet.Locale = CultureInfo.InvariantCulture;
        AdoTemplate.DataSetFill(dataSet, CommandType.Text, selectAll);
    }
}
```

Updating a DataSet can be done using a CommandBuilder, automatically created from the specified select command and select parameters, or by explicitly specifying the insert, update, delete commands and parameters. Below is an example, refer to the SDK documentation for additional overloads

```
public class DataSetDemo : AdoDaoSupport
{
    private string selectAll = @"select Address, City, CompanyName, ContactName, " +
        "ContactTitle, Country, Fax, CustomerID, Phone, PostalCode, " +
        "Region from Customers";

    public void DemoDataSetUpdateWithCommandBuilder()
    {
        DataSet dataSet = new DataSet();
        dataSet.Locale = CultureInfo.InvariantCulture;
        AdoTemplate.DataSetFill(dataSet, CommandType.Text, selectAll, new string[] { "Customers" } );

        AddAndEditRow(dataSet);

        AdoTemplate.DataSetUpdateWithCommandBuilder(dataSet, CommandType.Text, selectAll, null, "Customers");
    }

    public void DemoDataSetUpdateWithoutCommandBuilder()
    {
        DataSet dataSet = new DataSet();
        dataSet.Locale = CultureInfo.InvariantCulture;
        AdoTemplate.DataSetFill(dataSet, CommandType.Text, selectAll, new string[] { "Customers" } );

        AddAndEditRow(dataSet);
    }
}
```

```

string insertSql = @"INSERT Customers (CustomerID, CompanyName) VALUES (@CustomerId, @CompanyName)";
IDbParameters insertParams = CreateDbParameters();
insertParams.Add("CustomerId", DbType.String, 0, "CustomerId"); // .Value = "NewID";
insertParams.Add("CompanyName", DbType.String, 0, "CompanyName"); // .Value = "New Company Name";

string updateSql = @"update Customers SET Phone=@Phone where CustomerId = @CustomerId";
IDbParameters updateParams = CreateDbParameters();
updateParams.Add("Phone", DbType.String, 0, "Phone");//.Value = "030-0074322"; // simple change, last digit changed from
updateParams.Add("CustomerId", DbType.String, 0, "CustomerId");//.Value = "ALFKI";

AdoTemplate.DataSetUpdate(dataSet, "Customers",
                          CommandType.Text, insertSql, insertParams,
                          CommandType.Text, updateSql, updateParams,
                          CommandType.Text, null, null);
}

private static void AddAndEditRow(DataSet dataSet)
{
    DataRow dataRow = dataSet.Tables["Customers"].NewRow();
    dataRow["CustomerId"] = "NewID";
    dataRow["CompanyName"] = "New Company Name";
    dataRow["ContactName"] = "New Name";
    dataRow["ContactTitle"] = "New Contact Title";
    dataRow["Address"] = "New Address";
    dataRow["City"] = "New City";
    dataRow["Region"] = "NR";
    dataRow["PostalCode"] = "New Code";
    dataRow["Country"] = "New Country";
    dataRow["Phone"] = "New Phone";
    dataRow["Fax"] = "New Fax";
    dataSet.Tables["Customers"].Rows.Add(dataRow);

    DataRow alfkDataRow = dataSet.Tables["Customers"].Rows[0];
    alfkDataRow["Phone"] = "030-0074322"; // simple change, last digit changed from 1 to 2.
}
}

```

In the case of needing to set parameter SourceColumn or SourceVersion properties it may be more convenient to use IDbParameterBuilder.

20.14. TableAdapters and participation in transactional context

Typed DataSets need to have commands in their internal DataAdapters and command collections explicitly set with a connection/transaction in order for them to correctly participate with a surrounding transactional context. The reason for this is by default the code generated is explicitly managing the connections and transactions. This issue is very well described in the article [System.Transactions and ADO.NET 2.0](#) by ADO.NET guru Sahil Malik. Spring offers a convenience method that will use reflection to internally set the transaction on the table adapter's internal command collection to the ambient transaction. This method on the class Spring.Data.Support.TypedDataSetUtils and is named ApplyConnectionAndTx. Here is sample usage of a DAO method that uses a VS.NET 2005 generated typed dataset for a PrintGroupMapping table.

```

public PrintGroupMappingDataSet FindAll()
{
    PrintGroupMappingTableAdapter adapter = new PrintGroupMappingTableAdapter();
    PrintGroupMappingDataSet printGroupMappingDataSet = new PrintGroupMappingDataSet();

    printGroupMappingDataSet = AdoTemplate.Execute(delegate(IDbCommand command)
    {
        TypedDataSetUtils.ApplyConnectionAndTx(adapter, command);
        adapter.Fill(printGroupMappingDataSet.PrintGroupMapping);

        return printGroupMappingDataSet;
    })
    as PrintGroupMappingDataSet;

    return printGroupMappingDataSet;
}

```

}

This DAO method may be combined with other DAO operations inside a transactional context and they will all share the same connection/transaction objects.

There are two overloads of the method `ApplyConnectionAndTx` which differ in the second method argument, one takes an `IDbCommand` and the other `IDbProvider`. These are listed below

```
public static void ApplyConnectionAndTx(object typedDataSetAdapter, IDbCommand sourceCommand)

public static void ApplyConnectionAndTx(object typedDataSetAdapter, IDbProvider dbProvider)
```

The method that takes `IDbCommand` is a convenience if you will be using `AdoTemplate` callback's as the passed in command object will already have its connection and transaction properties set based on the current transactional context. The method that takes an `IDbProvider` is convenient to use when you have data access logic that is not contained within a single callback method but is instead spread among multiple classes. In this case passing the transactionally aware `IDbCommand` object can be intrusive on the method signatures. Instead you can pass in an instance of `IDbProvider` that can be obtained via standard dependency injection techniques or via a service locator style lookup.

20.15. Database operations as Objects

The `Spring.Data.Objects` and `Spring.Data.Objects.Generic` namespaces contains classes that allow one to access the database in a more object-oriented manner. By way of an example, one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. One can also execute stored procedures and run update, delete and insert statements.



Note

There is a view borne from experience acquired in the field amongst some of the Spring developers that the various RDBMS operation classes described below (with the exception of the `StoredProcedure` class) can often be replaced with straight `AdoTemplate` calls... often it is simpler to use and plain easier to read a DAO method that simply calls a method on a `AdoTemplate` direct (as opposed to encapsulating a query as a full-blown class).

It must be stressed however that this is just a *view*... if you feel that you are getting measurable value from using the RDBMS operation classes, feel free to continue using these classes.

20.15.1. AdoQuery

`AdoQuery` is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the `NewRowMapper(...)` method to provide a `IRowMapper` instance that can create one object per row obtained from iterating over the `IDataReader` that is created during the execution of the query. The `AdoQuery` class is rarely used directly since the `MappingAdoQuery` subclass provides a much more convenient implementation for mapping rows to .NET classes. Another implementations that extends `AdoQuery` is `MappingAdoQueryWithParameters` (See SDK docs for details).

The `AdoNonQuery` class encapsulates an `IDbCommand` 's `ExecuteNonQuery` method functionality. Like the `AdoQuery` object, an `AdoNonQuery` object is reusable, and like all `AdoOperation` classes, an `AdoNonQuery` can have parameters and is defined in SQL. This class provides two execute methods

- `IDictionary ExecuteNonQuery(params object[] inParameterValues)`

- `IDictionary ExecuteNonQueryByNamedParam(IDictionary inParams)`

This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

An example of an `AdoQuery` subclass to encapsulate an insert statement for a 'TestObject' (consisting only name and age columns) is shown below

```
public class CreateTestObjectNonQuery : AdoNonQuery
{
    private static string sql = "insert into TestObjects(Age,Name) values (@Age,@Name)";

    public CreateTestObjectNonQuery(IDbProvider dbProvider) : base(dbProvider, sql)
    {
        DeclaredParameters.Add("Age", DbType.Int32);
        DeclaredParameters.Add("Name", SqlDbType.NVarChar, 16);
        Compile();
    }

    public void Create(string name, int age)
    {
        ExecuteNonQuery(name, age);
    }
}
```

20.15.2. MappingAdoQuery

`MappingAdoQuery` is a reusable query in which concrete subclasses must implement the abstract `MapRow(..)` method to convert each row of the supplied `IDataReader` into an object. Find below a brief example of a custom query that maps the data from a relation to an instance of the `Customer` class.

```
public class TestObjectQuery : MappingAdoQuery
{
    private static string sql = "select TestObjectNo, Age, Name from TestObjects";

    public TestObjectQuery(IDbProvider dbProvider)
        : base(dbProvider, sql)
    {
        CommandType = CommandType.Text;
    }

    protected override object MapRow(IDataReader reader, int num)
    {
        TestObject to = new TestObject();
        to.ObjectNumber = reader.GetInt32(0);
        to.Age = reader.GetInt32(1);
        to.Name = reader.GetString(2);
        return to;
    }
}
```

20.15.3. AdoNonQuery

The `AdoNonQuery` class encapsulates an `IDbCommand`'s `ExecuteNonQuery` method functionality. Like the `AdoQuery` object, an `AdoNonQuery` object is reusable, and like all `AdoOperation` classes, an `AdoNonQuery` can have parameters and is defined in SQL. This class provides two execute methods

- `IDictionary ExecuteNonQuery(params object[] inParameterValue)`
- `IDictionary ExecuteNonQueryByNamedParam(IDictionary inParams)`

This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

```

public class CreateTestObjectNonQuery : AdoNonQuery
{
    private static string sql = "insert into TestObjects(Age,Name) values (@Age,@Name)";

    public CreateTestObjectNonQuery(IDbProvider dbProvider) : base(dbProvider, sql)
    {
        DeclaredParameters.Add("Age", DbType.Int32);
        DeclaredParameters.Add("Name", SqlDbType.NVarChar, 16);
        Compile();
    }

    public void Create(string name, int age)
    {
        ExecuteNonQuery(name, age);
    }
}

```

20.15.4. Stored Procedure

The `StoredProcedure` class is designed to make it as simple as possible to call a stored procedure. It takes advantage of metadata present in the database to look up names of in and out parameters.. This means that you don't have to explicitly declare parameters. You can of course still declare them if you prefer. There are two versions of the `StoredProcedure` class, one that uses generics and one that doesn't. Using the `StoredProcedure` class consists of two steps, first defining the in/out parameter and any object mappers and second executing the stored procedure.

The non-generic version of `StoredProcedure` is in the namespace `Spring.Data.Objects`. It contains the following methods to execute a stored procedure

- `IDictionary ExecuteScalar(params object[] inParameterValues)`
- `IDictionary ExecuteScalarByNamedParam(IDictionary inParams)`
- `IDictionary ExecuteNonQuery(params object[] inParameterValues)`
- `IDictionary ExecuteNonQueryByNamedParam(IDictionary inParams)`
- `IDictionary Query(params object[] inParameterValues)`
- `IDictionary QueryByNamedParam(IDictionary inParams)`

Each of these methods returns an `IDictionary` that contains the output parameters and/or any results from Spring's object mapping framework. The arguments to these methods can be a variable length argument list, in which case the order must match the parameter order of the stored procedure. If the argument is an `IDictionary` it contains parameter key/value pairs. Return values from stored procedures are contained under the key `"RETURN_VALUE"`.

The standard in/out parameters for the stored procedure can be set programmatically by adding to the parameter collection exposed by the property `DeclaredParameters`. For each result sets that is returned by the stored procedures you can registering either an `IResultSetExtractor`, `IRowCallback`, or `IRowMapper` by name, which is used later to extract the mapped results from the returned `IDictionary`.

Lets take a look at an example. The following stored procedure class will call the `CustOrdersDetail` stored procedure in the Northwind database, passing in the `OrderID` as a stored procedure argument and returning a collection of `OrderDetails` business objects.

```

public class CustOrdersDetailStoredProc : StoredProcedure
{
    private static string procedureName = "CustOrdersDetail";

    public CustOrdersDetailStoredProc(IDbProvider dbProvider) : base(dbProvider, procedureName)
    {
    }
}

```

```

    {
        DeriveParameters();
        AddRowMapper("orderDetailRowMapper", new OrderDetailRowMapper() );
        Compile();
    }

    public virtual IList GetOrderDetails(int orderId)
    {
        IDictionary outParams = Query(orderId);
        return outParams["orderDetailRowMapper"] as IList;
    }
}

```

The 'DeriveParameters' method saves you the trouble of having to declare each parameter explicitly. When using DeriveParameters it is often common to use the Query method that takes a variable length list of arguments. This assumes additional knowledge on the order of the stored procedure arguments. If you do not want to follow this loose shorthand convention, you can call the method QueryByNamesParameters instead passing in a IDictionary of parameter key/value pairs.



Note

If you would like to have the return value of the stored procedure included in the returned dictionary, pass in true as a method parameter to DeriveParameters().

The StoredProcedure class is threadsafe once 'compiled', an act which is usually done in the constructor. This sets up the cache of database parameters that can be used on each call to Query or QueryByNamedParam. The implementation of IRowMapper that is used to extract the business objects is 'registered' with the class and then later retrieved by name as a fictional output parameter. You may also register IRowCallback and IResultSetExtractor callback interfaces via the AddRowCallback and AddResultSetExtractor methods.

The generic version of StoredProcedure is in the namespace Spring.Data.Objects.Generic. It allows you to define up to two generic type parameters that will be used to process result sets returned from the stored procedure. An example is shown below

```

public class CustOrdersDetailStoredProc : StoredProcedure
{
    private static string procedureName = "CustOrdersDetail";

    public CustOrdersDetailStoredProc(IDbProvider dbProvider) : base(dbProvider, procedureName)
    {
        DeriveParameters();
        AddRowMapper("orderDetailRowMapper", new OrderDetailRowMapper<OrderDetails>() );
        Compile();
    }

    public virtual List<OrderDetails> GetOrderDetails(int orderId)
    {
        IDictionary outParams = Query<OrderDetails>(orderId);
        return outParams["orderDetailRowMapper"] as List<OrderDetails>;
    }
}

```

You can find ready to run code demonstrating the StoredProcedure class in the example 'Data Access' that is part of the Spring.NET distribution.

Chapter 21. Object Relational Mapping (ORM) data access

21.1. Introduction

The Spring Framework provides integration with *NHibernate* in terms of resource management, DAO implementation support, and transaction strategies. For example for *NHibernate*, there is first-class support with lots of IoC convenience features, addressing many typical *NHibernate* integration issues. All of these support packages for O/R (Object Relational) mappers comply with Spring's generic transaction and DAO exception hierarchies. There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against the 'plain' *NHibernate* APIs. In both cases, DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

You can use Spring's support for *NHibernate* without needing to use Spring IoC or transaction management functionality. The *NHibernate* support classes can be used in typical 3rd party library style. However, usage inside a Spring IoC container does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside a Spring container.

Some of the benefits of using the Spring Framework to create your ORM DAOs include:

- *Ease of testing.* Spring's IoC approach makes it easy to swap the implementations and config locations of *Hibernate SessionFactory* instances, *ADO.NET DbProvider* instances, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.
- *Common data access exceptions.* Spring can wrap exceptions from your O/R mapping tool of choice, converting them from proprietary exceptions to a common runtime *DataAccessException* hierarchy. You can still trap and handle exceptions anywhere you need to. Remember that *ADO.NET* exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with *ADO.NET* within a consistent programming model.
- *General resource management.* Spring application contexts can handle the location and configuration of *Hibernate ISessionFactory* instances, *ADO.NET DbProvider* instances and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of persistence resources. For example: related code using *NHibernate* generally needs to use the same *NHibernate Session* for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a *Session* to the current thread, either by using an explicit 'template' wrapper class at the code level or by exposing a current *Session* through the *Hibernate SessionFactory* (for DAOs based on plain *Hibernate* 1.2 API). Thus Spring solves many of the issues that repeatedly arise from typical *NHibernate* usage, for any transaction environment (local or distributed).
- *Integrated transaction management.* Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your *Hibernate/ADO.NET* related code being affected: for example, between local transactions and distributed, with the same full services (such as declarative transactions) available in both scenarios. As an additional benefit, *ADO.NET*-related code can fully integrate transactionally with the

code you use to do O/R mapping. This is useful for data access that's not suitable for O/R mapping which still needs to share common transactions with ORM operations.

The NHibernate Northwind example in the Spring distribution shows a NHibernate implementation of a persistence-technology agnostic DAO interfaces. (In the upcoming RC1 release the SpringAir example will demonstrate an ADO.NET and NHibernate based implementation of technology agnostic DAO interfaces.) The NHibernate Northwind example serves as a working sample application that illustrates the use of NHibernate in a Spring web application. It also leverages declarative transaction demarcation with different transaction strategies.

Both NHibernate 1.0 and NHibernate 1.2 are supported. Differences relate to the use of generics and new features such as contextual sessions. For information on the latter, refer to the section Implementing DAOs based on the plain NHibernate API. The NHibernate 1.0 support is in the assembly `Spring.Data.NHibernate` and the 1.2 support is in the assembly `Spring.Data.NHibernate12`.

At the moment the only ORM supported in NHibernate, but others can be integrated with Spring (in as much as makes sense) to offer the same value proposition.

21.2. NHibernate

We will start with a coverage of [NHibernate](#) in a Spring environment, using it to demonstrate the approach that Spring takes towards integrating O/R mappers. This section will cover many issues in detail and show different variations of DAO implementations and transaction demarcations. Most of these patterns can be directly translated to all other supported O/R mapping tools.

The following discussion focuses on Hibernate 1.0.4, the major differences with NHibernate 1.2 being the ability to participate in Spring transaction/session management via the normal NHibernate API instead of the 'template' approach. Spring supports both NHibernate 1.0 and NHibernate 1.2 via separate .dlls with the same internal namespace.

21.2.1. Resource management

Typical business applications are often cluttered with repetitive resource management code. Many projects try to invent their own solutions for this issue, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates strikingly simple solutions for proper resource handling, namely IoC via templating; for example infrastructure classes with callback interfaces, or applying AOP interceptors. The infrastructure cares for proper resource handling, and for appropriate conversion of specific API exceptions to a common infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct ADO.NET, the `AdoTemplate` class mentioned in a previous section cares for connection handling, and for proper conversion of ADO.NET data access exceptions (not even singly rooted in .NET 1.1) to Spring's `DataAccessException` hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. It supports both distributed and local transactions, via respective Spring transaction managers.

Spring also offers Hibernate support, consisting of a `HibernateTemplate` analogous to `AdoTemplate`, a `HibernateInterceptor`, and a `Hibernate` transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable as possible. All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain object instances that don't need to be Spring-aware. In a typical Spring application,

many important objects are plain .NET objects: data access templates, data access objects (that use the templates), transaction managers, business services (that use the data access objects and transaction managers), ASP.NET web pages (that use the business services), and so on.

21.2.2. Transaction Management

While NHibernate offers an API for transaction management you will quite likely find the benefits of using Spring's generic transaction management features to be more compelling to use, typically for use of a declarative programming model for transaction demarcation and easily mixing ADO.NET and NHibernate operations within a single transaction. See the chapter on transaction management for more information on Spring's transaction management features. There are two choices for transaction management strategies, one based on the NHibernate API and the other the .NET 2.0 TransactionScope API.

The first strategy is encapsulated in the class `Spring.Data.NHibernate.HibernateTransactionManager` in both the `Spring.Data.NHibernate` namespace. This strategy is preferred when you are using a single database. ADO.NET operations can also participate in the same transaction, either by using `AdoTemplate` or by retrieving the ADO.NET connection/transaction object pair stored in thread local storage when the transaction begins. Refer to the documentation of Spring's ADO.NET framework for more information on retrieving and using the connection/transaction pair without using `AdoTemplate`. You can use the `HibernateTransactionManager` and associated classes such as `SessionFactory`, `HibernateTemplate` directly as you would any third party API, however they are most commonly used through Spring's XML configuration file to gain the benefits of easy configuration for a particular runtime environment and as the basis for the configuration of a data access layer also configured using XML. An XML fragment showing the declaration of `HibernateTransactionManager` is shown below.

```
<object id="transactionManager"
      type="Spring.Data.NHibernate.HibernateTransactionManager, Spring.Data.NHibernate">

  <property name="DbProvider" ref="DbProvider"/>
  <property name="SessionFactory" ref="MySessionFactory"/>

</object>
```

The important property of `HibernateTransactionManager` are the references to the `DbProvider` and the `HibernateSessionFactory`. For more information on the `DbProvider`, refer to the chapter `DbProvider` and the following section on `SessionFactory` set up.

The second strategy is to use the class `Spring.Data.TxScopeTransactionManager` that uses .NET 2.0 `System.Transaction` namespace and its corresponding `TransactionScope` API. This is preferred when you are using multiple transactional resources, such as multiple databases.

Both strategies associate one `Hibernate Session` for the scope of the transaction (scope in the general demarcation sense, not `System.Transaction` sense). If there is no transaction then a new `Session` will be opened for each operation. The exception to this rule is when using the `OpenSessionInViewModule` in a web application in single session mode (see Section 21.2.8, “Web Session Management”). In this case the session will be created on the start of the web request and closed on the end of the request. Note that the session's flush mode will be set to `FlushMode.NEVER` at the start of the request. If a non-readonly transaction is performed, then during the scope of that transaction processing the flush mode will be changed to `AUTO`, and then set back to `NEVER` at the end of the transaction scope so that any changes to objects associated with the session during rendering will not be persisted back to the database when the session is closed at the end of the web request.

21.2.3. SessionFactory set up in a Spring container

To avoid tying application objects to hard-coded resource lookups, Spring allows you to define resources like a `DbProvider` or a `Hibernate SessionFactory` as objects in an application context. Application objects that need to

access resources just receive references to such pre-defined instances via object references (the DAO definition in the next section illustrates this). The following excerpt from an XML application context definition shows how to set up Spring's ADO.NET DbProvider and a Hibernate `SessionFactory` on top of it:

```
<objects xmlns="http://www.springframework.net"
  xmlns:db="http://www.springframework.net/database">

  <!-- Property placeholder configurer for database settings -->

  <object type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">
    <property name="ConfigSections" value="databaseSettings"/>
  </object>

  <!-- Database and NHibernate Configuration -->

  <db:provider id="DbProvider"
    provider="SqlServer-1.1"
    connectionString="Integrated Security=false; Data Source=(local);Integrated Security=true;Database=Northw

  <object id="MySessionFactory" type="Spring.Data.NHibernate.LocalSessionFactoryObject, Spring.Data.NHibernate">
    <property name="DbProvider" ref="DbProvider"/>
    <property name="MappingAssemblies">
      <list>
        <value>Spring.Northwind.Dao.NHibernate</value>
      </list>
    </property>
    <property name="HibernateProperties">
      <dictionary>

        <entry key="hibernate.connection.provider"
          value="NHibernate.Connection.DriverConnectionProvider"/>

        <entry key="hibernate.dialect"
          value="NHibernate.Dialect.MsSql2000Dialect"/>

        <entry key="hibernate.connection.driver_class"
          value="NHibernate.Driver.SqlClientDriver"/>

      </dictionary>
    </property>
  </object>
</objects>
```

Many of the properties on `LocalSessionFactoryObject` are those you will commonly configure, for example the property `MappingAssemblies` specifies a list of assemblies to search for hibernate mapping files. The property `HibernateProperties` are the familiar `NHibernate` properties used to set typical options such as dialect and driver class. The location of `NHibernate` mapping information can also be specified using Spring's `IResource` abstraction via the property `MappingResources`. The `IResource` abstraction supports opening an input stream from assemblies, file system, and http(s) based on a `Uri` syntax. You can also leverage the extensibility of `IResource` and thereby allow `NHibernate` to obtain its configuration information from locations such as a database or LDAP. For other properties you can configure them as you normal using the file `hibernate.cfg.xml` and refer to it via the property `ConfigFileNames`. This property is a string array so multiple configuration files are supported.

There are other properties in `LocalSessionFactoryObject` that relate to the integration of Spring with `NHibernate`. The property `ExposeTransactionAwareSessionFactory` is discussed below and allows you to use Spring's declarative transaction demarcation functionality with the standard `NHibernate` API (as compared to using `HibernateTemplate`).

The property `DbProvider` is used to infer two `NHibernate` configurations options.

- Infer the connection string, typically done via the hibernate property "hibernate.connection.connection_string".
- Delegate to the `DbProvider` itself as the NHibernate connection provider instead of listing it via property `hibernate.connection.provider` via `HibernateProperties`.

If you specify both the property `hibernate.connection.provider` and `DbProvider` (as shown above) the configuration of the property `hibernate.connection.provider` is used and a warning level message is logged. If you use Spring's `DbProvider` as the NHibernate connection provider then you can take advantage of `IDbProvider` implementations that will let you change the connection string at runtime such as `UserCredentialsDbProvider` and `MultiDelegatingDbProvider`.



Note

`UserCredentialsDbProvider` and `MultiDelegatingDbProvider` only change the connection string at runtime based on values in thread local storage and do not clear out the Hibernate cache that is unique to each `ISessionFactory` instance. As such, they are only useful for selecting at runtime a single database instance. Cleaning up an existing session factory when switching to a new database is left to user code. Creating a new session factory per connection string (assuming the same mapping files can be used across all databases connections) is not currently supported. To support this functionality, you can subclass `LocalSessionFactoryObject` and override the method `ISessionFactory newSessionFactory(Configuration config)` so that it returns an implementation of `ISessionFactory` that selects among multiple instances based on values in thread local storage, much like the implementation of `MultiDelegatingDbProvider`.

21.2.3.1. Using FluentNHibernate to configure mappings with LocalSessionFactoryObject

Direct support for configuration of NHibernate mapping files using FluentNHibernate will be included in a future release. Until then, to see how you can extend `LocalSessionFactoryObject` to support using FluentNHibernate follow the instructions on Benny Michielson's blog post here [<http://www.bennymichielsen.be/post/2009/01/04/Using-Fluent-NHibernate-in-SpringNet.aspx>].

21.2.3.2. Spring's IByteCodeProvider implementation

Introduced in Hibernate 2.1 is support for dependency injection of hibernate managed objects [<http://fabioaulo.blogspot.com/2009/05/nhibernate-ioc-integration.html>] via the `IBytecodeProvider` extension point. As of Spring 1.3 provides `Spring.Data.NHibernate.Bytecode.BytecodeProvider` as the default `IBytecodeProvider` implementation when using `LocalSessionFactory` object to configure an `ISessionFactory`. To use a different `IBytecodeProvider` configure it via the standard the Hibernate means, using `App.config` or `Web.config` via the element `<bytecode-provider type="..." />` inside the `<hibernate-configuration>` section or programmatically by setting `Environment.BytecodeProvider`.

21.2.4. Implementing DAOs based on plain Hibernate 1.2/2.x API

Hibernate 1.2 introduced a feature called "contextual Sessions", where Hibernate itself manages one current `ISession` per transaction. This is roughly equivalent to Spring's synchronization of one Hibernate `Session` per transaction. A corresponding DAO implementation looks like as follows, based on the plain Hibernate API:

```
public class ProductDaoImpl implements IProductDao {

    private SessionFactory sessionFactory;

    public ISessionFactory SessionFactory
    {
        get { return sessionFactory; }
        set { sessionFactory = value; }
    }
}
```

```

    public IList<Product> LoadProductsByCategory(String category) {
        return SessionFactory.GetCurrentSession()
            .CreateQuery("from test.Product product where product.category=?")
            .SetParameter(0, category)
            .List<Product>();
    }
}

public class HibernateCustomerDao : ICustomerDao {

    private ISessionFactory sessionFactory;

    public ISessionFactory SessionFactory
    {
        set { sessionFactory = value; }
    }

    public Customer SaveOrUpdate(Customer customer)
    {
        sessionFactory.GetCurrentSession().SaveOrUpdate(customer);
        return customer;
    }
}

```

The above DAO follows the Dependency Injection pattern: it fits nicely into a Spring IoC container, just like it would if coded against Spring's `HibernateTemplate`. Of course, such a DAO can also be set up in plain C# (for example, in unit tests): simply instantiate it and call `SessionFactory` property with the desired factory reference. As a Spring object definition, it would look as follows:

```

<objects>

  <object id="CustomerDao" type="Spring.Northwind.Dao.NHibernate.HibernateCustomerDao, Spring.Northwind.Dao.NHibernate">
    <property name="sessionFactory" ref="MySessionFactory"/>
  </object>

</objects>

```

The `SessionFactory` configuration to support this programming model can be done two ways, both via configuration of Spring's `LocalSessionFactoryObject`. You can enable the use of Spring's implementation of the `NHibernate` extension interface, `ICurrentSessionContext`, by setting the property 'ExposeTransactionAwareSessionFactory' to true on `LocalSessionFactoryObject`. This is just a short-cut for setting the `NHibernate` property `current_session_context_class` with the name of the implementation class to use.

The first way is shown below

```

<object id="sessionFactory" type="Spring.Data.NHibernate.LocalSessionFactoryObject, Spring.Data.NHibernate12">

  <property name="ExposeTransactionAwareSessionFactory" value="true" />

  <!-- other configuration settings omitted -->

</object>

```

Which is simply a shortcut for the following configuration

```

<object id="sessionFactory" type="Spring.Data.NHibernate.LocalSessionFactoryObject, Spring.Data.NHibernate12">

  <!-- other configuration settings omitted -->

  <property name="HibernateProperties">
    <dictionary>

      <!-- other dictionary entries omitted -->

      <entry key="hibernate.current_session_context_class"

```

```

        value="Spring.Data.NHibernate.SpringSessionContext, Spring.Data.NHibernate12"/>

    </dictionary>
</property>

</object>

```

The main advantage of this DAO style is that it depends on the Hibernate API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and will no doubt feel more natural to Hibernate developers.

21.2.4.1. Exception Translation

However, the DAO implementation as shown throws plain `HibernateException` which means that callers can only treat exceptions as generally fatal - unless they want to depend on Hibernate's own exception hierarchy. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This trade off might be acceptable to applications that are strongly Hibernate-based and/or do not need any special exception treatment. As an alternative you can use Spring's exception translation advice to convert the `NHibernate` exception to Spring's `DataAccessException` hierarchy.

Spring offers a solution allowing exception translation to be applied transparently through the `[Repository]` attribute:

```

@Repository
public class HibernateCustomerDao : ICustomerDao {

    // class body here

}

```

and register an exception translation post processor.

```

<objects>

    <!-- configure session factory (omitted for brevity) -->

    <!-- Exception translation object post processor -->
    <object type="Spring.Dao.Attributes.PersistenceExceptionTranslationPostProcessor, Spring.Data"/>

    <!-- Same DAO configuration as before -->
    <object id="CustomerDao" type="Spring.Northwind.Dao.NHibernate.HibernateCustomerDao, Spring.Northwind.Dao.NHibernate">
        <property name="sessionFactory" ref="MySessionFactory"/>
    </object>

</objects>

```

The `postprocessor` will automatically look for all exception translators (implementations of the `IPersistenceExceptionTranslator` interface) and advise all object marked with the `[Repository]` attribute so that the discovered translators can intercept and apply the appropriate translation on the thrown exceptions. Spring's `LocalSessionFactory` object implements the `IPersistenceExceptionTranslator` interface and performs the same exception translation as was done when using `HibernateTemplate`.

The `[Repository]` attribute is defined in the `Spring.Data` assembly, however it is used as a 'marker' attribute, and you can provide your own if you would like to avoid coupling your DAO implementation to a Spring attribute. This is done by setting `PersistenceExceptionTranslationPostProcessor`'s property `RepositoryAttributeType` to your own attribute type.



Note

In summary: DAOs can be implemented based on the plain Hibernate 1.2/2.0 API, while still being able to participate in Spring-managed transactions and exception translation.

21.2.5. Declarative transaction demarcation

Alternatively, one can use Spring's declarative transaction support, which essentially enables you to replace explicit transaction demarcation API calls in your C# code with an AOP transaction interceptor configured in a Spring container. You can either externalize the transaction semantics (like propagation behavior and isolation level) in a configuration file or use the Transaction attribute on the service method to set the transaction semantics.

An example showing attribute driven transaction is shown below

```
<objects>

  <object id="TransactionManager"
    type="Spring.Data.NHibernate.HibernateTransactionManager, Spring.Data.NHibernate">

    <property name="DbProvider" ref="DbProvider"/>
    <property name="SessionFactory" ref="MySessionFactory"/>

  </object>

  <!-- DAO definition not listed, see above for an example. -->

  <object id="FulfillmentService" type="Spring.Northwind.Service.FulfillmentService, Spring.Northwind.Service">
    <property name="CustomerDao" ref="CustomerDao"/>
    <property name="OrderDao" ref="OrderDao"/>
    <property name="ShippingService" ref="ShippingService"/>
  </object>

  <!-- Import 'standard xml' configuration for attribute driven declarative tx management -->
  <import resource="DeclarativeServicesAttributeDriven.xml"/>

</objects>
```

Note that with the new transaction namespace, you can replace the importing of DeclarativeServicesAttributeDriven.xml with the following single line, `<tx:attribute-driven/>` that more clearly expresses the intent as compared to the contents of DeclarativeServicesAttributeDriven.xml.

```
<objects xmlns="http://www.springframework.net"
  xmlns:tx="http://www.springframework.net/schema/tx">

  <object id="transactionManager"
    type="Spring.Data.NHibernate.HibernateTransactionManager, Spring.Data.NHibernate">

    <property name="DbProvider" ref="DbProvider"/>
    <property name="SessionFactory" ref="MySessionFactory"/>

  </object>

  <!-- DAO definition not listed, see above for an example. -->

  <object id="FulfillmentService" type="Spring.Northwind.Service.FulfillmentService, Spring.Northwind.Service">
    <property name="CustomerDao" ref="CustomerDao"/>
    <property name="OrderDao" ref="OrderDao"/>
    <property name="ShippingService" ref="ShippingService"/>
  </object>

  <tx:attribute-driven/>

</objects>
```

The placement of the transaction attribute in the service layer method is shown below.

```
public class FulfillmentService : IFulfillmentService
{
  // fields and properties for dao object omitted, see above
```

```

[Transaction(ReadOnly=false)]
public void ProcessCustomer(string customerId)
{
    //Find all orders for customer
    Customer customer = CustomerDao.FindById(customerId);

    foreach (Order order in customer.Orders)
    {
        //Validate Order
        Validate(order);

        //Ship with external shipping service
        ShippingService.ShipOrder(order);

        //Update shipping date
        order.ShippedDate = DateTime.Now;

        //Update shipment date
        OrderDao.SaveOrUpdate(order);

        //Other operations...Decrease product quantity... etc
    }
}
}

```

If you prefer to not use attribute to demarcate your transaction boundaries, you can import a configuration file with the following XML instead of using `<tx:attribute-driven/>`

```

<object id="TxProxyConfigurationTemplate" abstract="true"
    type="Spring.Transaction.Interceptor.TransactionProxyFactoryObject, Spring.Data">

    <property name="PlatformTransactionManager" ref="HibernateTransactionManager"/>

    <property name="TransactionAttributes">
        <name-values>
            <!-- Add common methods across your services here -->
            <add key="Process*" value="PROPAGATION_REQUIRED"/>
        </name-values>
    </property>
</object>

```

Refer to the documentation on Spring Transaction management for configuration of other features, such as rollback rules.

21.2.6. Programmatic transaction demarcation

Transactions can be demarcated in a higher level of the application, on top of such lower-level data access services spanning any number of operations. There are no restrictions on the implementation of the surrounding business service here as well, it just needs a Spring `PlatformTransactionManager`. Again, the latter can come from anywhere, but preferably as an object reference via a `TransactionManager` property - just like the `productDAO` should be set via a `setProductDao(...)` method. The following snippets show a transaction manager and a business service definition in a Spring application context, and an example for a business method implementation.

```

<objects>

    <object id="TransactionManager"
        type="Spring.Data.NHibernate.HibernateTransactionManager, Spring.Data.NHibernate">

        <property name="DbProvider" ref="DbProvider"/>
        <property name="SessionFactory" ref="MySessionFactory"/>

    </object>

    <!-- DAO definition not listed, see above for an example. -->

```



```
<object id="FulfillmentService" type="Spring.Northwind.Service.FulfillmentService, Spring.Northwind.Service">
  <property name="CustomerDao" ref="CustomerDao"/>
  <property name="OrderDao" ref="OrderDao"/>
  <property name="ShippingService" ref="ShippingService"/>
  <property name="TransactionManager" ref="TransactionManager"/>
</object>

</objects>
```

```
public class FulfillmentService : IFulfillmentService

    private TransactionTemplate transactionTemplate;

    private IProductDao productDao;

    private ICustomerDao customerDao;

    private IOrderDao orderDao;

    private IShippingService shippingService;

    public TransactionManager TransactionManager
    {
        set { transactionTemplate = new TransactionTemplate(value);
        }
    }
    public void ProcessCustomer(string customerId)
    {
        tt.Execute(delegate(ITransactionStatus status)
        {
            //Find all orders for customer
            Customer customer = CustomerDao.FindById(customerId);
            foreach (Order order in customer.Orders)
            {
                //Validate Order
                Validate(order);

                //Ship with external shipping service
                ShippingService.ShipOrder(order);

                //Update shipping date
                order.ShippedDate = DateTime.Now;

                //Update shipment date
                OrderDao.SaveOrUpdate(order);

                //Other operations...Decrease product quantity... etc
            }
            return null;
        });
    }
}
```

21.2.7. Transaction management strategies

Both `TransactionTemplate` and `TransactionInterceptor` (not yet seen explicitly in above configuration, `TransactionProxyFactoryObject` uses a `TransactionInterceptor`, you would have to specify it explicitly if you were using an ordinary `ProxyFactoryObject`.) delegate the actual transaction handling to a `PlatformTransactionManager` instance, which can be a `HibernateTransactionManager` (for a single `Hibernate SessionFactory`, using a `ThreadLocal Session` under the hood) or a `TxScopeTransactionManager` (delegating to MS-DTC for distributed transaction) for `Hibernate` applications. You could even use a custom `PlatformTransactionManager` implementation. So switching from native `Hibernate` transaction management to `TxScopeTransactionManager`, such as when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the `Hibernate` transaction manager with `Spring's TxScopeTransactionManager` implementation. Both transaction demarcation and data access code will work without changes, as they just use the generic transaction management APIs.

For distributed transactions across multiple Hibernate session factories, simply combine `TxScopeTransactionManager` as a transaction strategy with multiple `LocalSessionFactoryObject` definitions. Each of your DAOs then gets one specific `SessionFactory` reference passed into it's respective object property.

TO BE DONE

`HibernateTransactionManager` can export the ADO.NET Transaction used by Hibernate to plain ADO.NET access code, for a specific `DbProvider`. (matching connection string). This allows for high-level transaction demarcation with mixed Hibernate/ADO.NET data access!

21.2.8. Web Session Management

The open session in view pattern keeps the hibernate session open during page rendering so lazily loaded hibernate objects can be displayed. You configure its use by adding an additional custom HTTP module declaration as shown below

```
<system.web>
  <httpModules>
    <add name="OpenSessionInView" type="Spring.Data.NHibernate.Support.OpenSessionInViewModule, Spring.Data.NHibernate"/>
  </httpModules>

  ...

</system.web>
```

You can configure which `SessionFactory` the `OpenSessionInViewModule` will use by setting 'global' application key-value pairs as shown below. (this will change in future releases)

```
<appSettings>
  <add key="Spring.Data.NHibernate.Support.OpenSessionInViewModule.SessionFactoryObjectName" value="SessionFactory"/>
</appSettings>
```

The default behavior of the module is that a single session is currently used for the life of the request. Refer to the earlier section on Transaction Management in this chapter for more information on how sessions are managed in the `OpenSessionInViewModule`. You can also configure in the application setting the `EntityInterceptorObjectName` using the key `Spring.Data.NHibernate.Support.OpenSessionInViewModule.EntityInterceptorObjectName` and if `SingleSession` mode is used via the key `Spring.Data.NHibernate.Support.OpenSessionInViewModule.SingleSession`. If `SingleSession` is set to false, referred to as 'deferred close mode', then each transaction scope will use a new Session and kept open until the end of the web request. This has the drawback that the first level cache is not reused across transactions and that objects are required to be unique across all sessions. Problems can arise if the same object is associated with more than one hibernate session.



Important

By default, OSIV applies `FlushMode.NEVER` on every session it creates. This is because if OSIV flushed pending changes during "EndRequest" and an error occurs, all response has already been sent to the client. There would be no way of telling the client about the error.

By default this means you **MUST** explicitly demarcate transaction boundaries around non-readonly statements when using OSIV. For configuring transactions see Section 21.2.5, "Declarative transaction demarcation" or the **`Spring.Data.NHibernate.Northwind`** example application.

21.2.9. Session Scope

The class `Spring.Data.NHibernate.Support.SessionScope` allows for you to use a single NHibernate session across multiple transactions. The usage is shown below

```
using (new SessionScope())
{
    ... do multiple operations with a single session, possibly in multiple transactions.
}
```

Refer to the API documentation for information on overloaded constructor. At the end of the using block the session is automatically closed. All transactions within the scope use the same session, if you are using Spring's `HibernateTemplate` or using Spring's implementation of NHibernate 1.2's `ICurrentSessionContext` interface. See other sections in this chapter for further information on those usage scenarios.

21.2.10. Integration Testing

When using Spring's Integration Testing support, you should make sure that the hibernate session is flushed so that the database is updated, as compared to just updating the hibernate session cache. You can implement a base class as shown below to help with the integration testing

```
public abstract class NHibernateIntegrationTests : AbstractTransactionalSpringContextTests
{
    private SessionFactory sessionFactory;

    public ISessionFactory SessionFactory
    {
        get { return sessionFactory; }
        set { sessionFactory = value; }
    }

    protected override void OnSetUpInTransaction()
    {
        base.OnSetUpInTransaction();
        Assert.IsNotNull(SessionFactory);
        SessionFactory.GetCurrentSession().FlushMode = FlushMode.Always;
        SessionFactory.GetCurrentSession().CacheMode = CacheMode.Ignore;
    }
}
```

Part III. The Web

This part of the reference documentation covers the Spring Framework's support for the presentation tier, specifically web-based presentation tiers.

- Chapter 22, *Spring.NET Web Framework*
- Chapter 23, *ASP.NET AJAX*

Chapter 22. Spring.NET Web Framework

22.1. Introduction to Spring.NET Web Framework

The Spring.NET Web Framework increases your productivity when you write ASP.NET WebForms applications by offering capabilities not found in other .NET web frameworks.

The Spring.NET Web Framework makes it easy to write 'thin and clean' web applications. "Thin" refers to WebForm's role as a small as possible adapter between the HTML- based world of the web and the Object-oriented world of your application. The business logic does not reside in the web tier; it resides in the application layer with which your web form communicates. "Clean" refers to the framework's appropriate separation of concerns, separating web specific processing such as copying data out and into from element from a data model from calling into a business tier and redirecting to the next page. This results in an event-handler that does not contain any reference to UI elements thereby making it possible to test your event handler code in integration style tests. The Spring.NET Web Framework reduces the incidental complexity of common tasks in the web tier, for example, the conversion of HTML control data to objects and then vice-versa after the request is processed by the application layer.

Highlights of Spring's Web framework are:

- **Dependency Injection.** Provided for all ASP.NET artifacts, including pages and user controls, modules, providers, and HTTP handlers. Your pages, controls, and so on do not require any dependency on Spring in order to be configured via dependency injection.
- **Bidirectional data binding.** Allows you to declaratively define the data that will be marshaled out of your HTML and user controls and into a data model that in turn is generally submitted to the application layer. After the data model is updated in the application layer, those changes are automatically reflected in the HTML and user controls on post back. This process removes large amounts of tedious, error-prone boilerplate code.
- **Web object scopes.** Can be defined at the application, session, or request scope. This capability makes it easy to inject, for example, a session scoped shopping cart, into your page without any lower level programming.
- **Data model management.** Provides a mechanism similar to view state to help manage your data model. (While ASP.NET manages the view state of your form, it does not offer facilities to manage the data model that you build up to submit to the application layer.)
- **UI-agnostic validation framework.** Enables you to declaratively define complex validation rules, for example, that take into account complex relationships in your data model. Spring's error controls easily render validation failure. Thus you can centralize your validation logic and also reuse it on the server side, for example, by using parameter validation advice described in the aspect library chapter.
- **Externalized page navigation through result mapping.** Instead of hard-coding URLs and data to direct where a page should go next and what data should be carried along, you can define and configure result mappings externally that associate logical names and a URL (+ data). This capability also allows you to encrypt the values that are sent through Response.Redirect.
- **Improved localization and master page support.** Provides advanced localization features (including image localization) and make it easy to declaratively configure which master page to apply to different parts of your web application.

All you know about ASP.NET development still applies. Spring's approach is to 'embrace and extend' the basic ASP.NET programming model to make you as productive as possible.

**Note**

Support for ASP.NET MVC is planned for Spring.NET 2.0.

This chapter describes the Spring.NET Web Framework in detail. The framework is not an all-or-nothing solution. For example, you can choose to use only dependency injection and bi-directional data binding. You can adopt the web framework incrementally, addressing problems areas in your current web application with a specific feature.

The Spring.NET distribution ships with a Web Quick Start application and a complete reference application, SpringAir. The Web QuickStart is the best way to learn each Spring.NET Web Framework (also referred to in this document as Spring.Web) feature, by following simple examples. The SpringAir reference application has a Spring.Web-enabled frontend that uses many best practices for Spring.NET web applications, so refer to it as you are reading this (reference) material (see Chapter 39, *SpringAir - Reference Application*).

22.2. Comparing Spring.NET and ASP.NET

Many developers dislike the ASP.NET programming model because currently it is not a "true MVC" (model-view-controller) implementation; controller-type logic within the page is too tightly coupled to the view. For example, event handlers within the page class typically have references to view elements, such as input controls, in many code behind locations, most typically the event handler. Controller-type logic, such as the code within page event handlers in ASP.NET, should not depend on the view elements.

However, ASP.NET has its good points. Server-side forms and controls make developers significantly more productive and allow you to significantly simplify page markup. They also make cross-browser issues easier to deal with, as each control can make sure that it renders correct markup based on the user's browser. The ability to hook custom logic into the lifecycle of the page, as well as to customize the HTTP processing pipeline, are also very powerful features. The ability to interact with the strongly typed server-side controls instead of manipulating string-based HTTP request collections, such as Form and QueryString, is a much needed layer of abstraction in web development.

Thus, instead of developing a new, pure and true MVC web framework as part of Spring.NET, Spring decided to extend ASP.NET so that most of its shortcomings are eliminated. With the introduction of a 'true MVC framework' to .NET there are several opportunities for integration with IoC containers such as Spring.NET. Furthermore, as Spring for Java has a very popular MVC framework, much of that experience and added value can be transliterated to help developers be more productive when using Spring's future support for ASP.NET MVC.

Spring.Web also supports the application of the dependency injection principle to one's ASP.NET `Pages` and `Controls` as well as to HTTP modules and custom provider modules. Thus application developers can easily inject service dependencies into web controllers by leveraging the power of the Spring.NET IoC container. See [Dependency Injection for ASP.NET Pages](#).

Event handlers in code-behind classes should not have to deal with ASP.NET UI controls directly. Such event handlers should rather work with the presentation model of the page, represented either as a hierarchy of domain objects or an ADO.NET `DataSet`. Spring.NET implemented a bidirectional data binding framework to handle the mapping of values to and from the controls on a page to the underlying data model. The data binding framework also transparently implements data type conversion and formatting, enabling application developers to work with fully typed data (domain) objects in the event handlers of code-behind files. See [Bidirectional Data Binding and Model Management](#).

The Spring.NET Web Framework also addresses concerns about the flow of control through an application. Typical ASP.NET applications use `Response.Redirect` Or `Server.Transfer` calls within `Page` logic to navigate

to an appropriate page after an action is executed. This usage often leads to hard-coded target URLs in the `Page`, which is never a good thing. Result mapping solves this problem by allowing application developers to specify aliases for action results that map to target URLs based on information in an external configuration file that can easily be edited. See [Result Mapping](#).

Standard localization support is also limited in versions of ASP.NET prior to ASP.NET 2.0. Even though Visual Studio 2003 generates a local resource file for each ASP.NET `Page` and user control, those resources are never used by the ASP.NET infrastructure. This means that application developers have to deal directly with resource managers whenever they need access to localized resources, which in the opinion of the Spring.NET team should not be the case. Spring.Web adds comprehensive support for localization using both local resource files and global resources that are configured within and for a Spring.NET container. See [Localization and Message Sources](#).

In addition to the aforementioned core features, Spring.Web ships with lesser features that might be useful to many application developers. Some of these additional features include back-ports of ASP.NET 2.0 features that can be used with ASP.NET 1.1, such as Master Page support. See [Master Pages in ASP.NET 1.1](#).

To implement some features, the Spring.NET team had to extend (as in the object-oriented sense) the standard ASP.NET `Page` and `UserControl` classes. This means that in order to take advantage of the *full* feature stack of Spring.Web (most notably bidirectional data binding, localization and result mapping), your code-behind classes must extend Spring.Web specific base classes such as `Spring.Web.UI.Page`. However, powerful features such as dependency injection for ASP.NET Pages, controls, and providers can be leveraged without having to extend Spring.Web-specific base classes. By taking advantage of *some* of the more useful features offered by Spring.Web, you will be coupling the presentation tier of your application(s) to Spring.Web. The choice of whether or not this is appropriate is, of course, left to you.

22.3. Automatic context loading and hierarchical contexts

22.3.1. Configuration of a web application

Spring.Web builds on top of the Spring.NET IoC container, and makes heavy use (internally) of the easy pluggability and standardized configuration afforded by the IoC container. ASP.NET `Pages` and `UserControls` that make up a typical Spring.Web-enabled application are configured with the same standard Spring.NET XML configuration syntax used for non web objects. To integrate with the ASP.NET runtime you need to make a few modifications to your `Web.config` file.

Spring.Web uses a custom `PageHandlerFactory` implementation to load and configure a Spring.NET IoC container, which is in turn used to locate an appropriate `Page` to handle a HTTP request. The `WebSupportModule` configures miscellaneous Spring infrastructure classes for use in a web environment, for example setting the storage strategy of `LogicalThreadContext` to be `HybridContextStorage`.

The instantiation and configuration of the Spring.NET IoC container by the Spring.Web infrastructure is wholly transparent to application developers, who typically never have to explicitly instantiate and configure an IoC container manually (by, for example, using the `new` operator in C#). To effect the transparent bootstrapping of the IoC container, you need to insert the following configuration snippet into the root `Web.config` file of every Spring.Web-enabled web application. (You can of course change the `verb` and `path` properties from the values that are shown.)



Note

If you are using the solution templates that ship with Spring.NET this configuration will be done for you automatically when the solution is created.

```

<system.web>
  <httpHandlers>
    <add verb="*" path="*.aspx" type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
  </httpHandlers>
  <httpModules>
    <add name="Spring" type="Spring.Context.Support.WebSupportModule, Spring.Web"/>
  </httpModules>
  ...
</system.web>

```

This snippet of standard ASP.NET configuration is only required in the *root* directory of each Spring.Web web application (that is, in the `Web.config` file present in the top level virtual directory of an ASP.NET web application).

The above XML configuration snippet directs the ASP.NET infrastructure to use Spring.NET's page factory, which in turn creates instances of the appropriate `.aspx` Page, possibly injects dependencies into that Page (as required), and then forwards the handling of the request to the Page.

After the Spring.Web page factory is configured, you also need to define a root application context by adding a Spring.NET configuration section to that same `Web.config` file. The final configuration file should resemble the following; your exact configuration may vary in particulars.

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.WebContextHandler, Spring.Web"/>
    </sectionGroup>
  </configSections>

  <spring>
    <context>
      <resource uri="~/Config/CommonObjects.xml"/>
      <resource uri="~/Config/CommonPages.xml"/>

      <!-- TEST CONFIGURATION -->
      <!--
      <resource uri="~/Config/Test/Services.xml"/>
      <resource uri="~/Config/Test/Dao.xml"/>
      -->

      <!-- PRODUCTION CONFIGURATION -->

      <resource uri="~/Config/Production/Services.xml"/>
      <resource uri="~/Config/Production/Dao.xml"/>

    </context>
  </spring>

  <system.web>
    <httpHandlers>
      <add verb="*" path="*.aspx" type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
    </httpHandlers>
    <httpModules>
      <add name="Spring" type="Spring.Context.Support.WebSupportModule, Spring.Web"/>
    </httpModules>
  </system.web>

</configuration>

```

Notes about the preceding configuration:

- Define a custom configuration section handler for the `<context>` element. If you use Spring.NET for many applications on the same web server, it might be easier to move the whole definition of the Spring.NET section group to your `machine.config` file.

- The custom configuration section handler is of the type `Spring.Context.Support.WebContextHandler` which in turn instantiates an IoC container of the type `Spring.Context.Support.WebApplicationContext`. This ensures that all features provided by `Spring.Web`, such as request and session-scoped object definitions, are handled properly.
- Within the `<spring>` element, define a root context element. Next, specify resource locations that contain the object definitions that are used within the web application (such as service or business tier objects) as child elements within the `<context>` element. Object definition resources can be fully-qualified paths or URLs, or non-qualified, as in the example above. Non-qualified resources are loaded using the default resource type for the context, which for the `WebApplicationContext` is the `WebResource` type.
- The object definition resources do not have to be the same resource type (for example, all `file://`, all `http://`, all `assembly://`, and so on). This means that you can load some object definitions from resources embedded directly within application assemblies (`assembly://`) while continuing to load other object definitions from web resources that can be more easily edited.

22.3.1.1. Configuration for IIS 7.0 on Windows Server 2008 and Windows Vista

There is some configuration that is specific to using IIS7, the appropriate code snippet to place in `web.config` shown below.

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
    <add name="Spring" type="Spring.Context.Support.WebSupportModule, Spring.Web"/>
  </modules>
  <handlers>
    <add name="SpringPageHandler" verb="*" path="*.aspx" type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
    <add name="SpringContextMonitor" verb="*" path="ContextMonitor.ashx" type="Spring.Web.Support.ContextMonitor, Spring.Web"/>
  </handlers>
</system.webServer>
```

22.3.2. Context hierarchy

ASP.NET has a hierarchical configuration mechanism that enables application developers to override configuration settings specified at a higher level in the web application directory hierarchy with configuration settings specified at the lower level.

For example, a web application's root `Web.config` file overrides settings from the (higher level) `machine.config` file. In the same fashion, settings specified within the `web.config` file within a subdirectory of a web application will override settings from the root `Web.config` and so on. You can also add settings to lower level `Web.config` files that were not previously defined anywhere.

`Spring.Web` leverages this ASP.NET feature to provide support for a context hierarchy. You can add new object definitions to lower level `Web.config` files or override existing ones per virtual directory.

What this means to application developers is that one can easily componentize an application by creating a virtual directory per component and creating a custom context for each component that contains the necessary configuration info for that particular context. The configuration for a lower level component generally contains only those definitions for the pages that the component consists of and (possibly) overrides for some definitions from the root context (for example, menus).

Because each such lower level component usually contains only a few object definitions, application developers are encouraged to embed those object definitions directly into the `Web.config` for the lower level context instead of relying on an external resource containing object definitions. This is easily accomplished by creating a component `Web.config` similar to the following one:


```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <context type="Spring.Context.Support.WebApplicationContext, Spring.Web">
      <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
      <object type="MyPage.aspx" parent="basePage">
        <property name="MyRootService" ref="myServiceDefinedInRootContext"/>
        <property name="MyLocalService" ref="myServiceDefinedLocally"/>
        <property name="Results">
          <!-- ... -->
        </property>
      </object>
      <object id="myServiceDefinedLocally" type="MyCompany.MyProject.Services.MyServiceImpl, MyAssembly"/>
    </objects>
  </spring>
</configuration>
```

The `<context/>` element seen above (contained within the `<spring/>` element) simply tells the Spring.NET infrastructure code to load (its) object definitions from the `spring/objects` section of the `web.config` configuration file.

If Spring.NET is used for multiple applications on the same server, you can avoid the need to specify the `<configSections/>` element as shown in the previous example, by moving the configuration handler definition for the `<objects>` element to a higher level (root) `web.config` file, or even to the level of the `machine.config` file.

This component-level context can reference definitions from its parent context(s). If a referenced object definition is not found in the current context, Spring.NET searches all ancestor contexts in the context hierarchy until it finds the object definition (or ultimately fails and throws an exception).

22.4. Dependency injection for ASP.NET pages

An example of how Spring.Web builds on the capabilities of ASP.NET is the way in which Spring.Web has used the code-behind class of the `Page` mechanism to satisfy the `Controller` portion of the MVC architectural pattern. In MVC-based (web) applications, the `Controller` is typically a thin wrapper around one or more service objects. It is important that service object dependencies be easily injected into `Page` Controllers. Accordingly, Spring.Web provides first class support for dependency injection in ASP.NET `Pages`. Application developers can inject any required service object dependencies (and indeed any other dependencies) into their `Pages` using the standard Spring.NET configuration instead of having to rely on custom service locators or manual object lookups in a Spring.NET application context.

After an application developer configures the Spring.NET web application context, the developer can easily create object definitions for the pages that compose that web application.

```
<objects xmlns="http://www.springframework.net">

  <object name="basePage" abstract="true">
    <property name="MasterPageFile" value="~/Web/StandardTemplate.master"/>
  </object>

  <object type="Login.aspx">
    <property name="Authenticator" ref="authenticationService"/>
  </object>
```

```
<object type="Default.aspx" parent="basePage"/>
</objects>
```

The preceding example contains three definitions:

- An abstract definition for the base page from which many other pages in the application will inherit. In this case, the definition simply specifies which page is to be referenced as the master page, but it typically also configures localization-related dependencies and root folders for images, scripts, and CSS stylesheets.
- A login page that neither inherits from the base page nor references the master page. This page shows how to inject a service object dependency into a page instance (the `authenticationService` is defined elsewhere).
- A default application page that, in this case, simply inherits from the base page in order to inherit the master page dependency, but apart from that it does not need any additional dependency injection configuration.

The configuration of ASP.NET pages differs from the configuration of other .NET classes in the value passed to the `type` attribute. As can be seen in the above configuration snippet, the `type` name is actually the path to the `.aspx` file for the `Page`, relative to its directory context. When configuring other .NET classes one would specify at minimum the fully qualified type name and the partial assembly name.

In the case of the above example, those definitions are in the root context, so `Login.aspx` and `Default.aspx` files also must be located in the root of the web application's virtual directory. The master page is defined using an absolute path because it could conceivably be referenced from child contexts that are defined within subdirectories of the web application.

The definitions for the `Login` and `Default` pages do not specify either of the `id` and `name` attributes, in marked contrast to typical object definitions in Spring.NET, where the `id` or `name` attributes are usually mandatory (although not always, as in the case of inner object definitions). In the case of Spring.Web managed `Page` instances, one typically wants to use the name of the `.aspx` file name as the identifier. If an `id` is not specified, the Spring.Web infrastructure will simply use the name of the `.aspx` file as the object identifier (minus any leading path information, and minus the file extension too).

Nothing prevents an application developer from specifying an `id` or `name` value explicitly; explicit naming can be useful when, for example, one wants to expose the same page multiple times using a slightly different configuration, such as `Add / Edit` pages. To use abstract object definitions and have your page inherit from them, use the `name` attribute instead of the `id` attribute on the abstract object definition.

22.4.1. Injecting dependencies into controls

Spring.Web also allows application developers to inject dependencies into controls (both user controls and standard controls) that are contained within a page. You can accomplish this globally for all controls of a particular `Type` by using the location of the `.ascx` as the object type identifier. This process is similar to injecting into `.aspx` pages, shown above.

```
<object type="~/controls/MyControl.ascx" abstract="true">
  <!-- inject dependencies here... -->
</object>
```



Note

In either case, be sure to mark the object definition as `abstract` (by adding `abstract="true"` to the attribute list of the `<object/>` element).

22.4.2. Injecting dependencies into custom HTTP modules

You can inject dependencies into custom HTTP modules by using the class `Spring.Context.Support.HttpApplicationConfigurer`. You register your custom HTTP module as you would normally; for example, a module of the type `HtmlCommentAppenderModule`, taken from the Web Quick Start, appends additional comments into the http response. It is registered as follows:

```
<httpModules>
  <add name="HtmlCommentAppender" type="HtmlCommentAppenderModule"/>
</httpModules>
```

To configure this module, you use naming conventions to identify the module name with configuration instructions in the Spring configuration file. The `ModuleTemplates` property of `HttpApplicationConfigurer` is a dictionary that takes as a key the name of the HTTP module, in this case `HtmlCommentAppender`, and the Spring object definition that describes how to perform dependency injection. The object definition is in the standard `<object/>` style that you are used to normally when configuring an object with Spring. An example is shown below. `HttpApplicationConfigurer` that configures the `HtmlCommentAppender`'s `AppendText` property.

```
<object name="HttpApplicationConfigurer" type="Spring.Context.Support.HttpApplicationConfigurer, Spring.Web">
  <property name="ModuleTemplates">
    <dictionary>
      <entry key="HtmlCommentAppender"> <!-- this name must match the module name -->
        <object>
          <!-- select "view source" in your browser on any page to see the appended html comment -->
          <property name="AppendText" value="My configured comment!" />
        </object>
      </entry>
    </dictionary>
  </property>
</object>
```

You can see this example in action in the Web Quick Start.

22.4.3. Injecting dependencies into HTTP handlers and handler factories

Performing dependency injection on instances of `IHttpHandlers` and `IHttpHandlerFactory` allows for a fully Spring-managed `<httpHandlers>` configuration section. To perform dependency injection on an `IHttpHandler` or `IHttpHandlerFactory`, register Spring's `MappingHandlerFactory` with a specific path or wildcard string (that is, `*.aspx`) using the standard configuration of an `<httpHandler>` in `web.config`. For example:

```
<system.web>
  <httpHandlers>
    <!--
      the lines below map *any* request ending with *.ashx or *.whatever
      to the global(!) MappingHandlerFactory. Further "specialication"
      of which handler to map to is done within MappingHandlerFactory's configuration -
      use MappingHandlerFactoryConfigurer for this (see below)
    -->
    <add verb="*" path="*.ashx" type="Spring.Web.Support.MappingHandlerFactory, Spring.Web" validate="true"/>
    <add verb="*" path="*.whatever" type="Spring.Web.Support.MappingHandlerFactory, Spring.Web" validate="false"/>
  </httpHandlers>
</system.web>
```

Spring's `MappingHandlerFactory` serves a layer of indirection so that you can configure multiple handler mappings with Spring. You do this by configuring a `IDictionary HandlerMap` property on the class `MappingHandlerFactoryConfigurer`. The dictionary key is a regular expression that matches the request URL, and the value is a reference to the name of a Spring managed instance of an `IHttpHandler` or `IHttpHandlerFactory`. The Spring managed instance is configured via dependency injection using the standard `<object/>` XML configuration schema.

The configuration of `MappingHandlerFactoryConfigurer` is shown:

```

<objects xmlns="http://www.springframework.net">

  <!-- configures the global GenericHandlerFactory instance -->
  <object name="mappingHandlerFactoryConfigurer" type="Spring.Web.Support.MappingHandlerFactoryConfigurer, Spring.Web">
    <property name="HandlerMap">
      <dictionary>
        <!-- map any request ending with *.whatever to NoOpHandler -->
        <entry key=".whatever$" value="myCustomHandler" />
        <entry key=".ashx$" value="standardHandlerFactory" />
      </dictionary>
    </property>
  </object>

  <object name="standardHandlerFactory" type="Spring.Web.Support.DefaultHandlerFactory, Spring.Web" />

  <!-- defines a standard singleton that will handle *.whatever requests -->
  <object name="myCustomHandler" type="MyCustomHttpHandler, App_Code">
    <property name="MessageText" value="This text is injected via Spring" />
  </object>

  <!--
    used for configuring ~/DemoHandler.ashx custom handler
    note, that this is an abstract definition because 'type' is not specified
  -->
  <object name="DemoHandler.ashx">
    <property name="OutputText">
      <value>This text is injected via Spring</value>
    </property>
  </object>
</objects>

```

Spring's `DefaultHandlerFactory` uses the .NET class `System.Web.UI.SimpleHandlerFactory` to create handler instances and configures each instance by using an object definition whose name matches the request URL's filename. The abstract object definition of `DemoHandler.ashx` is an example of this approach. You can also configure standard classes that implement the `IHttpHandler` interface as demonstrated in the example above for the class `MyCustomHttpHandler`.

Refer to the Web Quick Start application too see this in action.

22.4.4. Injecting dependencies in custom ASP.NET providers

Custom providers can be configured via dependency injection with Spring. The approach to configuration for providers is to use a family of adapters that correspond 1-to-1 with the standard ASP.NET providers that are registered via the standard ASP.NET mechanism. The adapters inherit from their correspondingly named provider class in the .NET class library.

- `MembershipProviderAdapter`
- `ProfileProviderAdapter`
- `RoleProviderAdapter`
- `SiteMapProviderAdapter`

Here is an example of how to register the adapter for membership providers.

```

<membership defaultProvider="mySqlMembershipProvider">
  <providers>
    <clear/>
    <add connectionStringName=" " name="mySqlMembershipProvider" type="Spring.Web.Providers.MembershipProviderAdapter" />
  </providers>
</membership>

```

The name of the provider must match the name of the object in the Spring configuration that will serve as the actual provider implementation. Configurable versions of the providers are found in ASP.NET so that you can use the

full functionality of Spring to configure these standard provider implementations, by using property placeholders, and so on. The providers are:

- ConfigurableActiveDirectoryMembershipProvider
- ConfigurableSqlMembershipProvider
- ConfigurableSqlProfileProvider
- ConfigurableSqlRoleProvider
- ConfigurableXmlSiteMapProvider

This example configuration taken from the Web Quick Start application sets the description property and connection string.

```
<object id="mySqlMembershipProvider" type="Spring.Web.Providers.ConfigurableSqlMembershipProvider">
  <property name="connectionStringName" value="MyLocalSQLServer" />
  <property name="parameters">
    <name-values>
      <add key="description" value="membershipprovider description" />
    </name-values>
  </property>
</object>
```

Your own custom providers of course will contain additional configuration specific to your implementation.

22.4.5. Customizing control dependency injection

You may need to customize Spring.Web's dependency injection processing, such as when using GridViews or other complex 3rd party custom controls. Often these controls are not configured using dependency injection but Spring considers each control and its nested child controls as candidates for DI. With very large (>1000) nested controls that candidate evaluation process can unnecessarily slow down your page. To address this problem, you can tell Spring to not attempt to configure via DI the sections of your page that contain these controls or implementing the interface [ISupportsWebDependencyInjection](#) and explicitly ask Spring to inject dependencies on a particular control. These approaches are shown below

```
[C#]
class MyControl : Control, ISupportsWebDependencyInjection
{
    private IApplicationContext _defaultApplicationContext;

    public IApplicationContext DefaultApplicationContext
    {
        get { return _defaultApplicationContext; }
        set { _defaultApplicationContext = value; }
    }

    override protected AddedControl( Control control, int index )
    {
        // handle DI for children ourselves -
        // defaults to a call to InjectDependenciesRecursive
        WebUtils.InjectDependenciesRecursive( _defaultApplicationContext, control );
        base.AddedControl( control, index );
    }
}
```

A Spring server control, [Panel](#), provides an easier way to turn off dependency injection for parts of your page:

```
<spring:Panel runat="server"
  suppressDependencyInjection="true"
  renderContainerTag="false">

  .. put your heavy controls here - they won't be touched by DI
```

```
</spring:Panel>
```

By wrapping the performance-sensitive parts of your page within this panel, you can easily turn off DI by setting the attribute `suppressDependencyInjection` to true. By default `<spring:Panel/>` will not render a container tag (`<div>`, ``, and so on). You can modify this behavior by setting the attribute `renderContainerTag` accordingly.

22.5. Web object scopes

Spring.NET web applications support an additional attribute within object definition elements that allows you to control the scope of an object:

```
<object id="myObject" type="MyType, MyAssembly" scope="application | session | request"/>
```

Possible values for the scope attribute are `application`, `session`, and `request`. Application scope is the default, and is used for all objects with an undefined scope attribute. This scope creates a single instance of an object for the duration of the IIS application, so that the objects works exactly like the standard singleton objects in non-web applications. Session scope defines objects so that an instance is created for each `HttpSession`. This scope is ideal for objects such as user profile, shopping cart, and so on that you want bound to a single user.

Request scope creates one instance per HTTP request. Unlike calls to prototype objects, calls to `IApplcationContext.GetObject` return the same instance of the request-scoped object during a single HTTP request. This allows you, for example, to inject the same request-scoped object into multiple pages and then use server-side transfer to move from one page to another. As all the pages are executed within the single HTTP request in this case, they share the same instance of the injected object.

Objects can only reference other objects that are in the same or broader scope. This means that application-scoped objects can only reference other application-scoped objects, session-scoped objects can reference both session and application-scoped objects, and request-scoped objects can reference other request-, session-, or application-scoped objects. Also, prototype objects (including all ASP.NET web pages defined within Spring.NET context) can reference singleton objects from any scope, as well as other prototype objects.

22.6. Support for ASP.NET 1.1 master pages in Spring.Web

Support for ASP.NET 1.1 master pages in Spring.Web is very similar to the support for master pages in ASP.NET 2.0.

A web developer can define a layout template for the site as a master page and specify content placeholders that other pages can then reference and populate. A sample master page (`MasterLayout.ascx`) could look like this:

```
<%@ Control language="c#" Codebehind="MasterLayout.ascx.cs" AutoEventWireup="false" Inherits="MyApp.Web.UI.MasterLayout" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>Master Page</title>
    <link rel="stylesheet" type="text/css" href="<%= Context.Request.ApplicationPath %>/css/styles.css">
    <spring:ContentPlaceHolder id="head" runat="server"/>
  </head>
  <body>
    <form runat="server">
      <table cellpadding="3" width="100%" border="1">
        <tr>
          <td colspan="2">
            <spring:ContentPlaceHolder id="title" runat="server">
              <!-- default title content -->
            </spring:ContentPlaceHolder>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

```

        <tr>
            <td>
                <spring:ContentPlaceholder id="leftSidebar" runat="server">
                    <!-- default left side content -->
                </spring:ContentPlaceholder>
            </td>
            <td>
                <spring:ContentPlaceholder id="main" runat="server">
                    <!-- default main area content -->
                </spring:ContentPlaceholder>
            </td>
        </tr>
    </table>
</form>
</body>
</html>

```

In the preceding code, the master page defines the overall layout for the page, in addition to four content placeholders that other pages can override. The master page can also include default content within the placeholder that will be displayed if a derived page does not override the placeholder.

A page that uses this master page (Child.aspx) might look like this:

```

<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<%@ Page language="c#" Codebehind="Child.aspx.cs" AutoEventWireup="false" Inherits="ArtFair.Web.UI.Forms.Child" %>
<html>
    <body>

        <spring:Content id="leftSidebarContent" contentPlaceholderId="leftSidebar" runat="server">
            <!-- left sidebar content -->
        </spring:Content>

        <spring:Content id="mainContent" contentPlaceholderId="main" runat="server">
            <!-- main area content -->
        </spring:Content>

    </body>
</html>

```

The `<spring:Content />` control in the example uses the `contentPlaceholderId` attribute (property) to specify exactly which placeholder from the master page is to be overridden. Because this particular page does not define content elements for the head and title placeholders, the content elements are defined by the default content supplied in the master page.

Both the `ContentPlaceholder` and `Content` controls can contain any valid ASP.NET markup: HTML, standard ASP.NET controls, user controls, and so on.



Tip

Technically, the `<html>` and `<body>` tags from the previous example are not strictly necessary because they are already defined in the master page. However, if these tags are omitted, Visual Studio 2003 complains about a schema, and IntelliSense does not work. So it is much easier to work in the HTML view if those tags are included. They are ignored when the page is rendered.

22.6.1. Linking child pages to their master page file

The `Spring.Web.UI.Page` class exposes a property called `MasterPageFile`, which you can use to specify the master page.

The recommended way to do this is by leveraging the Spring.NET IoC container and creating definitions similar to the following:

```

<?xml version="1.0" encoding="utf-8" ?>

```



```
<objects xmlns="http://www.springframework.net">

  <object name="basePage" abstract="true">
    <property name="MasterPageFile" value="~/MasterLayout.ascx"/>
  </object>

  <object type="Child.aspx" parent="basePage">
    <!-- inject other objects that page needs -->
  </object>

</objects>
```

This approach allows application developers to change the master page for a number of pages within a web application. You can still override the master page on a per context or per page basis by creating a new abstract page definition within a child context, or by specifying the `MasterPageFile` property directly.

22.7. Bidirectional data binding and data model management

The existing data binding support in ASP.NET is one-way only. It allows application developers to bind page controls to the data model and display information from the data model, but it does not permit the extraction of values from the controls when the form is submitted. Spring.Web adds bidirectional data binding to ASP.NET by allowing developers to specify data binding rules for their page, and by automatically evaluating configured data binding rules at the appropriate time in the page's lifecycle.

ASP.NET does support model management within the postbacks. It has a `ViewState` management, but that takes care of the control state only and does not address the state of any presentation model objects to which these controls are bound. To manage a model within ASP.NET, developers typically use an HTTP session object to store the model between the postbacks. This process results in boilerplate code that can and should be eliminated, which is exactly what Spring.Web does by providing a simple set of model management methods.

To take advantage of the bidirectional data binding and model management support provided by Spring.Web, you *will* have to couple your presentation layer to Spring.Web; this is because features *require* you to extend a `Spring.Web.UI.Page` instead of the usual `System.Web.UI.Page` class.

Spring.Web data binding is very easy to use. Simply override the protected `InitializeDataBindings` method and configure data binding rules for the page. You also need to override three model management methods: `InitializeModel`, `LoadModel` and `SaveModel`. This process is illustrated by an example from the SpringAir reference application. First, take a look at the page markup:

```
<%@ Page Language="c#" Inherits="TripForm" CodeFile="TripForm.aspx.cs" %>

<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
  <div style="text-align: center">
    <h4><asp:Label ID="caption" runat="server"></asp:Label></h4>
    <table>
      <tr class="formLabel">
        <td>&nbsp;</td>
        <td colspan="3">
          <spring:RadioButtonGroup ID="tripMode" runat="server">
            <asp:RadioButton ID="OneWay" runat="server" />
            <asp:RadioButton ID="RoundTrip" runat="server" />
          </spring:RadioButtonGroup>
        </td>
      </tr>
      <tr>
        <td class="formLabel" align="right">
          <asp:Label ID="leavingFrom" runat="server" /></td>
        <td nowrap="nowrap">
          <asp:DropDownList ID="leavingFromAirportCode" runat="server" />
        </td>
        <td class="formLabel" align="right">
          <asp:Label ID="goingTo" runat="server" /></td>
      </tr>
    </table>
  </div>
</asp:Content>
```



```

        <td nowrap="nowrap">
            <asp:DropDownList ID="goingToAirportCode" runat="server" />
        </td>
    </tr>
    <tr>
        <td class="formLabel" align="right">
            <asp:Label ID="leavingOn" runat="server" /></td>
        <td nowrap="nowrap">
            <spring:Calendar ID="departureDate" runat="server" Width="75px" AllowEditing="true" Skin="system" />
        </td>
        <td class="formLabel" align="right">
            <asp:Label ID="returningOn" runat="server" /></td>
        <td nowrap="nowrap">
            <div id="returningOnCalendar">
                <spring:Calendar ID="returnDate" runat="server" Width="75px" AllowEditing="true" Skin="system" />
            </div>
        </td>
    </tr>
    <tr>
        <td class="buttonBar" colspan="4">
            <br/>
            <asp:Button ID="findFlights" runat="server"/></td>
        </tr>
    </table>
</div>

</asp:Content>

```

Ignore for the moment the fact that none of the label controls have text defined; defining label controls is described later when we discuss localization in Spring.NET. For the purposes of the current discussion, a number of input controls are defined: tripMode radio button group, leavingFromAirportCode and goingToAirportCode dropdown lists, as well as two Spring.NET Calendar controls, departureDate and returnDate.

Take a look at the model to which you bind the form:

```

namespace SpringAir.Domain
{
    [Serializable]
    public class Trip
    {
        // fields
        private TripMode mode;
        private TripPoint startingFrom;
        private TripPoint returningFrom;

        // constructors
        public Trip()
        {
            this.mode = TripMode.RoundTrip;
            this.startingFrom = new TripPoint();
            this.returningFrom = new TripPoint();
        }

        public Trip(TripMode mode, TripPoint startingFrom, TripPoint returningFrom)
        {
            this.mode = mode;
            this.startingFrom = startingFrom;
            this.returningFrom = returningFrom;
        }

        // properties
        public TripMode Mode
        {
            get { return this.mode; }
            set { this.mode = value; }
        }

        public TripPoint StartingFrom
        {
            get { return this.startingFrom; }
            set { this.startingFrom = value; }
        }
    }
}

```

```

    }

    public TripPoint ReturningFrom
    {
        get { return this.returningFrom; }
        set { this.returningFrom = value; }
    }
}

[Serializable]
public class TripPoint
{
    // fields
    private string airportCode;
    private DateTime date;

    // constructors
    public TripPoint()
    {}

    public TripPoint(string airportCode, DateTime date)
    {
        this.airportCode = airportCode;
        this.date = date;
    }

    // properties
    public string AirportCode
    {
        get { return this.airportCode; }
        set { this.airportCode = value; }
    }

    public DateTime Date
    {
        get { return this.date; }
        set { this.date = value; }
    }
}

[Serializable]
public enum TripMode
{
    OneWay,
    RoundTrip
}
}

```

As you can see, Trip class uses the TripPoint class to represent departure and return, which are exposed as StartingFrom and ReturningFrom properties. It also uses TripMode enumeration to specify whether the trip is one way or return trip, which is exposed as Mode property.

Here is the code-behind class that ties everything together:

```

public class TripForm : Spring.Web.UI.Page
{
    // model
    private Trip trip;
    public Trip Trip
    {
        get { return trip; }
        set { trip = value; }
    }

    // service dependency, injected by Spring IoC container
    private IBookingAgent bookingAgent;
    public IBookingAgent BookingAgent
    {
        set { bookingAgent = value; }
    }

    // model management methods
    protected override void InitializeModel()

```

```

{
    trip = new Trip();
    trip.Mode = TripMode.RoundTrip;
    trip.StartingFrom.Date = DateTime.Today;
    trip.ReturningFrom.Date = DateTime.Today.AddDays(1);
}

protected override void LoadModel(object savedModel)
{
    trip = (Trip) savedModel;
}

protected override object SaveModel()
{
    return trip;
}

// data binding rules
protected override void InitializeDataBindings()
{
    BindingManager.AddBinding("tripMode.Value", "Trip.Mode");
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue", "Trip.StartingFrom.AirportCode");
    BindingManager.AddBinding("goingToAirportCode.SelectedValue", "Trip.ReturningFrom.AirportCode");
    BindingManager.AddBinding("departureDate.SelectedDate", "Trip.StartingFrom.Date");
    BindingManager.AddBinding("returnDate.SelectedDate", "Trip.ReturningFrom.Date");
}

// event handler for findFlights button, uses injected 'bookingAgent'
// service and model 'trip' object to find flights
private void SearchForFlights(object sender, EventArgs e)
{
    FlightSuggestions suggestions = bookingAgent.SuggestFlights(trip);
    if (suggestions.HasOutboundFlights)
    {
        // redirect to SuggestedFlights page
    }
}
}

```

Note the following about the three preceding pieces of code:

1. When the page is initially loaded (`IsPostBack == false`), the `InitializeModel()` method is called, which initializes the trip object by creating a new instance and setting its properties to desired values. Right before the page is rendered, the `SaveModel()` method is invoked, and the value it returns is stored within the HTTP session. On each postback, the `LoadModel()` method is called, and the value returned by the previous call to `SaveModel` is passed to `SaveModel` as an argument.

In this particular case the implementation is very simple because our whole model is just the `trip` object. As such, `SaveModel()` simply returns the `trip` object, and `LoadModel()` casts the `SaveModel()` argument to `Trip` and assigns it to the `trip` field within the page. In more complex scenarios, the `SaveModel()` method will typically return a dictionary that contains your model objects. Those values will be read from the dictionary within the `LoadModel()` method.

2. `InitializeDataBindings` method defines the binding rules for all of the five input controls on the form. The controls are represented by the variables `tripMode`, `leavingFromAirportCode`, `goingToAirportCode`, `departureDate`, and `returnDate`. The binding rules are created by invoking the `AddBinding` method on the `BindingManager` exposed by the page. The `AddBinding` method is heavily overloaded and it allows you to specify a *binding direction* and a *formatter* to use in addition to the *source and target binding expressions* that are used above. These optional parameters are discussed later in this chapter. For now, focus on the source and target expressions.

The Spring.NET data binding framework uses Spring.NET Expression Language to define binding expressions. In most cases, as in the example above, both source and target expression will evaluate to a property or a field within one of the controls or a data model. This is always the case when you are setting a

bidirectional binding, as both binding expressions need to be "settable". The `InitializeDataBindings` method is executed only once *per page type*. Basically, all binding expressions are parsed the first time the page is instantiated, and are then cached and used by all instances of that same page type that are created at a later time. This is done for performance reasons, as data binding expression parsing on every postback is unnecessary and would add a significant overhead to the overall page processing time.

3. Notice that the `SearchForFlights` event handler has no dependencies on the view elements. It simply uses the injected `bookingAgent` service and a trip object in order to obtain a list of suggested flights. Furthermore, if you make any modifications to the trip object within your event handler, bound controls are updated accordingly just before the page is rendered.



Note

The lack of view elements in the event handler accomplishes one of the major goals we set out to achieve, allowing developers to remove view element references from the page event handlers and decouple controller-type methods from the view.

22.7.1. Data binding under the hood

This section describes how data binding is actually implemented, the extension points, and additional features that make the data binding framework usable in real-world applications.

The Spring.NET data binding framework revolves around two main interfaces: `IBinding` and `IBindingContainer`. The `IBinding` interface is definitely the more important one of the two, as it has to be implemented by all binding types. This interface defines several methods, with some of them being overloaded for convenience:

```
public interface IBinding
{
    void BindSourceToTarget(object source, object target, ValidationErrors validationErrors);

    void BindSourceToTarget(object source, object target, ValidationErrors validationErrors,
        IDictionary variables);

    void BindTargetToSource(object source, object target, ValidationErrors validationErrors);

    void BindTargetToSource(object source, object target, ValidationErrors validationErrors,
        IDictionary variables);

    void SetErrorMessage(string messageId, params string[] errorProviders);
}
```

The `BindSourceToTarget` method is used to extract and copy bound values from the source object to the target object, and `BindTargetToSource` does the opposite. Both method names and parameter types are generic because the data binding framework can be used to bind any two objects. Using it to bind web forms to model objects is just one of its possible uses, although a very common one and tightly integrated into the Spring.NET Web Framework.

The `ValidationErrors` parameter requires further explanation. Although the data binding framework is not in any way coupled to the data validation framework, they are in some ways related. For example, while the data validation framework is best suited to validate the populated model according to the business rules, the data binding framework is in a better position to validate data types during the binding process. However, regardless of where specific validation is performed, all error messages should be presented to the user in a consistent manner. In order to accomplish this, Spring.NET Web Framework passes the same `ValidationErrors` instance to binding methods and to any validators that might be executed within your event handlers. This process ensures that all error messages are stored together and are displayed consistently to the end user, using Spring.NET validation error controls.

The last method in the `IBinding` interface, `SetErrorMessage`, enables you to specify the resource id of the error message to be displayed in case of binding error, as well as a list of strings, that server as identifiers to tag error messages for the purposes of linking specific error messages to locations in the page markup. We will see an example of the `SetErrorMessage` usage in a later section.

The `IBindingContainer` interface extends the `IBinding` interface and adds the following members:

```
public interface IBindingContainer : IBinding
{
    bool HasBindings { get; }

    IBinding AddBinding(IBinding binding);
    IBinding AddBinding(string sourceExpression, string targetExpression);
    IBinding AddBinding(string sourceExpression, string targetExpression, BindingDirection direction);
    IBinding AddBinding(string sourceExpression, string targetExpression, IFormatter formatter);
    IBinding AddBinding(string sourceExpression, string targetExpression, BindingDirection direction,
        IFormatter formatter);
}
```

The `IBindingContainer` interface has several overloaded `AddBinding` methods. `AddBinding(IBinding binding)` is the most generic one, as it can be used to add any binding type to the container. The other four are convenience methods that provide a simple way to add the most commonly used implementation of the `IBinding` interface, `SimpleExpressionBinding`. The `SimpleExpressionBinding` was used under the covers in the example at the beginning of this section to bind our web form to a `Trip` instance when calling methods on the property `BindingManager`. Note that the `BindingManager` property is of the type `IBindingContainer`. `SimpleExpressionBinding` uses Spring.NET Expression Language (SpEL) to extract and to set values within source and target objects.

In the `TripForm` example, the configuration of the `BindingManager` shows the basic usage of how SpEL can be used to specify a `sourceExpression` and `targetExpression` arguments. This code section is repeated below

```
protected override void InitializeDataBindings()
{
    BindingManager.AddBinding("tripMode.Value", "Trip.Mode");
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue", "Trip.StartingFrom.AirportCode");
    BindingManager.AddBinding("goingToAirportCode.SelectedValue", "Trip.ReturningFrom.AirportCode");
    BindingManager.AddBinding("departureDate.SelectedDate", "Trip.StartingFrom.Date");
    BindingManager.AddBinding("returnDate.SelectedDate", "Trip.ReturningFrom.Date");
}
```

In this case, the first argument is a `sourceExpression` evaluated in the context of the page itself. The `sourceExpression` `'tripMode.Value'` represents the value in the HTML control and the `targetExpression` `"Trip.Mode"` represents the value it will be mapped onto when the page is rendered. When the post-back happens values from in `"Trip.Mode"` get placed back into the HTML control `"tripMode.Value"`. This is a common case in which bi-directional data mapping is symmetric in terms of the `sourceExpression` and `targetExpression` for both the initial rendering of the page and when the post-back occurs. There other overloaded methods that take `BindingDirection` and `IFormatter` arguments are discussed in the next section.

22.7.1.1. Binding direction

The `direction` argument determines whether the binding is bidirectional or unidirectional. By default, all data bindings are bidirectional unless the `direction` argument is set to either `BindingDirection.SourceToTarget` or `BindingDirection.TargetToSource`. If one of these values is specified, binding is evaluated only when the appropriate `BindDirection` method is invoked, and is completely ignored in the other direction. This configuration is very useful when you want to bind some information from the model into non-input controls, such as labels.

However, unidirectional data bindings are also useful when your form does not have a simple one-to-one mapping to a presentation model. In the earlier trip form example, the presentation model was intentionally designed to

allow for simple one-to-one mappings. For the sake of discussion, let's add the `Airport` class and modify our `TripPoint` class as follows:

```
namespace SpringAir.Domain
{
    [Serializable]
    public class TripPoint
    {
        // fields
        private Airport airport;
        private DateTime date;

        // constructors
        public TripPoint()
        {}

        public TripPoint(Airport airport, DateTime date)
        {
            this.airport = airport;
            this.date = date;
        }

        // properties
        public Airport Airport
        {
            get { return this.airport; }
            set { this.airport = value; }
        }

        public DateTime Date
        {
            get { return this.date; }
            set { this.date = value; }
        }
    }

    [Serializable]
    public class Airport
    {
        // fields
        private string code;
        private string name;

        // properties
        public string Code
        {
            get { return this.code; }
            set { this.code = value; }
        }

        public string Name
        {
            get { return this.name; }
            set { this.name = value; }
        }
    }
}
```

Instead of the string property `AirportCode`, our `TripPoint` class now exposes an `Airport` property of type `Airport`, which is defined in the preceding example. What was formerly a simple string-to-string binding, with the airport code selected in a dropdown being copied directly into the `TripPoint.AirportCode` property and vice versa, now becomes a not-so-simple string-to-`Airport` binding. So let's see how we can solve this mismatch problem of converting a string to an `Airport` instance and an `Airport` instance to a string.

Binding from the model to the control, namely the `Airport` to the string, is still very straightforward. You set up one-way bindings from the model to controls: The Model-To-Control is represented more generally by the enumeration, `BindingDirection.TargetToSource`.

```
protected override void InitializeDataBindings()
```

```

{
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue", "Trip.StartingFrom.Airport.Code", BindingDirection.
    BindingManager.AddBinding("goingToAirportCode.SelectedValue", "Trip.ReturningFrom.Airport.Code", BindingDirection.Ta
    ...
}

```

You extract the airport code value from the `Trip.StartingFrom.Airport.Code` instead of `Trip.StartingFrom.AirportCode` since now the `Code` is encapsulated inside the `Airport` class. Unfortunately, binding from the control to the model the same way won't work, we need a way to create an `Airport` instance from a string. Instead, you need to find an instance of the `Airport` class based on the airport code and set the `TripPoint.Airport` property to it. Fortunately, Spring.NET data binding makes this simple, especially because you already have `airportDao` object defined in the Spring context (see `SpringAir` Spring context configuration file for details.). The `AirportDao` has a `GetAirport(string airportCode)` finder method. You set up data bindings from source to target (control-to-model) that will invoke this finder method when the page is submitted and the binding infrastructure maps the `sourceExpression` onto the `targetExpression`.evaluating the source expression.

Our complete set of bindings for these two drop-down lists will then look like this:

```

protected override void InitializeDataBindings()
{
    BindingManager.AddBinding("@(airportDao).GetAirport(leavingFromAirportCode.SelectedValue)", "Trip.StartingFrom.Airport
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue", "Trip.StartingFrom.Airport.Code", BindingDirection

    BindingManager.AddBinding("@(airportDao).GetAirport(goingToAirportCode.SelectedValue)", "Trip.ReturningFrom.Airport"
    BindingManager.AddBinding("goingToAirportCode.SelectedValue", "Trip.ReturningFrom.Airport.Code", BindingDirection.Ta
    ...
}

```

By using a pair of bindings for each control, one for each direction and using SpEL's feature to reference objects defined in the Spring context, you can resolve this data binding issue.

22.7.1.2. formatter argument

The last overloaded methods of `IBindingContainer` we need to discuss are those that take a `IFormatter` argument. is an argument to the `AddBinding` method. This argument allows you to specify a formatter that you use to parse string value from the input control before it is bound to the model, and to format strongly typed model value before the model is bound to the control.

You typically use one of the formatters provided in the `Spring.Globalization.Formatters` namespace, but if your requirements cannot be satisfied by a standard formatter, you can write your own by implementing a simple `IFormatter` interface:

```

public interface IFormatter
{
    string Format(object value);
    object Parse(string value);
}

```

Standard formatters provided with Spring.NET are: `CurrencyFormatter`, `DateTimeFormatter`, `FloatFormatter`, `IntegerFormatter`, `NumberFormatter` and `PercentFormatter`, which are sufficient for most usage scenarios.

22.7.1.3. Type conversion

Because the data binding framework uses the same expression evaluation engine as the Spring.NET IoC container, it uses any registered type converters to perform data binding. Many type converters are included with Spring.NET (take a look at the classes in `Spring.Objects.TypeConverters` namespace) and are automatically registered for you, but you can implement your own custom converters and register them by using standard Spring.NET type converter registration mechanisms.

22.7.1.4. Data binding events

Spring.Web's base `Page` class adds two events to the standard .NET page lifecycle: `DataBound` and `DataUnbound`.

You can register for an `DataUnbound` event which will be fired after the data model is updated with values from the controls. Specifically, in terms of the Page lifecycle, it is fired right after the `Load` event and only on postbacks, because it not make sense to update the data model with the controls' initial values.

The `DataBound` event is fired after controls are updated with values from the data model. This event occurs right before the `PreRender` event.

The fact that the data model is updated immediately after the `Load` event and that controls are updated right before the `PreRender` event means that your event handlers can work with a correctly updated data model, as they execute after the `Load` event, and that any changes you make to the data model within event handlers are reflected in the controls immediately afterwards, as the controls are updated prior to the actual rendering.

22.7.1.5. Rendering binding errors

If errors occur in the databinding (for example, in trying to bind a string 'hello' to an integer property on the model), you can specify how those fundamental binding errors should be rendered. The following snippet is from the Web Quick Start 'RobustEmployeeInfo' example:

```
[Default.aspx.cs]

protected override void InitializeDataBindings()
{
    // collect txtId.Text binding errors in "id.errors" collection
    BindingManager.AddBinding("txtId.Text", "Employee.Id").SetErrorMessage("ID has to be an integer", "id.errors");
    ...

[Default.aspx]
...
<asp:TextBox ID="txtId" runat="server" />
<!-- output validation errors from "id.errors" collection -->
<spring:ValidationError Provider="id.errors" runat="server" />
...
```

The `SetErrorMessage` specifies the message text or resource id of the error message to be displayed. This is followed by a variable length list of strings that serve to as a means to assign a friendly name to associate with this error should it occur. The same 'tag', or error provider name, can be used across different calls to 'AddBinding'. This is commonly the case if you want to present several errors together in the page. In the preceding example, the 'tag' or error provider name is "id.errors" will be rendered in Spring's `ValidationError` User Control, for example as shown below in this fragment of page markup. Validation controls are discussed more extensively in this section.

```
<td>
    <asp:TextBox ID="txtId" runat="server" EnableViewState="false" />
    <spring:ValidationError ID="errId" Provider="id.errors" runat="server" /><!-- read msg from "id.error" provi
</td>
```

22.7.1.6. HttpRequestListBindingContainer

[HttpRequestListBindingContainer](#) extracts posted raw values from the request and populates the specified `IList` by creating objects of the type specified and populating each object according to the `requestBindings` collection.

Please check out the Web Quick Start sample's demo of [HttpRequestListBindingContainer](#). Below is an excerpt from that example showing how to use a [HttpRequestListBindingContainer](#).

```
protected override void InitializeDataBindings()
{
    // HttpRequestListBindingContainer unbinds specified values from Request -> Productlist
    HttpRequestListBindingContainer requestBindings =
```



```

new HttpRequestListBindingContainer("sku,name,quantity,price", "Products", typeof(ProductInfo));
requestBindings.AddBinding("sku", "Sku");
requestBindings.AddBinding("name", "Name");
requestBindings.AddBinding("quantity", "Quantity", quantityFormatter);
requestBindings.AddBinding("price", "Price", priceFormatter);

BindingManager.AddBinding(requestBindings);
}

```



Note

Because browsers do not send the values of unchecked checkboxes, you cannot use `HttpRequestListBindingContainer` with `<input type="checkbox">` html controls.

22.7.2. Using DataBindingPanel

To simplify use of Spring's Data Binding feature on web pages and controls, Spring.Web provides a special `DataBindingPanel` container control. A `DataBindingPanel` does not render any html code itself, but allows you to define additional, data binding-related attributes for its child controls.

```

<%@ Page Language="C#" CodeFile="Default.aspx.cs" Inherits="DataBinding_EasyEmployeeInfo_Default" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<html>
<body>
<spring:DataBindingPanel ID="ctlDataBindingPanel" runat="server">
  <table cellpadding="3" cellspacing="3" border="0">
    <tr>
      <td>Employee ID:</td>
      <td>
        <asp:TextBox ID="txtId" runat="server" BindingTarget="Employee.Id" />
      </td>
    </tr>
    <tr>
      <td>First Name:</td>
      <td><asp:TextBox ID="txtFirstName" runat="server" BindingTarget="Employee.FirstName" /></td>
    </tr>
  </table>
</spring:DataBindingPanel>
</body>
</html>

```

Using `DataBindingPanel`, you can specify the binding information directly on the control declaration. The following attributes are recognized by a `DataBindingPanel`:

- `BindingTarget` corresponds to the target expression used in `IBindingContainer.AddBinding()`.
- `BindingSource` corresponds to the source expression used in `IBindingContainer.AddBinding()`. For standard controls you don't need to specify the source expression. If you are binding to some custom control, of course you must specify this attribute.
- `BindingDirection` is one of the values of the `BindingDirection` enumeration.
- `BindingFormatter` is the object name of a custom formatter. The formatter instance is obtained by a call to `IApplicationContext.GetObject()` each time it is needed.
- `BindingType` is the type of a completely customized binding. Note that a custom binding type must implement the following constructor signature:

```
ctor(string source, string target, BindingDirection, IFormatter)
```



Note

The Visual Studio Web Form Editor complains about binding attributes because it does not recognize them. You can safely ignore those warnings.

22.7.3. Customizing model persistence

As mentioned in the chapter introduction, model management needs an application developer to override `InitializeModel()`, `SaveModel()` and `LoadModel()` in order to store model information between requests in the user's session. On web farms, storing information in a user's session is not a good strategy. You can choose another persistence strategy by setting the `ModelPersistenceMedium` property on Spring's base `Page` or `UserControl` class (e.g. `Spring.Web.UI.UserControl`)

```
<object id="modelPersister" type="Sample.DatabaseModelPersistenceMedium, MyCode"/>

<object type="UserRegistration.aspx">
  <property name="ModelPersistenceMedium" ref="modelPersister"/>
</object>
```

To implement any arbitrary persistence strategy, implement the `IModelPersistenceMedium` interface:

```
public interface IModelPersistenceMedium
{
    // Load the model for the specified control context.
    object LoadFromMedium( Control context );

    // Save the specified model object.
    void SaveToMedium( Control context, object modelToSave );
}
```

22.8. Localization and message sources

Although the .NET framework has excellent localization support, the support within ASP.NET 1.x is incomplete. Spring provides support for localization in ASP.NET 1.1 apps in the manner of ASP.NET 2.0. Despite the initial focus on richer localization for ASP.NET 1.1 applications, using Spring's localization features in ASP.NET 2.0 or higher applications does provide some useful additional features with a similar programming model, such as image localization, push mechanisms, and built-in support for user culture management via various mechanisms.

Every `.aspx` page in an ASP.NET project has a resource file associated with it, but those resources are never used by the current ASP.NET infrastructure). ASP.NET 2.0 changes this and allow application developers to use local resources for pages. In the meantime, the Spring.NET team built in to Spring.Web support for using local pages resources, thus allowing ASP.NET 1.1 application developers to using ASP.NET 2.0-like page resources.

Spring.Web supports several different approaches to localization within a web application, which can be mixed and matched as appropriate. You can use push and pull mechanisms, as well as globally defined resources when a local resource cannot be found. Spring.Web also supports user culture management and image localization, which are described in later sections.



Tip

For introductory information covering ASP.NET globalization and localization, see [Globalization Architecture for ASP.NET](#) and [Localization Practices for ASP.NET 2.0](#) by Michele Leroux Bustamante.

22.8.1. Working with localizers

A localizer is an object that implements the `Spring.Globalization.ILocalizer` interface. `Spring.Globalization.AbstractLocalizer` is a convenient base class for localization: this class has one abstract method, `LoadResources`. This method must load and return a list of all resources that must be automatically applied from the resource store.

To apply resources automatically, a localizer needs to be injected into all pages that require automatic resource application. You typically accomplish configuration using dependency injection of a page base page definition that other page definitions will inherit from. The injected localizer inspects the resource file when the page is first requested, caches the resources that start with the '\$this' marker string value, and applies the values to the controls that populate the page prior to the page being rendered.

Spring.NET ships with one concrete implementation of a localizer, `Spring.Globalization.Localizers.ResourceSetLocalizer`, that retrieves a list of resources to apply from the local resource file. Future releases of Spring.NET may provide other localizers that read resources from an XML file or even from a flat text file that contains resource name-value pairs that allow application developers to store resources within the files in a web application instead of as embedded resources in an assembly. Of course, if an application developer prefers to store such resources in a database, the developer can write a custom `ILocalizer` implementation that loads a list of resources to apply from a database.

You typically configure the localizer to be used within an abstract base definition for those pages that require localization:

```
<object id="localizer" type="Spring.Globalization.Localizers.ResourceSetLocalizer, Spring.Core"/>

<object name="basePage" abstract="true">
  <description>
    Pages that reference this definition as their parent
    (see examples below) will automatically inherit following properties.
  </description>
  <property name="Localizer" ref="localizer"/>
</object>
```

Of course, nothing prevents an application developer from defining a different localizer for each page in the application; in any case, one can always override the localizer defined in a base (page) definition. Alternatively, if one does want any resources to be applied automatically one can completely omit the localizer definition.

One last thing to note is that Spring.NET `UserControl` instances will (by default) inherit the localizer and other localization settings from the page that they are contained within, but one can similarly also override that behavior using explicit dependency injection.

22.8.2. Automatic localization with localizers ("push" localization)

With push localization, an application developer specifies localization resources in the resource file for the page, and the framework automatically applies those resources to the user controls on the page. For example, an application developer could define a page such as `UserRegistration.aspx`:

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<%@ Page Language="c#" Codebehind="UserRegistration.aspx.cs"
    AutoEventWireup="false" Inherits="ArtFair.Web.UI.Forms.UserRegistration" %>
<html>
  <body>
    <spring:Content id="mainContent" contentPlaceholderId="main" runat="server">
      <div align="right">
        <asp:LinkButton ID="english" Runat="server" CommandArgument="en-US">English</asp:LinkButton>&nbsp;
        <asp:LinkButton ID="serbian" Runat="server" CommandArgument="sr-SP-Latn">Srpski</asp:LinkButton>
      </div>
      <table>
        <tr>
          <td><asp:Label id="emailLabel" Runat="server"/></td>
          <td><asp:TextBox id="email" Runat="server" Width="150px"/></td>
        </tr>
        <tr>
          <td><asp:Label id="passwordLabel" Runat="server"/></td>
          <td><asp:TextBox id="password" Runat="server" Width="150px"/></td>
        </tr>
      </table>
    </spring:Content>
  </body>
</html>
```

```

        <td><asp:Label id="passwordConfirmationLabel" Runat="server"/></td>
        <td><asp:TextBox id="passwordConfirmation" Runat="server" Width="150px"/></td>
    </tr>
    <tr>
        <td><asp:Label id="nameLabel" Runat="server"/></td>
        <td><asp:TextBox id="name" Runat="server" Width="150px"/></td>
    </tr>
    ...

    <tr>
        <td colspan="2">
            <asp:Button id="saveButton" Runat="server"/>&nbsp;  
            <asp:Button id="cancelButton" Runat="server"/>
        </td>
    </tr>
</table>
</spring:Content>
</body>
</html>

```

In the preceding .aspx code, none of the `Label` or `Button` controls have had a value assigned to the `Text` property. The values of the `Text` property for these controls are stored in the local resource file (of the page) using the following convention to identify the resource (string).

```
$this.controlId.propertyName
```

The corresponding local resource file, `UserRegistration.aspx.resx`, is shown below.

```

<root>
  <data name="$this.emailLabel.Text">
    <value>Email:</value>
  </data>
  <data name="$this.passwordLabel.Text">
    <value>Password:</value>
  </data>
  <data name="$this.passwordConfirmationLabel.Text">
    <value>Confirm password:</value>
  </data>
  <data name="$this.nameLabel.Text">
    <value>Full name:</value>
  </data>
  ...

  <data name="$this.countryLabel.Text">
    <value>Country:</value>
  </data>
  <data name="$this.saveButton.Text">
    <value>$messageSource.save</value>
  </data>
  <data name="$this.cancelButton.Text">
    <value>$messageSource.cancel</value>
  </data>
</root>

```



Viewing .resx file in Visual Studio 2003

To view the .resx file for a page, you may need to enable "Project/Show All Files" in Visual Studio. When "Show All Files" is enabled, the .resx file appears like a "child" of the code-behind page.

When Visual Studio creates the .resx file, it includes an `xds:schema` element and several `reshead` elements. Your data elements will follow the `reshead` elements. When working with the .resx files, you may want to choose "Open With" from the context menu and select the "Source Code" text editor.

There is no way to visually edit resources in a RESX file. Lutz Roeder has created a tool named **Resourcer** [<http://www.lutzroeder.com/dotnet/>] that you can use to edit them



Creating a .resx file in Visual Studio 2005/8

To create a resource file in VS 2005, open your control or page in design mode and select "Tools/Generate local resource" from the menu.

You must create a localizer for the page to enable automatic localization:

```
<object id="localizer" type="Spring.Globalization.Localizers.ResourceSetLocalizer, Spring.Core"/>

<object type="UserRegistration.aspx">
  <property name="Localizer" ref="localizer"/>
</object>
```

For more information on configuring localizers see Section 22.8.1, "Working with localizers"

22.8.3. Global message sources

Two resource definitions from the previous section require some additional explanation:

```
<data name="$this.saveButton.Text">
  <value>$messageSource.save</value>
</data>
<data name="$this.cancelButton.Text">
  <value>$messageSource.cancel</value>
</data>
```

In some cases it makes sense to apply a resource that is defined *globally* as opposed to locally. In this example, it makes better sense to define values for the `Save` and `Cancel` buttons globally as they will probably be used throughout the application.

The above example demonstrates how one can achieve that by defining a *resource redirection* expression as the value of a local resource by prefixing a global resource name with the following string.

```
$messageSource.
```

In the preceding example, this string tells the localizer to use the `save` and `cancel` portions of the resource key as lookup keys to retrieve the actual values from a global message source. You need to define a resource redirect only once, typically in the invariant resource file. Any lookup for a resource redirect falls back to the invariant culture, and results in a global message source lookup using the correct culture.

Global resources are (on a per-context basis) defined as a plain vanilla object definition using the reserved name of `messageSource`, which you can add to your Spring.NET configuration file:

```
<object id="messageSource" type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
  <property name="ResourceManagers">
    <list>
      <value>MyApp.Web.Resources.Strings, MyApp.Web</value>
    </list>
  </property>
</object>
```



for .NET 2.0 or higher

To use resources from your `App_GlobalResources` folder, specify `App_GlobalResources` as the assembly name:

```
<value>Resources.Strings, App_GlobalResources</value>
```

See the SpringAir example application for more. The global resources are cached within the Spring.NET `IApplicationContext` and are accessible through the Spring.NET `IMessageSource` interface.

The `Spring.Web.Page` and `UserControl` classes have a reference to their owning `IApplicationContext` and its associated `IMessageSource`. As such, they automatically redirect resource lookups to a global message source if a local resource cannot be found.

Currently, the `ResourceSetMessageSource` is the only message source implementation that ships with Spring.NET.

22.8.4. Applying resources manually ("pull" localization)

Although automatic localization as described above works well for many form-like pages, it doesn't work nearly as well for controls defined within any iterative controls, because the IDs for such iterative controls are not fixed. Nor does automatic localization work well if you need to display the same resource multiple times within the same page. For example, think of the header columns for outgoing and return flights tables within the *SpringAir* application (see Chapter 39, *SpringAir - Reference Application*).

These situations call for a pull-style mechanism for localization, which is a simple `GetMessage` call:

```
<asp:Repeater id="outboundFlightList" Runat="server">
  <HeaderTemplate>
    <table border="0" width="90%" cellpadding="0" cellspacing="0" align="center" class="suggestedTable">
      <thead>
        <tr class="suggestedTableCaption">
          <th colspan="6">
            <%= GetMessage("outboundFlights") %>
          </th>
        </tr>
        <tr class="suggestedTableColnames">
          <th><%= GetMessage("flightNumber") %></th>
          <th><%= GetMessage("departureDate") %></th>
          <th><%= GetMessage("departureAirport") %></th>
          <th><%= GetMessage("destinationAirport") %></th>
          <th><%= GetMessage("aircraft") %></th>
          <th><%= GetMessage("seatPlan") %></th>
        </tr>
      </thead>
      <tbody>
    </HeaderTemplate>
```

The `GetMessage` method is available within both the `Spring.Web.UI.Page` and `Spring.Web.UI.UserControl` classes, and it falls back automatically to a global message source lookup if a local resource is not found.

22.8.5. Localizing images within a web application

Unlike text resources, which can be stored within embedded resource files, XML files, or even a database, images in a typical web application are usually stored as files on the file system. Using a combination of directory naming conventions and a custom ASP.NET control, `Spring.Web` allows you to localize images within the page as easily as you do text resources.

The `Spring.Web.Page` class exposes the `ImagesRoot` property, with which you define the root directory where images are stored. The default value is `Images`, which means that the localizer expects to find an `Images` directory within the application root. But you can set the property to any value in the definition of the page.

To localize images, you create a directory for each localized culture under the `ImagesRoot` directory:

```
/MyApp
  /Images
    /en
    /en-US
    /fr
    /fr-CA
    /sr-SP-Cyrl
```

```
/sr-SP-Latn
...
```

Once an appropriate folder hierarchy is in place, you put the localized images in the appropriate directories and make sure that different translations of the same image have the same image name within the folders. To place a localized image on a page, you use the `<spring:LocalizedImage>`:

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"
    AutoEventWireup="false" Inherits="SpringAir.Web.StandardTemplate" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
<body>
    <spring:LocalizedImage id="logoImage" imageName="spring-air-logo.jpg" borderWidth="0" runat="server" />
</body>
</html>
```

This control will find the most specific directory that contains an image with the specified name using standard localization fallback rules and the user's culture. For example, if the user's culture is 'en-US', the localizer will look for the `spring-air-logo.jpg` file in `Images/en-US`, then in `Images/en` and finally, if the image file has still not been found, in the root `Images` directory (which for all practical purposes serves as an invariant culture folder).

22.8.6. User culture management

In addition to global and local resource management, Spring.Web supports user culture management by exposing the current `CultureInfo` through the `UserCulture` property on the `Page` and `UserControl` classes.

The `UserCulture` property delegates culture resolution to an implementation of the `Spring.Globalization.ICultureResolver` interface. You specify the culture resolver to use by configuring the `CultureResolver` property of the `Page` class in the relevant object definition:

```
<object name="BasePage" abstract="true">
    <property name="CultureResolver">
        <object type="Spring.Globalization.Resolvers.CookieCultureResolver, Spring.Web"/>
    </property>
</object>
```

Several useful implementations of `ICultureResolver` ship as part of Spring.Web, so it is unlikely that application developers need to implement their own culture resolver. However, you do need to implement your own culture resolver, the resulting implementation should be fairly straightforward as you need to implement only two methods. The following sections discuss each available implementation of the `ICultureResolver` interface.

22.8.6.1. DefaultWebCultureResolver

`DefaultWebCultureResolver`, the default culture resolver implementation, is used if you do not specify a culture resolver for a page, or if you inject a `DefaultWebCultureResolver` into a page definition explicitly. The latter case (explicit injection) is sometimes useful because you can specify a culture that should always be used, by defining the `DefaultCulture` property on the resolver.

The `DefaultWebCultureResolver` looks first at the `DefaultCulture` property and return its value if said property value is not null. If it is null, the `DefaultWebCultureResolver` falls back to request header inspection. If no 'Accept-Language' request headers are present, the resolver returns the UI culture of the currently executing thread.

22.8.6.2. RequestCultureResolver

The `RequestCultureResolver` resolver operates similar to the `DefaultWebCultureResolver`, except that it always checks request headers *first*, and only then falls back to the value of the `DefaultCulture` property or the culture code of the current thread.

22.8.6.3. SessionCultureResolver

The `SessionCultureResolver` resolver looks for culture information in the user's session and returns the information if it finds it. If not, `SessionCultureResolver` falls back to the behavior of the `DefaultWebCultureResolver`.

22.8.6.4. CookieCultureResolver

This resolver looks for culture information in a cookie, and return it if it finds one. If not, it falls back to the behavior of the `DefaultWebCultureResolver`.



Warning

`CookieCultureResolver` does not work if your application uses `localhost` as the server URL, which is a typical setting in a development environment.

To work around this limitation, use `SessionCultureResolver` during development and switch to `CookieCultureResolver` before you deploy the application in a production. This is easily accomplished in `Spring.Web` (simply change the config file) but is something that you should be aware of.

22.8.7. Changing cultures

To change the culture, application developers need to define one of the culture resolvers that support culture changes, such as `SessionCultureResolver` or `CookieCultureResolver` in the Spring application context. For example,

You also can write a custom `ICultureResolver` that persists culture information in a database, as part of a user's profile.

```
<object id="cultureResolver" type="Spring.Globalization.Resolvers.SessionCultureResolver, Spring.Web" />
```

Once that requirement is satisfied, you set the `UserCulture` property to a new `CultureInfo` object before the page is rendered. In the following `.aspx` example, two link buttons can be used to change the user's culture. In the code-behind, this is all one need do to set the new culture. A code snippet for the code-behind file (`UserRegistration.aspx.cs`) is shown below.

```
protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.english.Command += new CommandEventHandler(this.SetLanguage);
    this.serbian.Command += new CommandEventHandler(this.SetLanguage);

    base.OnInit(e);
}

private void SetLanguage(object sender, CommandEventArgs e)
{
    this.UserCulture = new CultureInfo((string) e.CommandArgument);
}
```

22.9. Result mapping

In many ASP.NET applications, no built-in way exists to externalize the flow of the application. The most common way of defining application flow is by hardcoding calls to the `Response.Redirect` and `Server.Transfer` methods within event handlers.

This approach is problematic because any changes to the flow of an application necessitates code changes (with the attendant recompilation, testing, redeployment, and so on). A better way, which works in many MVC ([Model-View-Controller](#)) web frameworks, is to enable you to externalize the mapping of action results to target pages.

Spring.Web adds this functionality to ASP.NET by allowing you to define result mappings within the definition of a page, and to then simply use logical result names within event handlers to control application flow.

In Spring.Web, a logical result is encapsulated and defined by the `Result` class; thus you can configure results like any other object:

```
<objects xmlns="http://www.springframework.net">

  <object id="homePageResult" type="Spring.Web.Support.Result, Spring.Web">
    <property name="TargetPage" value="~/Default.aspx"/>
    <property name="Mode" value="Transfer"/>
    <property name="Parameters">
      <dictionary>
        <entry key="literal" value="My Text"/>
        <entry key="name" value="%{UserInfo.FullName}"/>
        <entry key="host" value="%{Request.UserHostName}"/>
      </dictionary>
    </property>
  </object>

  <object id="loginPageResult" type="Spring.Web.Support.Result, Spring.Web">
    <property name="TargetPage" value="Login.aspx"/>
    <property name="Mode" value="Redirect"/>
  </object>

  <object type="UserRegistration.aspx" parent="basePage">
    <property name="UserManager" ref="userManager"/>
    <property name="Results">
      <dictionary>
        <entry key="userSaved" value-ref="homePageResult"/>
        <entry key="cancel" value-ref="loginPageResult"/>
      </dictionary>
    </property>
  </object>

</objects>
```

The only property for which you *must* supply a value for each result is the `TargetPage` property. The value of the `Mode` property can be `Transfer`, `TransferNoPreserve`, or `Redirect`, and defaults to `Transfer` if none is specified. `TransferNoPreserve` issues a server-side transfer with 'preserveForm=false', so that `QueryString` and `Form` data are not preserved.

If your target page requires parameters, you can define them with the `Parameters` dictionary property. You specify literal values or object navigation expressions for such parameter values. An expression is evaluated in the context of the page in which the result is being referenced. In the preceding example, any page that uses the `homePageResult` needs to expose a `UserInfo` property on the page class itself.



Note

In Spring.NET 1.1.0 and earlier, the prefix indicated an object navigation expression in the `Parameters` dictionary property was the dollar sign, for example, `${UserInfo.FullName}`. This convention conflicted with the prefix used to perform property replacement, the dollar sign, as described in the section `PropertyPlaceholderConfigurer`. As a workaround you can differentiate the prefix and suffix used in `PropertyPlaceholderConfigurer`, for example prefix = `$${` and suffix = `}`. In Spring.NET 1.1.1, a new prefix character, the percent sign (i.e. `%{UserInfo.FullName}`.) can be used in the `Parameters` dictionary to avoid this conflict, so you can keep the familiar NAnt style `PropertyPlaceholderConfigurer` defaults.

Parameters are handled differently depending on the result mode. For redirect results, every parameter is converted to a string, then URL encoded, and finally appended to a redirect query string. Parameters for transfer results are added to the `HttpContext.Items` collection before the request is transferred to the target page. Transfers are more flexible because any object can be passed as a parameter between pages. They are also more efficient because they don't require a round-trip to the client and back to the server, so transfer mode is recommended as the preferred result mode (it is also the current default).



Tip

If you need to customize how a redirect request is generated, for example, to encrypt the request parameters, subclass the `Request` object and override one or more protected methods, for example `string BuildUrl(string resolvedPath, IDictionary resolvedParameters)`. See the API documentation for additional information.

The preceding example shows independent result object definitions, which are useful for global results such as a home- and login- page. Result definitions are only used by one page should be simply embedded within the definition of a page, either as inner object definitions or using a special shortcut notation for defining a result definition:

```
<object type="~/UI/Forms/UserRegistration.aspx" parent="basePage">
  <property name="UserManager">
    <ref object="userManager"/>
  </property>

  <property name="Results">
    <dictionary>
      <entry key="userSaved" value="redirect:UserRegistered.aspx?status=Registration Successful,user=${UserInfo}"/>
      <entry key="cancel" value-ref="homePageResult"/>
    </dictionary>
  </property>
</object>
```

The short notation for the result must adhere to the following format...

```
[<mode>:]<targetPage>[?param1,param2,...,paramN]
```

Possible values for the `mode` value referred to in the preceding notation snippet:

- `redirect`: calls `Response.Redirect(string)`
- `redirectNoAbort`: calls `Response.Redirect(string, false)`
- `transfer`: calls `Server.Transfer(string)`
- `TransferNoPreserve`: calls `Server.Transfer(string, false)`

These values correspond to the values of the `ResultMode` enumeration. A comma separates parameters instead of an ampersand; this avoids laborious ampersand escaping within an XML object definition. The use of the ampersand character is still supported if required, but you then have to specify the ampersand character using the well known `&` entity reference.

After you define your results, you can use them within the event handlers of your pages (`UserRegistration.aspx.cs`):

```
private void SaveUser(object sender, EventArgs e)
{
```

```

    UserManager.SaveUser(UserInfo);
    SetResult("userSaved");
}

public void Cancel(object sender, EventArgs e)
{
    SetResult("cancel");
}

protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.saveButton.Click += new EventHandler(this.SaveUser);
    this.cancelButton.Click += new EventHandler(this.Cancel);

    base.OnInit(e);
}

```

You can further refactor the preceding example and use defined constants, which is advisable when a logical result name such as "home" is likely to be referenced by many pages.

22.9.1. Registering user defined transfer modes

You can also register a custom interpreter that can parse the shorthand string representation that creates a Result object. To do this you should view the result mapping string representation as consisting of two parts:

```
<resultmode>:<textual result representation>
```

The interface `IResultFactory` is responsible for creating an `IResult` object from these two pieces:

```

public interface IResultFactory
{
    IResult CreateResult( string resultMode, string resultText );
}

```

You use a `ResultFactoryRegistry` to associate a given `resultmode` string with an `IResultFactory` implementation:

```

class MySpecialResultLogic : IResult
{
    ...
}

class MySpecialResultLogicFactory : IResultFactory
{
    IResult Create( string mode, string expression ) {
        /* ... convert 'expression' into MySpecialResultLogic */
    }
}

// register with global factory
ResultFactoryRegistry.RegisterResultFactory( "mySpecialMode", new MySpecialResultLogicFactory );

```

You then use the custom continue mode in your page:

```

<-- configure your Results -->
<object type="mypage.aspx">
  <property name="Results">
    <dictionary>
      <entry key="continue" value="mySpecialMode:<some MySpecialResultLogic string representation>" />
    </dictionary>
  </property>
</object>

```

The result redirection is done as before, by calling `myPage.SetResult("cancel");`

22.10. Client-side scripting

ASP.NET supports client-side scripting through the use of the `Page.RegisterClientScriptBlock` and `Page.RegisterStartupScript` methods. However, neither method allows you to output a registered script markup within a `<head>` section of a page, which in many cases is exactly what you need to do.

22.10.1. Registering scripts within the head HTML section

Spring.Web adds several methods to enhance client-side scripting to the base `Spring.Web.UI.Page` class: `RegisterHeadScriptBlock` and `RegisterHeadScriptFile`, each with a few overrides. You can call these methods from your custom pages and controls in order to register script blocks and script files that must be included in the `<head>` section of the final HTML page.

You must use the `<spring:Head>` server-side control to define your `<head>` section instead of using the standard HTML `<head>` element. This is shown below.

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"
    AutoEventWireup="false" Inherits="SpringAir.Web.StandardTemplate" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <spring:Head runat="server" id="Head1">
    <title>
      <spring:ContentPlaceHolder id="title" runat="server">
        <%= GetMessage("default.title") %>
      </spring:ContentPlaceHolder>
    </title>
    <LINK href="<%= CssRoot %>/default.css" type="text/css" rel="stylesheet">
    <spring:ContentPlaceHolder id="head" runat="server"></spring:ContentPlaceHolder>
  </spring:Head>
  <body>
    ...
  </body>
</html>
```

The preceding example above shows how you typically set-up a `<head>` section within a master page template to be able to change the title value and to add additional elements to the `<head>` section from the child pages using `<spring:ContentPlaceholder>` controls. However, only the `<spring:Head>` declaration is required in order for Spring.NET `Register*` scripts to work properly.

22.10.2. Adding CSS definitions to the head section

In a similar fashion, you can add references to CSS files, or even specific styles, directly to the `<head>` HTML section using `Page.RegisterStyle` and `Page.RegisterStyleFile` methods. The latter one simply allows you to include a reference to an external CSS file, while the former one allows you to define embedded style definitions by specifying the style name and definition as the parameters. The final list of style definitions registered this way will be rendered within the single embedded `style` section of the final HTML document.

22.10.3. Well-known directories

To make the manual inclusion of client-side scripts, CSS files and images easier, the `Spring.Web` `Page` class exposes several properties that help you reference such artifacts with absolute paths. This capability gives web application developers convenience functionality straight out of the box if they stick to common conventions such as a web application (directory) structure.

These properties are `ScriptsRoot`, `CssRoot` and `ImagesRoot`. They have default values of `Scripts`, `CSS` and `Images`, which work well if you create and use these directories in your web application root. However, if you

prefer to place them somewhere else, you can always override default values by injecting new values into your page definitions (you will typically inject these values only in the base page definition, as they are normally shared by all the pages in the application). An example of such configuration is shown below:

```
<object name="basePage" abstract="true">
  <description>
    Convenience base page definition for all the pages.

    Pages that reference this definition as their parent (see the examples below)
    will automatically inherit the following properties....

  </description>
  <property name="CssRoot" value="Web/CSS"/>
  <property name="ImagesRoot" value="Web/Images"/>
</object>
```

22.11. Spring user controls

Spring provides several custom user controls that are located in the `Spring.Web.UI.Controls` namespace. This section lists the controls and points to other documentation to provide additional information. Check the SDK docs for descriptions of controls that are not mentioned here.

22.11.1. Validation controls

You can specify the location in the web page where validation errors are to be rendered by using the `ValidationSummary` and `ValidationError` controls. Two controls exist because they have different defaults for how errors are rendered. `ValidationSummary` is used to display potentially multiple errors identified by the validation framework. `ValidationError` is used to display field-level validation errors. Please refer to the section ASP.NET usage tips in the chapter on the Validation Framework more information.

22.11.2. Databinding controls

Some standard controls are not easy to use with Spring's databinding support. Examples are check boxes and radio button groups. Here you should use the `CheckBoxList` and `RadioButtonGroup` controls. You can do databinding itself by using the `DataBindingPanel` instead of the using the `BindingManager` API within the code behind page.

22.11.3. Calendar control

A pop-up DHTML calendar control is provided. It is a slightly modified version of the [Dynarch.com DHTML Calendar](#) control written by Mihai Bazon.

22.11.4. Panel control

You can suppress dependency injection for controls inside your ASP.NET by using the `Panel` control. See Customizing control dependency injection .

Chapter 23. ASP.NET AJAX

23.1. Introduction

Spring's ASP.NET AJAX integration allows for a plain .NET object (PONO), that is one that doesn't have any attributes or special base classes, to be exported as a web service, configured via dependency injection, 'decorated' by applying AOP, and then exposed to client side JavaScript.

23.2. Web Services

Spring.NET, and particularly Spring.Web, improved [support for web services](#) in .NET with the `WebServiceExporter`. Exporting of an ordinary plain .NET object as a web service is achieved by registering a custom implementation of the `WebServiceHandlerFactory` class as the HTTP handler for *.asmx requests.

[Microsoft ASP.NET AJAX](#) introduced a new HTTP handler `System.Web.Script.Services.ScriptHandlerFactory` to allow a Web Service to be invoked from the browser by using JavaScript.

Spring's integration allows for both Spring.Web and ASP.NET AJAX functionality to be used together by creating a new HTTP handler.

23.2.1. Exposing Web Services

The `WebServiceExporter` combined with the new HTTP handler exposes PONOs as Web Services in your ASP.NET AJAX application.

In order for a Web service to be accessed from script, the `WebServiceExporter` should decorate the Web Service class with the `ScriptServiceAttribute`. The code below is taken from the sample application `Spring.Web.Extensions.Sample`, aka the 'AJAX' shortcut in the installation. :

```
<object id="ContactWebService" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="ContactService"/>
  <property name="Namespace" value="http://Spring.Examples.Atlas/ContactService"/>
  <property name="Description" value="Contact Web Services"/>
  <property name="TypeAttributes">
    <list>
      <object type="System.Web.Script.Services.ScriptServiceAttribute, System.Web.Extensions"/>
    </list>
  </property>
</object>
```

All that one needs to do in order to use the `WebServiceExporter` is:

1. *Configure the Web.config file of your ASP.NET AJAX application as a Spring.Web application.*

```
<sectionGroup name="spring">
  <section name="context" type="Spring.Context.Support.WebContextHandler, Spring.Web"/>
</sectionGroup>
```

```
<spring>
```

```
<context>
  <resource uri="~/Spring.config"/>
</context>
</spring>
```

2. Register the HTTP handler and the Spring HttpModule under the `system.web` section.

```
<httpHandlers>
  <remove verb="*" path="*.asmx"/>
  <add verb="*" path="*.asmx" validate="false" type="Spring.Web.Script.Services.ScriptHandlerFactory, Spring.Web.Extensions" />
  <add verb="*" path="*_AppService.axd" validate="false" type="System.Web.Script.Services.ScriptHandlerFactory, System.Web.Extensions" />
  <add verb="GET,HEAD" path="ScriptResource.axd" type="System.Web.Handlers.ScriptResourceHandler, System.Web.Extensions, Version=1.0.61025.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
</httpHandlers>

<httpModules>
  <add name="ScriptModule" type="System.Web.Handlers.ScriptModule, System.Web.Extensions, Version=1.0.61025.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
  <add name="SpringModule" type="Spring.Context.Support.WebSupportModule, Spring.Web" />
</httpModules>
```

3. Register the HTTP handler and the Spring HttpModule under `system.webServer` section.

```
<modules>
  <add name="ScriptModule" preCondition="integratedMode" type="System.Web.Handlers.ScriptModule, System.Web.Extensions, Version=1.0.61025.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
  <add name="SpringModule" type="Spring.Context.Support.WebSupportModule, Spring.Web" />
</modules>
<handlers>
  <remove name="WebServiceHandlerFactory-Integrated" />
  <add name="ScriptHandlerFactory" verb="*" path="*.asmx" preCondition="integratedMode" type="Spring.Web.Script.Services.ScriptHandlerFactory, Spring.Web.Extensions" />
  <add name="ScriptHandlerFactoryAppServices" verb="*" path="*_AppService.axd" preCondition="integratedMode" type="System.Web.Script.Services.ScriptHandlerFactory, System.Web.Extensions, Version=1.0.61025.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
  <add name="ScriptResource" preCondition="integratedMode" verb="GET,HEAD" path="ScriptResource.axd" type="System.Web.Handlers.ScriptResourceHandler, System.Web.Extensions, Version=1.0.61025.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
</handlers>
```

You can find a full Web.config file in the example that comes with this integration.

23.2.2. Calling Web Services by using JavaScript

A proxy class is generated for each Web Service. Calls to Web Services methods are made by using this proxy class. When using the `WebServiceExporter`, the name of the proxy class is equal to the `WebServiceExporter`'s id.

```
// This function calls the Contact Web service method
// passing simple type parameters and the callback function
function GetEmails(prefix, count)
{
  ContactWebService.GetEmails(prefix, count, GetEmailsOnSucceeded);
}
```

Part IV. Services

This part of the reference documentation covers the Spring Framework's integration with .NET distributed technologies such as .NET Remoting, Enterprise Services, Web Services. Integration with WCF Services is forthcoming. Please refer to the introduction chapter for more details.

- Chapter 24, *Introduction to Spring Services*
- Chapter 25, *.NET Remoting*
- Chapter 26, *.NET Enterprise Services*
- Chapter 27, *Web Services*
- Chapter 28, *Windows Communication Foundation (WCF)*

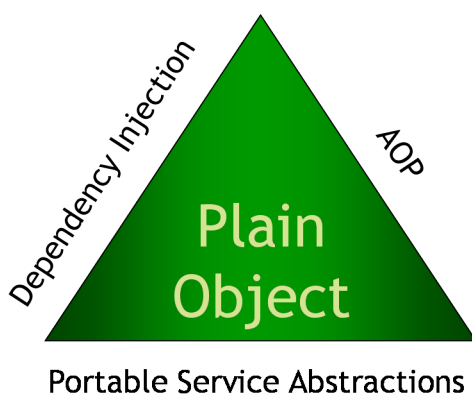
Chapter 24. Introduction to Spring Services

24.1. Introduction

The goal of Spring's integration with distributed technologies is to adapt plain .NET objects so they can be used with a specific distributed technology. This integration is designed to be as non-intrusive as possible. If you need to expose an object to a remote process then you can define an exporter for that object. Similarly, on the client side you define an corresponding endpoint accessor. Of course, the object's methods still need to be suitable for remoting, i.e. coarse grained, to avoid making unnecessary and expensive remote calls.

Since these exporters and client side endpoint accessors are defined using meta data for Spring IoC container, you can easily use dependency injection on them to set initial state and to 'wire up' the presentation tier, such as web forms, to the service layer. In addition, you may apply AOP aspects to the exported classes and/or service endpoints to apply behavior such as logging, security, or other custom behavior that may not be provided by the target distributed technology. The Spring specific terminology for this approach to object distribution is known as Portable Service Abstractions (PSA). As a result of this approach, you can decide much later in the development process the technical details of how you will distribute your objects as compared to traditional code centric approaches. Changing of the implementation is done through configuration of the IoC container and not by recompilation. Of course, you may choose to not use the IoC container to manage these objects and use the exporter and service endpoints programmatically.

The diagram shown below is a useful way to demonstrate the key abstractions in the Spring tool chest and their interrelationships. The four key concepts are; plain .NET objects, Dependency Injection, AOP, and Portable Service Abstractions. At the heart sits the plain .NET object that can be instantiated and configured using dependency injection. Then, optionally, the plain object can be adapted to a specific distributed technology. Lastly, additional behavior can be applied to objects. This behavior is typically that which can not be easily address by traditional OO approaches such as inheritance. In the case of service layer, common requirements such as 'the service layer must be transactional' are implemented in a manner that naturally expresses that intention in a single place, as compared to scattered code across the service layer.



Spring implements this exporter functionality by creating a proxy at runtime that meets the implementation requirements of a specific distributed technology. In the case of .NET Remoting the proxy will inherit from MarshalByRef, for EnterpriseServices it will inherit from ServiceComponent and for aspx web services, WebMethod attributes will be added to methods. Client side functionality is often implemented by a thin layer over the client access mechanism of the underlying distributed technology, though in some cases such as client

side access to web services, you have the option to create a proxy on the fly from the .wsdl definition, much like you would have done using the command line tools.

The common implementation theme for you as a provider of these service objects is to implement an interface. This is generally considered a best practice in its own right, you will see most pure WCF examples following this practice, and also lends itself to a straightforward approach to unit testing business functionality as stub or mock implementations may be defined for testing purposes.

The assembly `Spring.Services.dll` contains support for .NET Remoting, Enterprise Services and ASMX Web Services. Support for WCF services is planned for Spring 1.2 and is currently in the CVS repository if you care to take an early look.

Chapter 25. .NET Remoting

25.1. Introduction

Spring's .NET Remoting support allows you to export a 'plain .NET object' as a .NET Remoted object. By "plain .NET object" we mean classes that do not inherit from a specific infrastructure base class such as `MarshalByRefObject`. On the server side, Spring's .NET Remoting exporters will automatically create a proxy that implements `MarshalByRefObject`. You register SAO types as either `SingleCall` or `Singleton` and also configure on a per-object basis lifetime and leasing parameters. On the client side you can obtain CAO references to server proxy objects in a manner that promotes interface based design best practices when developing .NET remoting applications. The current implementation requires that your plain .NET objects implements a business service interface. Additionally you can add AOP advice to both SAO and CAO objects.

You can leverage the IoC container to configure the exporter and service endpoints. A remoting specific xml-schema is provided to simplify the remoting configuration but you can still use the standard reflection-like property based configuration schema. You may also opt to not use the IoC container to configure the objects and use Spring's .NET Remoting classes Programmatically, as you would with any third party library.

A sample application, often referred to in this documentation, is in the distribution under the directory "examples \Spring\Spring.Calculator" and may also be found via the start menu by selecting the 'Calculator' item.

25.2. Publishing SAOs on the Server

Exposing a Singleton SAO service can be done in two ways. The first is through programmatic or administrative type registration that makes calls to `RemotingConfiguration.RegisterWellKnownServiceType`. This method has the limitation that you must use a default constructor and you can not easily configure the singleton state at runtime since it is created on demand. The second way is to publish an object instance using `RemotingServices.Marshal`. This method overcomes the limitations of the first method. Example server side code for publishing an SAO singleton object with a predefined state is shown below

```
AdvancedMBRCalculator calc = new AdvancedMBRCalculator(217);
RemotingServices.Marshal(calc, "MyRemotedCalculator");
```

The class `AdvancedMBRCalculator` used above inherits from `MarshalByRefObject`.

If your design calls for configuring a singleton SAO, or using a non-default constructor, you can use the Spring IoC container to create the SAO instance, configure it, and register it with the .NET remoting infrastructure. The `SaoExporter` class performs this task and most importantly, will automatically create a proxy class that inherits from `MarshalbyRefObject` if your business object does not already do so. The following XML taken from the Remoting QuickStart demonstrates its usage to an SAO Singleton object

25.2.1. SAO Singleton

```
<object id="singletonCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services">
  <constructor-arg type="int" value="217"/>
</object>

<!-- Registers the calculator service as a SAO in 'Singleton' mode. -->
<object name="saoSingletonCalculator" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculator" />
  <property name="ServiceName" value="RemotedSaoSingletonCalculator" />
</object>
```

This XML fragment shows how an existing object "singletonCalculator" defined in the Spring context is exposed under the url-path name "RemotedSaoSingletonCalculator". (The fully qualified url is tcp://localhost:8005/RemotedSaoSingleCallCalculator using the standard .NET channel configuration shown further below.) AdvancedCalculator class implements the business interface IAdvancedCalculator. The current proxy implementation requires that your business objects implement an interface. The interfaces' methods will be the ones exposed in the generated .NET remoting proxy. The initial memory of the calculator is set to 217 via the constructor. The class AdvancedCalculator *does not* inherit from MarshalByRefObject. Also note that the exporter sets the lifetime of the SAO Singleton to infinite so that the singleton will not be garbage collected after 5 minutes (the .NET default lease time). If you would like to vary the lifetime properties, they are InitialLeaseTime, RenewOnCallTime, and SponsorshipTimeout.

A custom schema is provided to make the object declaration even easier and with intellisense support for the attributes. This is shown below

```
<objects xmlns="http://www.springframework.net"
  xmlns:r="http://www.springframework.net/remoting">

  <r:saoExporter targetName="singletonCalculator"
    serviceName="RemotedSaoSingletonCalculator" />

  ... other object definitions

</objects>
```

Refer to the end of this chapter for more information on Spring's .NET custom schema.

25.2.2. SAO SingleCall

The following XML fragment shows how to expose the calculator service in SAO 'SingleCall' mode.

```
<object id="prototypeCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services"
  singleton="false">
  <constructor-arg type="int" value="217"/>
</object>

<object name="saoSingleCallCalculator" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="prototypeCalculator" />
  <property name="ServiceName" value="RemotedSaoSingleCallCalculator" />
</object>
```

Note that we change the singleton attribute of the plain .NET object as configured by Spring in the <object> definition and not an attribute on the SaoExporter. The object referred to in the TargetName parameter can be an AOP proxy to a business object. For example, if we were to apply some simple logging advice to the singleton calculator, the following standard AOP configuration is used to create the target for the SaoExporter

```
<object id="singletonCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="target" ref="singletonCalculator"/>
  <property name="interceptorNames">
    <list>
      <value>Log4NetLoggingAroundAdvice</value>
    </list>
  </property>
</object>

<object name="saoSingletonCalculatorWeaved" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculatorWeaved" />
  <property name="ServiceName" value="RemotedSaoSingletonCalculatorWeaved" />
</object>
```



Note

As generally required with a .NET Remoting application, the arguments to your service methods should be Serializable.

25.2.1. Console Application Configuration

When using `SaoExporter` you can still use the standard remoting administration section in the application configuration file to register the channel. `ChannelServices` as shown below

```
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp" port="8005" />
    </channels>
  </application>
</system.runtime.remoting>
```

A console application that will host this Remoted object needs to initialize the .NET Remoting infrastructure with a call to `RemotingConfiguration` (since we are using the `.config` file for channel registration) and then start the Spring application context. This is shown below

```
RemotingConfiguration.Configure("RemoteApp.exe.config");

IApplicationContext ctx = ContextRegistry.GetContext();

Console.Out.WriteLine("Server listening...");

Console.ReadLine();
```

You can also put in the configuration file an instance of the object `Spring.Remoting.RemotingConfigurer` to make the `RemotingConfiguration` call show above on your behalf during initialization of the IoC container. The `RemotingConfigurer` implements the `IObjectFactoryPostProcessor` interface, which gets called after all object definitions have been loaded but before they have been instantiated, (SeeSection 5.9.2, “Customizing configuration metadata with ObjectFactoryPostProcessors” for more information). The `RemotingConfigurer` has two properties you can configure. `Filename`, that specifies the filename to load the .NET remoting configuration from (if null the default file name is used) and `EnsureSecurity` which makes sure the channel is encrypted (available only on .NET 2.0). As a convenience, the custom Spring remoting schema can be used to define an instance of this class as shown below, taken from the `Remoting QuickStart`

```
<objects xmlns="http://www.springframework.net"
  xmlns:r="http://www.springframework.net/remoting">

  <r:configurer filename="Spring.Calculator.RemoteApp.exe.config" />

</objects>
```

The `ReadLine` prevents the console application from exiting. You can refer to the code in `RemoteApp` in the `Remoting QuickStart` to see this code in action.

25.2.3. IIS Application Configuration

If you are deploying a .NET remoting application inside IIS there is a [sample project](#) that demonstrates the necessary configuration using `Spring.Web`.

`Spring.Web` ensures the application context is initialized, but if you don't use `Spring.Web` the idea is to start the initialization of the Spring IoC container inside the application start method defined in `Global.asax`, as shown below

```
void Application_Start(object sender, EventArgs e)
{
    // Code that runs on application startup

    // Ensure Spring has loaded configuration registering context
```

```

Spring.Context.IApplicationContext ctx = new Spring.Context.Support.XmlApplicationContext(
    HttpContext.Current.Server.MapPath("Spring.Config"));
Spring.Context.Support.ContextRegistry.RegisterContext(ctx);
}

```

In this example, the Spring configuration file is named `Spring.Config`. Inside `Web.config` you add a standard `<system.runtime.remoting>` section. Note that you do not need to specify the port number of your channels as they will use the port number of your web site. Ambiguous results have been reported if you do specify the port number. Also, in order for IIS to recognize the remoting request, you should add the suffix `'.rem'` or `'.soap'` to the target name of your exported remote object so that the correct IIS handler can be invoked.

25.3. Accessing a SAO on the Client

Administrative type registration on the client side lets you easily obtain a reference to a SAO object. When a type is registered on the client, using the new operator or using the reflection API will return a proxy to the remote object instead of a local reference. Administrative type registration on the client for a SAO object is performed using the `wellknown` element in the client configuration section. However, this approach requires that you expose the implementation of the class on the client side. Practically speaking this would mean linking in the server assembly to the client application, a generally recognized bad practice. This dependency can be removed by developing remote services based on a business interface. Aside from remoting considerations, the separation of interface and implementation is considered a good practice when designing OO systems. In the context of remoting, this means that the client can obtain a proxy to a specific implementation with *only* a reference to the interface assembly. To achieve the decoupling of client and server, a separate assembly containing the interface definitions is created and shared between the client and server applications.

There is a simple means for following this design when the remote object is a SAO object. A call to `Activator.GetObject` will instantiate a SAO proxy on the client. For CAO objects another mechanism is used and is discussed later. The code to obtain the SAO proxy is shown below

```

ICalculator calc = (ICalculator)Activator.GetObject (
    typeof (ICalculator),
    "tcp://localhost:8005/MyRemotedCalculator");

```

To obtain a reference to a SAO proxy within the IoC container, you can use the object factory `SaoFactoryObject` in the Spring configuration file. The following XML taken from the `Remoting QuickStart` demonstrates its usage.

```

<object id="calculatorService" type="Spring.Remoting.SaoFactoryObject, Spring.Services">
  <property name="ServiceInterface" value="Spring.Calculator.Interfaces.IAdvancedCalculator, Spring.Calculator.Contract" />
  <property name="ServiceUrl" value="tcp://localhost:8005/RemotedSaoSingletonCalculator" />
</object>

```

The `ServiceInterface` property specifies the type of proxy to create while the `ServiceUrl` property creates a proxy bound to the specified server and published object name.

Other objects in the IoC container that depend on an implementation of the interface `ICalculator` can now refer to the object `"calculatorService"`, thereby using a remote implementation of this interface. The exposure of dependencies among objects within the IoC container lets you easily switch the implementation of `ICalculator`. By using the IoC container changing the application to use a local instead of remote implementation is a configuration file change, not a code change. By promoting interface based programming, the ability to switch implementation makes it easier to unit test the client application, since unit testing can be done with a mock implementation of the interface. Similarly, development of the client can proceed independent of the server implementation. This increases productivity when there are separate client and server development teams. The two teams agree on interfaces before starting development. The client team can quickly create a simple, but functional implementation and then integrate with the server implementation when it is ready.

25.4. CAO best practices

Creating a client activated object (CAO) is typically done by administrative type registration, either Programatically or via the standard .NET remoting configuration section. The registration process allows you to use the 'new' operator to create the remote object and requires that the implementation of the object be distributed to the client. As mentioned before, this is not a desirable approach to developing distributed systems. The best practice approach that avoids this problem is to create an SAO based factory class on the server that will return CAO references to the client. In a manner similar to how Spring's generic object factory can be used as a replacement creating a factory per class, we can create a generic SAO object factory to return CAO references to objects defined in Spring's application context. This functionality is encapsulated in Spring's `CaoExporter` class. On the client side a reference is obtained using `CaoFactoryObject`. The client side factory object supports creation of the CAO object using constructor arguments. In addition to reducing the clutter and tedium around creating factory classes specific to each object type you wish to expose in this manner, this approach has the additional benefit of not requiring any type registration on the client or server side. This is because the act of returning an instance of a class that inherits from `MarshalByRefObject` across a remoting boundary automatically returns a CAO object reference. For more information on this best-practice, refer to the last section, Section 25.8, "Additional Resources", for some links to additional resources.

25.5. Registering a CAO object on the Server

To expose an object as a CAO on the server you should declare an object in the standard Spring configuration that is a 'prototype', that is the singleton property is set to false. This results in a new object being created each time it is retrieved from Spring's IoC container. An implementation of `ICaoRemoteFactory` is what is exported via a call to `RemotingServices.Marshal`. This implementation uses Spring's IoC container to create objects and then dynamically create a .NET remoting proxy for the retrieved object. Note that the default lifetime of the remote object is set to infinite (null is returned from the implementation of `InitializeLifetimeService()`).

This is best shown using an example from the Remoting Quickstart application. Here is the definition of a simple calculator object,

```
<object id="prototypeCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services"
    singleton="false">
    <constructor-arg type="int" value="217" />
</object>
```

To export this as a CAO object we can declare the `CaoExporter` object directly in the server's XML configuration file, as shown below

```
<object id="caoCalculator" type="Spring.Remoting.CaoExporter, Spring.Services">
    <property name="TargetName" value="prototypeCalculator" />
    <property name="Infinite" value="false" />
    <property name="InitialLeaseTime" value="2m" />
    <property name="RenewOnCallTime" value="1m" />
</object>
```

Note the property 'TargetName' is set to the name, not the reference, of the non-singleton declaration of the 'AdvancedCalculator' class.

Alternatively, you can use the remoting schema and declare the CAO object as shown below

```
<r:caoExporter targetName="prototypeCalculator" infinite="false">
    <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m" />
</r:caoExporter>
```


25.5.1. Applying AOP advice to exported CAO objects

Applying AOP advice to exported CAO objects is done by referencing the advised object name to the CAO exporter. Again, taking an example from the Remoting QuickStart, a calculator with logging around advice is defined as shown below.

```
<object id="prototypeCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="targetSource">
    <object type="Spring.Aop.Target.PrototypeTargetSource, Spring.Aop">
      <property name="TargetObjectName" value="prototypeCalculator" />
    </object>
  </property>
  <property name="interceptorNames">
    <list>
      <value>ConsoleLoggingAroundAdvice</value>
    </list>
  </property>
</object>
```

If this declaration is unfamiliar to you, please refer to Chapter 13, *Aspect Oriented Programming with Spring.NET* for more information. The CAO exporter then references with the name 'prototypeCalculatorWeaved' as shown below.

```
<r:caoExporter targetName="prototypeCalculatorWeaved" infinite="false">
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m" />
</r:caoExporter>
```

25.6. Accessing a CAO on the Client

On the client side a CAO reference is obtained by using the `CaoFactoryObject` as shown below

```
<object id="calculatorService" type="Spring.Remoting.CaoFactoryObject, Spring.Services">
  <property name="RemoteTargetName" value="prototypeCalculator" />
  <property name="ServiceUrl" value="tcp://localhost:8005" />
</object>
```

This definition corresponds to the exported calculator from the previous section. The property 'RemoteTargetName' identifies the object on the server side. Using this approach the client can obtain an reference though standard DI techniques to a remote object that implements the `IAdvancedCalculator` interface. (As always, that doesn't mean the client should treat the object as if it was an in-process object).

Alternatively, you can use the Remoting schema to shorten this definition and provide intellisense code completion

```
<r:caoFactory id="calculatorService"
  remoteTargetName="prototypeCalculator"
  serviceUrl="tcp://localhost:8005" />
```

25.6.1. Applying AOP advice to client side CAO objects.

Applying AOP advice to a client side CAO object is done just like any other object. Simply use the id of the object created by the `CaoFactoryObject` as the AOP target, i.e. 'calculatorService' in the previous example.

25.7. XML Schema for configuration

Please install the XSD schemas into VS.NET as described in Chapter 34, *Visual Studio.NET Integration*. XML intellisense for the attributes of the `saoExporter`, `caoExporter` and `caoFactory` should be self explanatory as they mimic the standard property names used to configure .NET remote objects.

25.8. Additional Resources

Two articles that describe the process of creating a standard SAO factory for returning CAO objects are [Implementing Broker with .NET Remoting Using Client-Activated Objects](#) on MSDN and [Step by Step guide to CAO creation through SAO class factories](#) on Glacial Components website.

Chapter 26. .NET Enterprise Services

26.1. Introduction

Spring's .NET Enterprise Services support allows you to export a 'plain .NET object' as a .NET Remoted object. By "plain .NET object" we mean classes that do not inherit from a specific infrastructure base class such as `ServiceComponent`.

You can leverage the IoC container to configure the exporter and service endpoints. You may also opt to not use the IoC container to configure the objects and use Spring's .NET Enterprise Services classes Programmatically, as you would with any third party library.

26.2. Serviced Components

Services components in .NET are able to use COM+ services such as declarative and distributed transactions, role based security, object pooling messaging. To access these services your class needs to derive from the class `System.EnterpriseServices.ServiceComponent`, adorn your class and assemblies with relevant attributes, and configure your application by registering your serviced components with the COM+ catalog. The overall landscape of accessing and using COM+ services within .NET goes by the name .NET Enterprise Services.

Many of these services can be provided without the need to derive from a `ServiceComponent` though the use of Spring's Aspect-Oriented Programming functionality. Nevertheless, you may be interested in exporting your class as a serviced component and having client access that component in a location transparent manner. By using Spring's `ServiceComponentExporter`, `EnterpriseServicesExporter` and `ServiceComponentFactory` you can easily create and consume serviced components without having your class inherit from `ServiceComponent` and automate the manual deployment process that involves strongly signing your assembly and using the `regsvcs` utility.

Note that the following sections do not delve into the details of programming .NET Enterprise Services. An excellent reference for such information is Christian Nagel's "Enterprise Services with the .NET Framework" Spring.NET includes an example of using these classes, the 'calculator' example. More information can be found in the section, .NET Enterprise Services example.

26.3. Server Side

One of the main challenges for the exporting of a serviced component to the host is the need for them to be contained within a physical assembly on the file system in order to be registered with the COM+ Services. To make things more complicated, this assembly has to be strongly named before it can be successfully registered.

Spring provides two classes that allow all of this to happen.

- `Spring.Enterprise.ServiceComponentExporter` is responsible for exporting a single component and making sure that it derives from `ServiceComponent` class. It also allows you to specify class-level and method-level attributes for the component in order to define things such as transactional behavior, queuing, etc.
- `Spring.Enterprise.EnterpriseServicesExporter` corresponds to a COM+ application, and it allows you to specify list of components that should be included in the application, as well as the application name and other assembly-level attributes

Let's say that we have a simple service interface and implementation class, such as these:

```
namespace MyApp.Services
{
    public interface IUserManager
    {
        User GetUser(int userId);
        void SaveUser(User user);
    }

    public class SimpleUserManager : IUserManager
    {
        private IUserDao userDao;
        public IUserDao UserDao
        {
            get { return userDao; }
            set { userDao = value; }
        }

        public User GetUser(int userId)
        {
            return UserDao.FindUser(userId);
        }

        public void SaveUser(User user)
        {
            if (user.IsValid)
            {
                UserDao.SaveUser(user);
            }
        }
    }
}
```

And the corresponding object definition for it in the application context config file:

```
<object id="userManager" type="MyApp.Services.SimpleUserManager">
  <property name="UserDao" ref="userDao"/>
</object>
```

Let's say that we want to expose user manager as a serviced component so we can leverage its support for transactions. First we need to export our service using the exporter `ServicedComponentExporter` as shown below

```
<object id="MyApp.EnterpriseServices.UserManager" type="Spring.Enterprise.ServicedComponentExporter, Spring.Services">
  <property name="TargetName" value="userManager"/>
  <property name="TypeAttributes">
    <list>
      <object type="System.EnterpriseServices.TransactionAttribute, System.EnterpriseServices"/>
    </list>
  </property>
  <property name="MemberAttributes">
    <dictionary>
      <entry key="*">
        <list>
          <object type="System.EnterpriseServices.AutoCompleteAttribute, System.EnterpriseServices"/>
        </list>
      </entry>
    </dictionary>
  </property>
</object>
```

The exporter defined above will create a composition proxy for our `SimpleUserManager` class that extends `ServicedComponent` and delegates method calls to `SimpleUserManager` instance. It will also adorn the proxy class with a `TransactionAttribute` and all methods with an `AutoCompleteAttribute`.

The next thing we need to do is configure an exporter for the COM+ application that will host our new component:

```
<object id="MyComponentExporter" type="Spring.Enterprise.EnterpriseServicesExporter, Spring.Services">
  <property name="ApplicationName" value="My COM+ Application"/>
  <property name="Description" value="My enterprise services application."/>
  <property name="AccessControl">
```

```
<object type="System.EnterpriseServices.ApplicationAccessControlAttribute, System.EnterpriseServices">
  <property name="AccessChecksLevel" value="ApplicationComponent"/>
</object>
</property>
<property name="Roles">
  <list>
    <value>Admin : Administrator role</value>
    <value>User : User role</value>
    <value>Manager : Administrator role</value>
  </list>
</property>
<property name="Components">
  <list>
    <ref object="MyApp.EnterpriseServices.UserManager"/>
  </list>
</property>
<property name="Assembly" value="MyComPlusApp"/>
</object>
```

This exporter will put all proxy classes for the specified list of components into the specified assembly, sign the assembly, and register it with the specified COM+ application name. If application does not exist it will create it and configure it using values specified for Description, AccessControl and Roles properties.

26.4. Client Side

Because serviced component classes are dynamically generated and registered, you cannot instantiate them in your code using the new operator. Instead, you need to use `Spring.Enterprise.ServicedComponentFactory` definition, which also allows you to specify the configuration template for the component as well as the name of the remote server the component is running on, if necessary. An example is shown below

```
<object id="enterpriseUserManager" type="Spring.Enterprise.ServicedComponentFactory, Spring.Services">
  <property name="Name" value="MyApp.EnterpriseServices.UserManager"/>
  <property name="Template" value="userManager"/>
</object>
```

You can then inject this instance of the `IUserManager` into a client class and use it just like you would use original `SimpleUserManager` implementation. As you can see, by coding your services as plain .Net objects, against well defined service interfaces, you gain easy pluggability for your service implementation though this configuration, while keeping the core business logic in a technology agnostic PONO, i.e. Plain Ordinary .Net Object.

Chapter 27. Web Services

27.1. Introduction

While the out-of-the-box support for web services in .NET is excellent, there are a few areas that the Spring.NET team thought could use some improvement. Spring adds the ability to perform dependency injection on standard asmx web services. Spring's .NET Web Services support also allows you to export a 'plain .NET object' as a .NET web service. By "plain .NET object" we mean classes that do not contain infrastructure specific attributes, such as `WebMethod`. On the server side, Spring's .NET web service exporters will automatically create a proxy that adds web service attributes. On the client side you can use Spring IoC container to configure a client side proxy that you generated with standard command line tools. Additionally, Spring provides the functionality to create the web service proxy dynamically at runtime (much like running the command line tools but at runtime and without some of the tools quirks) and use dependency injection to configure the resulting proxy class. On both the server and client side, you can apply AOP advice to add behavior such as logging, exception handling, etc. that is not easily encapsulated within an inheritance hierarchy across the application.

27.2. Server-side

One thing that the Spring.NET team didn't like much is that we had to have all these .asmx files lying around when all said files did was specify which class to instantiate to handle web service requests.

Second, the Spring.NET team also wanted to be able to use the Spring.NET IoC container to inject dependencies into our web service instances. Typically, a web service will rely on other objects, service objects for example, so being able to configure which service object implementation to use is very useful.

Last, but not least, the Spring.NET team did not like the fact that creating a web service is an implementation task. Most (although not all) services are best implemented as normal classes that use coarse-grained service interfaces, and the decision as to whether a particular service should be exposed as a remote object, web service, or even an enterprise (COM+) component, should only be a matter of configuration, and not implementation.

An example using the web service exporter can be found in quickstart example named 'calculator'. More information can be found here 'Web Services example'.

27.2.1. Removing the need for .asmx files

Unlike web pages, which use .aspx files to store presentation code, and code-behind classes for the logic, web services are completely implemented within the code-behind class. This means that .asmx files serve no useful purpose, and as such they should neither be necessary nor indeed required at all.

Spring.NET allows application developers to expose existing web services easily by registering a custom implementation of the `WebServiceHandlerFactory` class and by creating a standard Spring.NET object definition for the service.

By way of an example, consider the following web service...

```
namespace MyCompany.MyApp.Services
{
    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService
    {
        [WebMethod]
        public string HelloWorld()
        {
        }
    }
}
```

```

        return "Hello World!";
    }
}

```

This is just a standard class that has methods decorated with the `WebMethod` attribute and (at the class-level) the `WebService` attribute. Application developers can create this web service within Visual Studio just like any other class.

All that one need to do in order to publish this web service is:

1. Register the `Spring.Web.Services.WebServiceFactoryHandler` as the HTTP handler for `.asmx` requests within one's `web.config` file.*

```

<system.web>
  <httpHandlers>
    <add verb="*" path="*.asmx" type="Spring.Web.Services.WebServiceHandlerFactory, Spring.Web"/>
  </httpHandlers>
</system.web>

```

Of course, one can register any other extension as well, but typically there is no need as Spring.NET's handler factory will behave exactly the same as a standard handler factory if said handler factory cannot find the object definition for the specified service name. In that case the handler factory will simply look for an `.asmx` file.

If you are using IIS7 the following configuration is needed

```

<system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <handlers>
    <add name="SpringWebServiceSupport" verb="*" path="*.asmx" type="Spring.Web.Services.WebServiceHandlerFactory, Spring.Web.Services" />
  </handlers>
</system.webServer>

```

2. Create an object definition for one's web service.

```

<object name="HelloWorld" type="MyComany.MyApp.Services.HelloWorldService, MyAssembly" abstract="true"/>

```

Note that one is not absolutely required to make the web service object definition `abstract` (via the `abstract="true"` attribute), but this is a recommended best practice in order to avoid creating an unnecessary instance of the service. Because the .NET infrastructure creates instances of the target service object internally for each request, all Spring.NET needs to provide is the `System.Type` of the service class, which can be retrieved from the object definition even if it is marked as `abstract`.

That's pretty much it as we can access this web service using the value specified for the `name` attribute of the object definition as the service name:

```

http://localhost/MyWebApp/HelloWorld.asmx

```

27.2.2. Injecting dependencies into web services

For arguments sake, let's say that we want to change the implementation of the `HelloWorld` method to make the returned message configurable.

One way to do it would be to use some kind of message locator to retrieve an appropriate message, but that locator needs to be implemented. Also, it would certainly be an odd architecture that used dependency injection throughout the application to configure objects, but that resorted to the service locator approach when dealing with web services.

Ideally, one should be able to define a property for the message within one's web service class and have Spring.NET inject the message value into it:

```
namespace MyApp.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }

    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService : IHelloWorld
    {
        private string message;
        public string Message
        {
            set { message = value; }
        }

        [WebMethod]
        public string HelloWorld()
        {
            return this.message;
        }
    }
}
```

The problem with standard Spring.NET DI usage in this case is that Spring.NET does not control the instantiation of the web service. This happens deep in the internals of the .NET framework, thus making it quite difficult to plug in the code that will perform the configuration.

The solution is to create a dynamic server-side proxy that will wrap the web service and configure it. That way, the .NET framework gets a reference to a proxy type from Spring.NET and instantiates it. The proxy then asks a Spring.NET application context for the actual web service instance that will process requests.

This proxying requires that one export the web service explicitly using the `Spring.Web.Services.WebServiceExporter` class; in the specific case of this example, one must also not forget to configure the `Message` property for said service:

```
<object id="HelloWorld" type="MyApp.Services.HelloWorldService, MyApp">
  <property name="Message" value="Hello, World!"/>
</object>

<object id="HelloWorldExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="HelloWorld"/>
</object>
```

The `WebServiceExporter` copies the existing web service and method attribute values to the proxy implementation (if indeed any are defined). Please note however that existing values can be overridden by setting properties on the `WebServiceExporter`.



Interface Requirements

In order to support some advanced usage scenarios, such as the ability to expose an AOP proxy as a web service (allowing the addition of AOP advices to web service methods), Spring.NET requires those objects that need to be exported as web services to implement a (service) interface.

Only methods that belong to an interface will be exported by the `WebServiceExporter`.

27.2.3. Exposing PONO as Web Services

Now that we are generating a server-side proxy for the service, there is really no need for it to have all the attributes that web services need to have, such as `WebMethod`. Because .NET infrastructure code never really sees the "real" service, those attributes are redundant as the proxy needs to have them on its methods, because that's what .NET deals with, but they are not necessary on the target service's methods.

This means that we can safely remove the `WebService` and `WebMethod` attribute declarations from the service implementation, and what we are left with is a plain old .NET object (a PONO). The example above would still work, because the proxy generator will automatically add `WebMethod` attributes to all methods of the exported interfaces.

However, that is still not the ideal solution. You would lose information that the optional `WebService` and `WebMethod` attributes provide, such as service namespace, description, transaction mode, etc. One way to keep those values is to leave them within the service class and the proxy generator will simply copy them to the proxy class instead of creating empty ones, but that really does defeat the purpose.

To add specific attributes to the exported web service, you can set all the necessary values within the definition of the service exporter, like so...

```
<object id="HelloWorldExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="HelloWorld"/>
  <property name="Namespace" value="http://myCompany/services"/>
  <property name="Description" value="My exported HelloWorld web service"/>
  <property name="MemberAttributes">
    <dictionary>
      <entry key="HelloWorld">
        <object type="System.Web.Services.WebMethodAttribute, System.Web.Services">
          <property name="Description" value="My Spring-configured HelloWorld method."/>
          <property name="MessageName" value="ZdravoSvete"/>
        </object>
      </entry>
    </dictionary>
  </property>
</object>

// or, once configuration improvements are implemented...
<web:service targetName="HelloWorld" namespace="http://myCompany/services">
  <description>My exported HelloWorld web service.</description>
  <methods>
    <method name="HelloWorld" messageName="ZdravoSvete">
      <description>My Spring-configured HelloWorld method.</description>
    </method>
  </methods>
</web:service>
```

Based on the configuration above, Spring.NET will generate a web service proxy for all the interfaces implemented by a target and add attributes as necessary. This accomplishes the same goal while at the same time moving web service metadata from implementation class to configuration, which allows one to export pretty much *any* class as a web service.

The `WebServiceExporter` also has a `TypeAttributes IList` property for applying attributes at the type level.



Note

The attribute to confirm to the WSI basic profile 1.1 is not added by default. This will be added in a future release. In the meantime use the `TypeAttributes IList` property to add `[WebServiceBinding(ConformsTo=WsProfiles.BasicProfile1_1)]` to the generated proxy.

One can also export only certain interfaces that a service class implements by setting the `Interfaces` property of the `WebServiceExporter`.



Distributed Objects Warning

Distributed Objects Warning

Just because you *can* export any object as a web service, doesn't mean that you *should*. Distributed computing principles still apply and you need to make sure that your services are not chatty and that arguments and return values are `Serializable`.

You still need to exercise common sense when deciding whether to use web services (or remoting in general) at all, or if local service objects are all you need.

27.2.4. Exporting an AOP Proxy as a Web Service

It is often useful to be able to export an AOP proxy as a web service. For example, consider the case where you have a service that is wrapped with an AOP proxy that you want to access both locally and remotely (as a web service). The local client would simply obtain a reference to an AOP proxy directly, but any remote client needs to obtain a reference to an exported web service proxy, that delegates calls to an AOP proxy, that in turn delegates them to a target object while applying any configured AOP advice.

Effecting this setup is actually fairly straightforward; because an AOP proxy is an object just like any other object, all you need to do is set the `WebServiceExporter`'s `TargetName` property to the `id` (or indeed the `name` or `alias`) of the AOP proxy. The following code snippets show how to do this...

```
<object id="DebugAdvice" type="MyApp.AOP.DebugAdvice, MyApp" />

<object id="TimerAdvice" type="MyApp.AOP.TimerAdvice, MyApp" />

<object id="MyService" type="MyApp.Services.MyService, MyApp" />

<object id="MyServiceProxy" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="TargetName" value="MyService" />
  <property name="IsSingleton" value="true" />
  <property name="InterceptorNames">
    <list>
      <value>DebugAdvice</value>
      <value>TimerAdvice</value>
    </list>
  </property>
</object>

<object id="MyServiceExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="MyServiceProxy" />
  <property name="Name" value="MyService" />
  <property name="Namespace" value="http://myApp/webservices" />
  <property name="Description" value="My web service" />
</object>
```

That's it as every call to the methods of the exported web service will be intercepted by the target AOP proxy, which in turn will apply the configured debugging and timing advice to it.

27.3. Client-side

On the client side, the main objection the Spring.NET team has is that client code becomes tied to a proxy *class*, and not to a service *interface*. Unless you make the proxy class implement the service interface manually, as described by Juval Lowy in his book "Programming .NET Components", application code will be less flexible

and it becomes very difficult to plug in different service implementation in the case when one decides to use a new and improved web service implementation or a local service instead of a web service.

The goal for Spring.NET's web services support is to enable the easy generation of client-side proxies that implement a specific service interface.

27.3.1. Using VS.NET generated proxy

The problem with the web-service proxy classes that are generated by VS.NET or the WSDL command line utility is that they don't implement a service interface. This tightly couples client code with web services and makes it impossible to change the implementation at a later date without modifying and recompiling the client.

Spring.NET provides a simple `IFactoryObject` implementation that will generate a *"proxy for proxy"* (however obtuse that may sound). Basically, the `Spring.Web.Services.WebServiceProxyFactory` class will create a proxy for the VS.NET- / WSDL-generated proxy that implements a specified service interface (thus solving the problem with the web-service proxy classes mentioned in the preceding paragraph).

At this point, an example may well be more illustrative in conveying what is happening; consider the following interface definition that we wish to expose as a web service...

```
namespace MyCompany.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }
}
```

In order to be able to reference a web service endpoint through this interface, you need to add a definition similar to the example shown below to your client's application context:

```
<object id="HelloWorld" type="Spring.Web.Services.WebServiceProxyFactory, Spring.Services">
  <property name="ProxyType" value="MyCompany.WebServices.HelloWorld, MyClientApp"/>
  <property name="ServiceInterface" value="MyCompany.Services.IHelloWorld, MyServices"/>
</object>
```

What is important to notice is that the underlying implementation class for the web service does not have to implement the same `IHelloWorld` service interface... so long as matching methods with compliant signatures exist (a kind of duck typing), Spring.NET will be able to create a proxy and delegate method calls appropriately. If a matching method cannot be found, the Spring.NET infrastructure code will throw an exception.

That said, if you control both the client and the server it is probably a good idea to make sure that the web service class on the server implements the service interface, especially if you plan on exporting it using Spring.NET's `WebServiceExporter`, which requires an interface in order to work.

27.3.2. Generating proxies dynamically

The `WebServiceProxyFactory` can also dynamically generate a web-service proxy. The XML object definition for this factory object is shown below

```
<object id="calculatorService" type="Spring.Web.Services.WebServiceProxyFactory, Spring.Services">
  <property name="ServiceUri" value="http://myServer/Calculator/calculatorService.asmx"/>
  <!--<property name="ServiceUri" value="file://~/calculatorService.wsdl"/>-->
  <property name="ServiceInterface" value="Spring.Calculator.Interfaces.IAdvancedCalculator, Spring.Calculator.Contr
```

```

<!-- Dependency injection on Factory's product : the proxy instance of type SoapHttpClientProtocol -->
<property name="ProductTemplate">
  <object>
    <property name="Timeout" value="10000" /> <!-- 10s -->
  </object>
</property>
</object>

```

One use-case where this proxy is very useful is when dealing with typed data sets through a web service. Leaving the pros and cons of this approach aside, the current behavior of the proxy generator in .NET is to create wrapper types for the typed dataset. This not only pollutes the solution with extraneous classes but also results in multiple wrapper types being created, one for each web service that uses the typed dataset. This can quickly get confusing. The proxy created by Spring allows you to reference you typed datasets directly, avoiding the above mentioned issues.

27.3.3. Configuring the proxy instance

The `WebServiceProxyFactory` also implements the `interface`, `Spring.Objects.Factory.IConfigurableFactoryObject`, allowing to specify configuration for the product that the `WebServiceProxyFactory` creates. This is done by specifying the `ProductTemplate` property. This is particularly useful for securing the web service. An example is shown below.

```

<object id="PublicarAltasWebService" type="Spring.Web.Services.WebServiceProxyFactory, Spring.Services">
  <property name="ProxyType" value="My.WebService" />
  <property name="ServiceInterface" value="My.IWebServiceInterface" />
  <property name="ProductTemplate">
    <object>
      <!-- Configure the web service URL -->
      <property name="Url" value="https://localhost/MyApp/webservice.jws" />
      <!-- Configure the Username and password for the web service -->
      <property name="Credentials">
        <object type="System.Net.NetworkCredential, System">
          <property name="UserName" value="user"/>
          <property name="Password" value="password"/>
        </object>
      </property>
      <!-- Configure client certificate for the web service -->
      <property name="ClientCertificates">
        <list>
          <object id="MyCertificate" type="System.Security.Cryptography.X509Certificates.X509Certificate2, System">
            <constructor-arg name="fileName" value="Certificate.p12" />
            <constructor-arg name="password" value="notgoingtotellyou" />
          </object>
        </list>
      </property>
    </object>
  </property>
</object>

```

For an example of how using SOAP headers for authentication using the `WebServiceExporter` and `WebServiceProxyFactory`, refer to this [solution](#) on our wiki.

Chapter 28. Windows Communication Foundation (WCF)

28.1. Introduction

Spring's WCF support allows you to configure your WCF services via dependency injection and add additional behavior to them using Aspect-Oriented programming (AOP).

For those who would like to get their feet wet right way, check out the WcfQuickStart application in the examples directory.

28.2. Configuring WCF services via Dependency Injection

The technical approach used to perform dependency injection is based on dynamically creating an implementation of your service interface (a dynamic proxy) that retrieves a configured instance of your service type from the Spring container. This dynamic proxy is then the final service type that is hosted.



Note

An alternative implementation approach that uses extensibility points in WCF to delegate to Spring to create and configure your WCF service was tried but proved to be limited in its range of applicability. This approach was first taken (afaik) by Oran Dennison on his blog [<http://orand.blogspot.com/2006/10/wcf-service-dependency-injection.html>] and several other folks on the web since then. The issue in using this approach is that if the service is configured to be a singleton, for example using `[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]` then the invocation of the `IInstanceProvider` is short-circuited. See the notes on the MSDN class documentation here [<http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.iinstanceprovider.aspx>]. While this would be the preferred approach, no acceptable work around was found.

28.2.1. Dependency Injection

In this approach you develop your WCF services as you would normally do. For example here is a sample service type taken from the quickstart example.

```
[ServiceContract(Namespace = "http://Spring.WcfQuickStart")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
    [OperationContract]
    double Subtract(double n1, double n2);
    [OperationContract]
    double Multiply(double n1, double n2);
    [OperationContract]
    double Divide(double n1, double n2);
    [OperationContract]
    string GetName();
}
```

The implementation for the methods is fairly obvious but an additional property, `SleepInSeconds`, is present. This is the property we will configure via dependency injection. Here is a partial listing of the implementation

```

public class CalculatorService : ICalculator
{
    private int sleepInSeconds;

    public int SleepInSeconds
    {
        get { return sleepInSeconds; }
        set { sleepInSeconds = value; }
    }

    public double Add(double n1, double n2)
    {
        Thread.Sleep(sleepInSeconds*1000);
        return n1 + n2;
    }

    // additional implementation not shown for brevity
}

```

To configure this object with Spring, provide the XML configuration metadata as shown below as you would with any Spring managed object.

```

<object id="calculator" singleton="false" type="Spring.WcfQuickStart.CalculatorService, Spring.WcfQuickStart.ServerApp"
    <property name="SleepInSeconds" value="1" />
</object>

```



Note

The object must be declared as a 'prototype' object, i.e. not a singleton, in order to interact correctly with WCF instancing.

To host this service type in a standalone application define an instance of a `Spring.ServiceModel.Activation.ServiceHostFactoryObject` and set its property `TargetName` to the id value of the previously defined service type. `ServiceHostFactoryObject` is a Spring `IFactoryObject` implementation. (See here for more information on `IFactoryObjects` and their interaction with the container.) The `ServiceHostFactoryObject` will create an instance of `Spring.ServiceModel.Activation.SpringServiceHost` that will be the `ServiceHost` instance associated with your service type. This configuration for this step is shown below.

```

<object id="calculatorServiceHost" type="Spring.ServiceModel.Activation.ServiceHostFactoryObject, Spring.Services"
    <property name="TargetName" value="calculator" />
</object>

```

Additional service configuration can be done declaratively in the standard `App.config` file as shown below

```

<system.serviceModel>
  <services>
    <service name="calculator" behaviorConfiguration="DefaultBehavior">
      <host> ... </host>
      <endpoint> ... </endpoint>
    </service>
    ...
  </services>
</system.serviceModel>

```



Note

It is important that the name of the service in the WCF declarative configuration section match the name of the Spring object definition

`Spring.ServiceModel.Activation.SpringServiceHost` is where the dynamic proxy for your service type is generated. This dynamic proxy will implement a single 'WCF' interface, the same on that your service type implements. The implementation of the service interface methods on the proxy will delegate to a wrapped 'target' object which is the object instance retrieved by name from the Spring container using the Spring API, `ApplicationContext.GetObject(name)`. Since the object retrieved in this manner is fully configured, your WCF service is as well.

Outside of a standalone application you can also use the class `Spring.ServiceModel.Activation.ServiceHostFactory` (which inherits from `System.ServiceModel.Activation.ServiceHostFactory`) to host your services so that they can be configured via dependency injection. To use the dynamic proxy approach described here you should still refer to the name of the service as the name of the object definition used to configure the service type in the Spring container.

There are not many disadvantages to this approach other than the need to specify the service name as the name of the object definition in the Spring container and to ensure that `singleton=false` is used in the object definition. You can also use `Spring.ServiceModel.Activation.ServiceHostFactory` to host your service inside IIS but should still refer to the service by the name of the object in the Spring container.

28.3. Apply AOP advice to WCF services

In either approach to performing dependency injection you can apply additional AOP advice to your WCF services in the same way as you have always done in Spring. The following configuration shows how to apply some simple performance monitoring advice to all services in the `Spring.WcfQuickStart` namespace and is taken from the QuickStart example.

```
<object id="serviceOperation" type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut, Spring.Aop">
  <property name="pattern" value="Spring.WcfQuickStart.*"/>
</object>

<object id="perfAdvice" type="Spring.WcfQuickStart.SimplePerformanceInterceptor, Spring.WcfQuickStart.ServerApp">
  <property name="Prefix" value="Service Layer Performance"/>
</object>

<aop:config>
  <aop:advisor pointcut-ref="serviceOperation" advice-ref="perfAdvice"/>
</aop:config>
```

The `aop:config` section implicitly uses Spring's autoproxying features to add additional behavior to any objects defined in the container that match the pointcut criteria.

28.4. Creating client side proxies declaratively

To create a client side proxy based on the use of `ChannelFactory<T>`, you can use this rather ugly 'boiler plate' XML snippet that takes uses Spring's support for calling factory methods on object instances.

```
<!-- returns ChannelFactory<ICalculator>("calculatorEndpoint").CreateChannel() -->

<object id="serverAppCalculator" type="Spring.WcfQuickStart.ICalculator, Spring.WcfQuickStart.ClientApp"
  factory-object="serverAppCalculatorChannelFactory"
  factory-method="CreateChannel" />

<object id="serverAppCalculatorChannelFactory"
  type="System.ServiceModel.ChannelFactory<Spring.WcfQuickStart.ICalculator>, System.ServiceModel">

  <constructor-arg name="endpointConfigurationName" value="serverAppCalculatorEndpoint" />

</object>
```



Note

This will be shortened using a custom namespace in a future release

The value 'serverAppCalculatorEndpoint' refers to the name of an endpoints in the <client> section of the standard WCF configuration inside of App.config.

28.5. Exporting PONO's as WCF Services

Much like the approach taken for .asmx web services Spring provides an exporter that will add [ServiceContract] and [OperationContract] attributes by default to all public interface methods on a given (PONO) class. The exporter class is `Spring.ServiceModel.ServiceExporter` and has various options to fine-tune what interfaces are exported and the specific attributes that get applied to each method and on that class. Here is a simple example

```
<object id="HelloWorldExporter" type="Spring.ServiceModel.ServiceExporter, Spring.Services">
  <property name="TargetName" value="HelloWorld"/>
  <property name="MemberAttributes">
    <dictionary>
      <entry key="SayHelloWorld">
        <object type="System.ServiceModel.OperationContractAttribute, System.ServiceModel">
          <property name="IsOneWay" value="false"/>
          <!-- configure any other OperationContractAttribute properties here -->
        </object>
      </entry>
    </dictionary>
  </property>
</object>
```

Spring does not provide any means to add [DataContract] or [DataMember] attributes to method arguments of your service operations. As such, either you will do that yourself or you may choose to use a serializer other than `DataContractSerializer`, for example one that relies on method arguments that implement the `ISerializable` interface, having the [Serializable] attribute, or are serializable via the `XmlSerializer`. Use the latter serializers is a good way to migrate from an existing RCP based approach, such as using .NET remoting, to WCF in order to take advantage of the WCF runtime and avoid editing much existing code. You can then incrementally refactor and/or create new operations that use `DataContractSerializer`.

Part V. Integration

This part of the reference documentation covers the Spring Framework's integration with a number of related enterprise .NET technologies.

- Chapter 29, *Message Oriented Middleware - Apache ActiveMQ and TIBCO EMS*
- Chapter 30, *Message Oriented Middleware - TIBCO EMS*
- Chapter 31, *Message Oriented Middleware - MSMQ*
- Chapter 32, *Scheduling and Thread Pooling*
- Chapter 33, *Template Engine Support*

Chapter 29. Message Oriented Middleware - Apache ActiveMQ and TIBCO EMS

29.1. Introduction

The goal of Spring's messaging is to increase your productivity when writing an enterprise strength messaging middleware applications. Spring achieves these goals in several ways. First it provides several helper classes that remove from the developer the incidental complexity and resource management issues that arise when using messaging APIs. Second, the design of these messaging helper classes promote best practices in designing a messaging application by promoting a clear separation between the messaging middleware specific code and business processing that is technology agnostic. This is generally referred to a "plain old .NET object" (or PONO) programming model.

This chapter discusses Spring's messaging support for providers whose API was modeled after the Java Message Service (JMS) API. Vendors who provide a JMS inspired API include Apache ActiveMQ, TIBCO, IBM, and Progress Software. If you are using Microsoft's Message Queue, please refer to the specific MSMQ section. The description of Spring messages features in this chapter apply to all of these JMS vendors. However, the documentation focuses on showing code examples that use Apache ActiveMQ. For code examples and some features specific to TIBCO EMS please refer to this chapter.

29.1.1. Multiple Vendor Support

As there is no de facto-standard common API across messaging vendors, Spring provides an implementation of its helper classes for each of the major messaging middleware vendors. The naming of the classes you will interact with most frequently will either be identical for each provider, but located in a different namespace, or have their prefix change to be the three-letter-acronym commonly associated with the message provider. The list of providers supported by Spring is show below along with their namespace and prefix.

1. Apache ActiveMQ (NMS) in namespace `Spring.Messaging.Nms`. 'Nms' is used as the class prefix
2. TIBCO EMS in namespace `Spring.Messaging.Ems`. 'Ems' is used as the class prefix.
3. Websphere MQ in namespace `Spring.Messaging.Xms`, 'Xms' is used as the class prefix (in a future release)

JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages. For message production and the synchronous consumption of messages the a template class, named `NmsTemplate`, `EmsTemplate` (etc.) is used. Asynchronous message consumption is performed though a multi-threaded message listener container, `SimpleMessageListenerContainer`. This message listener container is used to create Message-Driven PONOs (MDPs) which refer to a messaging callback class that consists of just 'plain .NET object's and is devoid of any specific messaging types or other artifacts. The `IMessageConverter` interface is used by both the template class and the message listener container to convert between provider message types and PONOs.

The namespace `Spring.Messaging.<Vendor>.Core` contains the messaging template class (e.g. `NmsTemplate`). The template class simplifies the use of the messaging APIs by handling the creation and release of resources, much like the `AdoTemplate` does for ADO.NET. The JMS inspired APIs are low-level API, much like ADO.NET. As such, even the simplest of operations requires 10s of lines of code with the bulk of that code related to resource

management of intermediate API objects Spring's messaging support, both in Java and .NET, addresses the error-prone boiler plate coding style one needs when using these APIs.

The design principle common to Spring template classes is to provide helper methods to perform common operations and for more sophisticated usage, delegate the essence of the processing task to user implemented callback interfaces. The messaging template follows the same design. The message template class offer various convenience methods for the sending of messages, consuming a message synchronously, and exposing the message Session and MessageProducer to the user.

The namespace `Spring.Messaging.<VendorAcronym>.Support.Converter` provides a `IMessageConverter` abstraction to convert between .NET objects and messages. The namespace `Spring.Messaging.<VendorAcronym>.Support.Destinations` provides various strategies for managing destinations, such as providing a service locator for destinations stored in a directory service.

Finally, the namespace `Spring.Messaging.<VendorAcronym>.Connections` provides an implementations of the `ConnectionFactory` suitable for use in standalone applications.

The rest of the sections in this chapter discusses each of the major helper classes in detail. Please refer to the sample application that ships with Spring for additional hands-on usage.



Note

To simplify documenting features that are common across all provider implementations of Spring's helper classes a specific provider, Apache ActiveMQ, was selected. As such when you see 'NmsTemplate' in the documentation, it also refers to EmsTemplate, XmsTemplate, etc. unless specifically documented otherwise. The provider specific API classes are typically named after their JMS counterparts with the possible exception of a leading 'T' in front of interfaces in order to follow .NET naming conventions. In the documentation these API artifacts are referred to as 'ConnectionFactory', 'Session', 'Message', etc. without the leading 'T'.



Note

To view some of this chapters contents that are based on TIBCO EMS please refer to the TIBCO EMS chapter.

29.1.2. Separation of Concerns

The use of MessageConverters and a PONO programming model promote messaging best practices by applying the principal of Separation of Concerns to messaging based architectures. The infrastructure concern of publishing and consuming messages is separated from the concern of business processing. These two concerns are reflected in the architecture as two distinct layers, a message processing layer and a business processing layer. The benefit of this approach is that your business processing is decoupled from the messaging technology, making it more likely to survive technological changes over time and also easier to test. Spring's MessageConverters provides support for mapping messaging data types to PONOs. Aside from being the link between the two layers, MessageConverters provide a pluggable strategy to help support the evolution of a loosely coupled architecture over time. Message formats will change over time, typically by the addition of new fields. MessageConverters can be implemented to detect different versions of messages and perform the appropriate mapping logic to PONOs such so that multiple versions of a message can be supported simultaneously, a common requirement in enterprise messaging architectures.

29.1.3. Interoperability and provider portability

Messaging is a traditional area of Interoperability across heterogeneous systems with messaging vendors providing support on multiple operating systems (Windows, UNIX, Mainframes OS's) as well as multiple language bindings (C, C++, Java, .NET, Perl, etc.). In 199x the Java Community Process came up with a specification to provide a common API across messaging providers as well as define some common messaging functionality. This specification is known as the Java Message Service. From the API perspective, it can roughly be thought of as the messaging counterpart to the ADO.NET or JDBC APIs that provide portability across different database providers.

Given this history, when messaging vendors created their .NET APIs, many did so by creating their own JMS inspired API in .NET. There is no de facto-standard common API across messaging vendors. As such, portability across vendors using Spring's helper classes is done by changing the configuration schema in your configuration to a particular vendor and doing a 'search-and-replace' on the code base, changing the namespace and a few class names. While not ideal, using Spring will push you in the direction of isolating the messaging specific classes in its own layer and therefore will reduce the impact of the changes you make to the code when switch providers. Your business logic classes called into via Spring's messaging infrastructure will remain the same.

The NMS project from Apache addresses the lack of a common API across .NET messaging providers by providing an abstract interface based API for messaging and several implementations for different providers. At the time of this writing, the project is close to releasing a 1.0 version that supports ApacheMQ, MSMQ, and TIBCO EMS. There are a few outstanding issues at the moment that prevent one using NMS as a common API for all messaging providers but hopefully these issues will be resolved. Note, that NMS serves 'double' duty as the preferred API for messaging with ActiveMQ as well as a providing portability across different messaging providers.

29.1.4. The role of Messaging API in a 'WCF world'

Windows Communication Foundation (WCF) also supports message oriented middleware. Not surprisingly, a Microsoft Message Queuing (MSMQ) binding is provided as part of WCF. The WCF programming model is higher level than the traditional messaging APIs such as JMS and NMS since you are programming to a service interface and use metadata (either XML or attributes) to configure the messaging behavior. If you prefer to use this service-oriented, RPC style approach, then look to see if a vendor provides a WCF binding for your messaging provider. Note that even with the option of using WCF, many people prefer to sit 'closer to the metal' when using messaging middleware, to access specific features and functionality not available in WCF, or simply because they are more comfortable with that programming model.

A WCF binding for Apache NMS is being developed as a separate project under the Spring Extensions [<http://www.springframework.org/extensions/faq>] umbrella project. Stay tuned for details.

29.2. Using Spring Messaging

29.2.1. Messaging Template overview

Code that uses the messaging template classes (`NmsTemplate`, `EmsTemplate`, etc) only needs to implement callback interfaces giving them a clearly defined contract. The `IMessageCreator` callback interface creates a message given a Session provided by the calling code in `NmsTemplate`. In order to allow for more complex usage of the provider messaging API, the callback `ISessionCallback` provides the user with the provider specific messaging Session and the callback `IProducerCallback` exposes a provider specific Session and MessageProducer pair. See Section 29., "Session and Producer Callback".

Provider messaging APIs typically expose two types of send methods, one that takes delivery mode, priority, and time-to-live as quality of service (QOS) parameters and one that takes no QOS parameters which uses default values. Since there are many higher level send methods in `NmsTemplate`, the setting of the QOS parameters have been exposed as properties on the template class to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set using the property `ReceiveTimeout`.



Note

Instances of the `NmsTemplate` class are thread-safe once configured. This is important because it means that you can configure a single instance of a `NmsTemplate` and then safely inject this shared reference into multiple collaborators. To be clear, the `NmsTemplate` is stateful, in that it maintains a reference to a `ConnectionFactory`, but this state is not conversational state.

29.2.2. Connections

The `NmsTemplate` requires a reference to a `ConnectionFactory`. The `ConnectionFactory` serves as the entry point for working with the provider's messaging API. It is used by the client application as a factory to create connections to the messaging server and encapsulates various configuration parameters, many of which are vendor specific such as SSL configuration options.

To create a `ActiveMQ ConnectionFactory` define can create an object definition as shown

```
<object id="nmsConnectionFactory" type="Apache.NMS.ActiveMQ.ConnectionFactory, Apache.NMS.ActiveMQ">
  <constructor-arg index="0" value="tcp://localhost:61616"/>
</object>
```

`EmsTemplate` also requires a reference to a `ConnectionFactory`, however, it is not the 'native' `TIBCO.EMS.ConnectionFactory`. Instead the connection factory type is `Spring.Messaging.Ems.Common.IConnectionFactory`. See the documentation for TIBCO EMS supplier for more information here.

29.2.3. Caching Messaging Resources

The standard API usage of NMS and other JMS inspired APIs involves creating many intermediate objects. To send a message the following 'API' walk is performed

```
IConnectionFactory->IConnection->ISession->IMessageProducer->Send
```

Between the `ConnectionFactory` and the `Send` operation there are three intermediate objects that are created and destroyed. To optimise the resource usage and increase performance two implementations of `IConnectionFactory` are provided.

29.2.3.1. SingleConnectionFactory

`Spring.Messaging.Nms.Connections.SingleConnectionFactory` will return the same connection on all calls to `CreateConnection` and ignore calls to `Close`.

29.2.3.2. CachingConnectionFactory

`Spring.Messaging.Nms.Connections.CachingConnectionFactory` extends the functionality of `SingleConnectionFactory` and adds the caching of Sessions, MessageProducers, and MessageConsumers.

The initial cache size is set to 1, use the property `SessionCacheSize` to increase the number of cached sessions. Note that the number of actual cached sessions will be more than that number as sessions are cached based on their acknowledgment mode, so there can be up to 4 cached session instances when `SessionCacheSize` is set to one,

one for each `AcknowledgementMode`. `MessageProducers` and `MessageConsumers` are cached within their owning session and also take into account the unique properties of the producers and consumers when caching.

`MessageProducers` are cached based on their destination. `MessageConsumers` are cached based on a key composed of the destination, selector, `noLocal` delivery flag, and the durable subscription name (if creating durable consumers).

Here is an example configuration

```
<object id="connectionFactory" type="Spring.Messaging.Nms.Connections.CachingConnectionFactory, Spring.Messaging.Nms">
  <property name="SessionCacheSize" value="10" />
  <property name="TargetConnectionFactory">
    <object type="Apache.NMS.ActiveMQ.ConnectionFactory, Apache.NMS.ActiveMQ">
      <constructor-arg index="0" value="tcp://localhost:61616"/>
    </object>
  </property>
</object>
```

29.2.4. Dynamic Destination Management

In Java implementations of JMS, `Connections` and `Destinations` are 'administered objects' accessible through JNDI - a directory service much like ActiveDirectory. In .NET each vendor has selected a different approach to destination management. Some are JNDI inspired, allowing you to retrieve `Connections` and `Destinations` that were configured administratively. You can use these vendor specific APIs to perform dependency injection on references to JMS `Destination` objects in Spring's XML configuration file by creating an implementation of `IObjectFactory` or alternatively configuring the specific concrete class implementation for a messaging provider.

However, this approach of administered objects can be quite cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the messaging provider. Examples of such advanced destination management would be the creation of dynamic destinations or support for a hierarchical namespace of destinations. The `NmsTemplate` delegates the resolution of a destination name to a destination object by delegating to an implementation of the interface `IDestinationResolver`. `DynamicDestinationResolver` is the default implementation used by `NmsTemplate` and accommodates resolving dynamic destinations.

Quite often the destinations used in a messaging application are only known at runtime and therefore cannot be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well-known naming convention. Even though the creation of dynamic destinations are not part of the original JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a name defined by the user which differentiates them from temporary destinations and are often not registered in a directory service. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor specific. However, a simple implementation choice that is sometimes made by vendors is to use the `TopicSession` method `CreateTopic(string topicName)` or the `QueueSession` method `CreateQueue(string queueName)` to create a new destination with default destination properties. Depending on the vendor implementation, `DynamicDestinationResolver` may then also create a physical destination instead of only resolving one.

The boolean property `PubSubDomain` is used to configure the `NmsTemplate` with knowledge of what messaging 'domain' is being used. By default the value of this property is false, indicating that the point-to-point domain, `Queues`, will be used. This property is infrequently used as the provider messaging APIs are now largely agnostic as to which messaging 'domain' is used, referring to 'Destinations' rather than 'Queues' or 'Topics'. However, this property does influence the behavior of dynamic destination resolution via implementations of the `IDestinationResolver` interface.

You can also configure the [NmsTemplate](#) with a default destination via the property `DefaultDestination`. The default destination will be used with send and receive operations that do not refer to a specific destination.

29.2.5. Message Listener Containers

One of the most common uses of JMS is to concurrently process messages delivered asynchronously. A message listener container is used to receive messages from a message queue and drive the `IMessageListener` that is injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an Message-Driven PONO (MDP) and a messaging provider, and takes care of registering to receive messages, resource acquisition and release, exception conversion and suchlike. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegates boilerplate messaging infrastructure concerns to the framework.

A subclass of `AbstractMessageListenerContainer` is used to receive messages from JMS and drive the Message-Driven PONOs (MDPs) that are injected into it. There are one subclasses of `AbstractMessageListenerContainer` packaged with Spring - `SimpleMessageListenerContainer`. Additional subclasses, in particular to participate in distributed transactions (if the provider supports it), will be provided in future releases. `SimpleMessageListenerContainer` creates a fixed number of JMS sessions at startup and uses them throughout the lifespan of the container.

Creating and configuring a ActiveMQ `MessageListener` container is described in this section.

29.2.6. Transaction Management

Spring provides an implementation of the `IPlatformTransactionManager` interface for managing ActiveMQ messaging transactions. The class is `NmsTransactionManager` and it manages transactions for a single `ConnectionFactory`. This allows messaging applications to leverage the managed transaction features of Spring as described in Chapter 17, *Transaction management*. The `NmsTransactionManager` performs local resource transactions, binding a `Connection/Session` pair from the specified `ConnectionFactory` to the thread. `NmsTemplate` automatically detects such transactional resources and operates on them accordingly.

Using Spring's `SingleConnectionFactory` will result in a shared `Connection`, with each transaction having its own independent `Session`.

29.3. Sending a Message

The `NmsTemplate` contains three convenience methods to send a message. The methods are listed below.

- `void Send(IDestination destination, IMessageCreator messageCreator)`
- `void Send(string destinationName, IMessageCreator messageCreator)`
- `void Send(IMessageCreator messageCreator)`

The method differ in how the destination is specified. In first case the JMS `Destination` object is specified directly. The second case specifies the destination using a string that is then resolved to a messaging `Destination` object using the `IDestinationResolver` associated with the template. The last method sends the message to the destination specified by `NmsTemplate`'s `DefaultDestination` property.

All methods take as an argument an instance of `IMessageCreator` which defines the API contract for you to create the JMS message. The interface is show below

```
public interface IMessageCreator {
    IMessage CreateMessage(ISession session);
}
```

Intermediate Sessions and MessageProducers needed to send the message are managed by `NmsTemplate`. The session passed in to the method is never null. There is a similar set methods that use a delegate instead of the interface, which can be convenient when writing small implementation in .NET 2.0 using anonymous delegates. Larger, more complex implementations of the method 'CreateMessage' are better suited to an interface based implementation.

- `void SendWithDelegate(IDestination destination, MessageCreatorDelegate messageCreatorDelegate)`
- `void SendWithDelegate(string destinationName, MessageCreatorDelegate messageCreatorDelegate)`
- `void SendWithDelegate(MessageCreatorDelegate messageCreatorDelegate)`

The declaration of the delegate is

```
public delegate IMessage MessageCreatorDelegate(ISession session);
```

The following class shows how to use the `SendWithDelegate` method with an anonymous delegate to create a `MapMessage` from the supplied Session object. The use of the anonymous delegate allows for very terse syntax and easy access to local variables. The `NmsTemplate` is constructed by passing a reference to a `ConnectionFactory`.

```
public class SimplePublisher
{
    private NmsTemplate template;

    public SimplePublisher()
    {
        template = new NmsTemplate(new ConnectionFactory("tcp://localhost:61616"));
    }

    public void Publish(string ticker, double price)
    {
        template.SendWithDelegate("APP.STOCK.MARKETDATA",
            delegate(ISession session)
            {
                IMapMessage message = session.CreateMapMessage();
                message.Body.SetString("TICKER", ticker);
                message.Body.SetDouble("PRICE", price);
                message.NMSPriority = MsgPriority.Low;
                return message;
            });
    }
}
```

A zero argument constructor and `ConnectionFactory` property are also provided. Alternatively consider deriving from Spring's `NmsGatewaySupport` convenience base class which provides a `ConnectionFactory` property that will instantiate a `NmsTemplate` instance that is made available via the property `NmsTemplate`.

29.3.1. Using MessageConverters

In order to facilitate the sending of domain model objects, the `NmsTemplate` has various send methods that take a .NET object as an argument for a message's data content. The overloaded methods `ConvertAndSend` and `ReceiveAndConvert` in `NmsTemplate` delegate the conversion process to an instance of the `IMessageConverter` interface. This interface defines a simple contract to convert between .NET objects and JMS messages. The default implementation `SimpleMessageConverter` supports conversion between [String](#) and [TextMessage](#), [byte\[\]](#)

and [BytesMessage](#), and [System.Collections.IDictionary](#) and [MapMessage](#). By using the converter, you and your application code can focus on the business object that is being sent or received via messaging and not be concerned with the details of how it is represented as a JMS message. There is also an `XmlMessageConverter` that converts objects to an XML string and vice-versa for sending via a `TextMessage`. Please refer to the API documentation and example application for more information on configuring an [XmlMessageConverter](#).

The family of `ConvertAndSend` messages are similar to that of the `Send` method with the additional argument of type [IMessagePostProcessor](#). These methods are listed below.

- `void ConvertAndSend(object message)`
- `void ConvertAndSend(object message, IMessagePostProcessor postProcessor)`
- `void ConvertAndSend(string destinationName, object message)`
- `void ConvertAndSend(string destinationName, object message, IMessagePostProcessor postProcessor);`
- `void ConvertAndSend(Destination destination, object message)`
- `void ConvertAndSend(Destination destination, object message, IMessagePostProcessor postProcessor)`

The example below uses the default message converter to send a `Hashtable` as a message to the destination "APP.STOCK".

```
public void PublishUsingDict(string ticker, double price)
{
    IDictionary marketData = new Hashtable();
    marketData.Add("TICKER", ticker);
    marketData.Add("PRICE", price);
    template.ConvertAndSend("APP.STOCK.MARKETDATA", marketData);
}
```

To accommodate the setting of message's properties, headers, and body that can not be generally encapsulated inside a converter class, the `IMessageConverterPostProcessor` interface gives you access to the message after it has been converted but before it is sent. The example below demonstrates how to modify a message header and a property after a `Hashtable` is converted to a message using the `IMessagePostProcessor`. The methods `ConvertAndSendUsingDelegate` allow for the use of a delegate to perform message post processing. This family of methods is listed below

- `void ConvertAndSendWithDelegate(object message, MessagePostProcessorDelegate postProcessor)`
- `void ConvertAndSendWithDelegate(IDestination destination, object message, MessagePostProcessorDelegate postProcessor)`
- `void ConvertAndSendWithDelegate(string destinationName, object message, MessagePostProcessorDelegate postProcessor)`

The declaration of the delegate is

```
public delegate IMessage MessagePostProcessorDelegate(IMessage message);
```

The following code shows this in action.

```
public void PublishUsingDict(string ticker, double price)
```



```
{
    IDictionary marketData = new Hashtable();
    marketData.Add("TICKER", ticker);
    marketData.Add("PRICE", price);
    template.ConvertAndSendWithDelegate("APP.STOCK.MARKETDATA", marketData,
        delegate(IMessage message)
        {
            message.NMSPriority = MsgPriority.Low;
            message.NMSCorrelationID = new Guid().ToString();
            return message;
        });
}
```

29.. Session and Producer Callback

While the send operations cover many common usage scenarios, there are cases when you want to perform multiple operations on a JMS Session or MessageProducer. The SessionCallback and ProducerCallback expose the Session and Session / MessageProducer pair respectfully. The Execute() methods on NmsTemplate execute these callback methods.

- `public object Execute(IProducerCallback action)`
- `public object Execute(ProducerDelegate action)`
- `public object Execute(ISessionCallback action)`
- `public object Execute(SessionDelegate action)`

Where ISessionCallback and IProducerCallback are

```
public interface IProducerCallback
{
    object DoInNms(ISession session, IMessageProducer producer);
}
```

and

```
public interface ISessionCallback
{
    object DoInNms(ISession session);
}
```

The delegate signatures are listed below and mirror the interface method signature

```
public delegate object SessionDelegate(ISession session);

public delegate object ProducerDelegate(ISession session, IMessageProducer producer);
```

29.5. Receiving a message

29.5.1. Synchronous Reception

While messaging middleware is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded `Receive(...)` methods on `NmsTemplate` provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property `ReceiveTimeout` on `NmsTemplate` specifies how long the receiver should wait before giving up waiting for a message.

The `Receive` methods are listed below

- `public Message Receive()`
- `public Message Receive(Destination destination)`
- `public Message Receive(string destinationName)`
- `public Message ReceiveSelected(string messageSelector)`
- `public Message ReceiveSelected(string destinationName, string messageSelector)`
- `public Message ReceiveSelected(Destination destination, string messageSelector)`

The `Receive` method without arguments will use the `DefaultDestination`. The `ReceiveSelected` methods apply the provided message selector string to the `MessageConsumer` that is created.

The `ReceiveAndConvert` methods apply the template's message converter when receiving a message. The message converter to use is set using the property `MessageConverter` and is the `SimpleMessageConverter` implementation by default. These methods are listed below.

- `public object ReceiveAndConvert()`
- `public object ReceiveAndConvert(Destination destination)`
- `public object ReceiveAndConvert(string destinationName)`
- `public object ReceiveSelectedAndConvert(string messageSelector)`
- `public object ReceiveSelectedAndConvert(string destinationName, string messageSelector)`
- `public object ReceiveSelectedAndConvert(Destination destination, string messageSelector)`

29.5.2. Asynchronous Reception

Asynchronous reception of messages occurs by the messaging provider invoking a callback function. This is commonly an interface such as the [IMessageListener](#) interface shown below, taken from the TIBCO EMS provider.

```
public interface IMessageListener
{
    void OnMessage(Message message);
}
```

Other vendors may provide a delegate based version of this callback or even both a delegate and interface options. Apache ActiveMQ supports the use of delegates for message reception callbacks. As a programming convenience in `Spring.Messaging.Nms.Core` is an interface `IMessageListener` that can be used with NMS.

Below is a simple implementation of the `IMessageListener` interface that processes a message.

```
using Spring.Messaging.Nms.Core;
using Apache.NMS;
using Common.Logging;

namespace MyApp
{
    public class SimpleMessageListener : IMessageListener
    {
        private static readonly ILog LOG = LogManager.GetLogger(typeof(SimpleMessageListener));
```

```

private int messageCount;

public int MessageCount
{
    get { return messageCount; }
}

public void OnMessage(IMessage message)
{
    messageCount++;
    LOG.Debug("Message listener count = " + messageCount);
    ITextMessage textMessage = message as ITextMessage;
    if (textMessage != null)
    {
        LOG.Info("Message Text = " + textMessage.Text);
    } else
    {
        LOG.Warn("Can not process message of type " + message.GetType());
    }
}
}

```

Once you've implemented your message listener, it's time to create a message listener container.

You register your listener with a message listener container that specifies various messaging configuration parameters, such as the `ConnectionFactory`, and the number of concurrent consumers to create. There is an abstract base class for message listener containers, `AbstractMessageListenerContainer`, and one concrete implementation, `SimpleMessageListenerContainer`. `SimpleMessageListenerContainer` creates a fixed number of JMS Sessions/MessageConsumer pairs as set by the property `ConcurrentConsumers`. The default value of `ConcurrentConsumers` is one. Here is a sample configuration that uses the custom schema provided in Spring.NET to more easily configure `MessageListenerContainers`.

```

<objects xmlns="http://www.springframework.net"
    xmlns:nms="http://www.springframework.net/nms">

    <object id="ActiveMqConnectionFactory" type="Apache.NMS.ActiveMQ.ConnectionFactory, Apache.NMS.ActiveMQ">
        <constructor-arg index="0" value="tcp://localhost:61616"/>
    </object>

    <object id="ConnectionFactory" type="Spring.Messaging.Nms.Connections.CachingConnectionFactory, Spring.Messaging.Nms">
        <constructor-arg index="0" ref="ActiveMqConnectionFactory"/>
        <property name="SessionCacheSize" value="10"/>
    </object>

    <object id="MyMessageListener" type="MyApp.SimpleMessageListener, MyApp"/>

    <nms:listener-container connection-factory="ConnectionFactory" concurrency="10">
        <nms:listener ref="MyMessageListener" destination="APP.STOCK.REQUEST" />
    </nms:listener-container>

</objects>

```

The above configuration will create 10 threads that process messages off of the queue named "APP.STOCK.REQUEST". The threads are those owned by the messaging provider as a result of creating a `MessageConsumer`. Other important properties are `ClientID`, used to set the `ClientID` of the `Connection` and `MessageSelector` to specify the 'sql-like' message selector string. Durable subscriptions are supported via the properties `SubscriptionDurable` and `DurableSubscriptionName`. You may also register an exception listener using the property `ExceptionListener`.

Exceptions that are thrown during message processing can be passed to an implementation of `ExceptionHandler` and registered with the container via the property `ExceptionListener`. The registered `ExceptionHandler` will be invoked if the exception is of the type `NMSException` (or the equivalent root exception type for other providers). The `SimpleMessageListenerContainer` will log the exception at error level and not propagate the exception to

the provider. All handling of acknowledgement and/or transactions is done by the listener container. You can override the method `HandleListenerException` to change this behavior.

Please refer to the Spring SDK documentation for additional description of the features and properties of `SimpleMessageListenerContainer`.

29.5.3. The `ISessionAwareMessageListener` interface

The `ISessionAwareMessageListener` interface is a Spring-specific interface that provides a similar contract to the messaging provider's `IMessageListener` interface or Listener delegate/event, but also provides the message handling method with access to the Session from which the Message was received.

```
public interface ISessionAwareMessageListener
{
    void OnMessage(IMessage message, ISession session);
}
```

You can also choose to implement this interface and register it with the message listener container

29.5.4. `MessageListenerAdapter`

The `MessageListenerAdapter` class is the final component in Spring's asynchronous messaging support: in a nutshell, it allows you to expose almost any class to be invoked as a messaging callback (there are of course some constraints).

Consider the following interface definition. Notice that although the interface extends neither the `IMessageListener` nor `ISessionAwareMessageListener` interfaces, it can still be used as a Message-Driven PONO (MDP) via the use of the `MessageListenerAdapter` class. Notice also how the various message handling methods are strongly typed according to the contents of the various Message types that they can receive and handle.

```
public interface MessageHandler {

    void HandleMessage(string message);

    void HandleMessage(Hashtable message);

    void HandleMessage(byte[] message);

}
```

and a class that implements this interface...

```
public class DefaultMessageHandler : IMessageHandler {
    // stub implementations elided for brevity...
}
```

In particular, note how the above implementation of the `IMessageHandler` interface (the above `DefaultMessageHandler` class) has no messaging provider API dependencies at all. It truly is a PONO that we will make into an MDP via the following configuration.

```
<object id="MessageHandler" type="MyApp.DefaultMessageHandler, MyApp"/>

<object id="MessageListenerAdapter" type="Spring.Messaging.Nms.Listener.Adapter.MessageListenerAdapter, Spring.Messaging.Nms"
  <property name="HandlerObject" ref="MessageHandler"/>
</object>

<object id="MessageListenerContainer" type="Spring.Messaging.Nms.Listener.SimpleMessageListenerContainer, Spring.Messaging.Nms"
  <property name="ConnectionFactory" ref="ConnectionFactory"/>
  <property name="DestinationName" value="APP.REQUEST"/>
</object>
```

```
<property name="MessageListener" ref="MessageListenerAdapter"/>
</object>
```

The previous examples relies on the fact that the default `IMessageConverter` implementation of the `MessageListenerAdapter` is `SimpleMessageConverter` that can convert from messages to strings, `byte[]`, and hashtables and object from a `ITextMessage`, `IBytesMessage`, `IMapMessage`, and `IObjectMessage` respectfully.

Below is an example of another MDP that can only handle the receiving of NMS `ITextMessage` messages. Notice how the message handling method is actually called 'Receive' (the name of the message handling method in a `MessageListenerAdapter` defaults to 'HandleMessage'), but it is configurable (as you will see below). Notice also how the 'Receive(..)' method is strongly typed to receive and respond only to NMS `ITextMessage` messages.

```
public interface TextMessageHandler {

    void Receive(ITextMessage message);

}
```

```
public class TextMessageHandler implements ITextMessageHandler {
    // implementation elided for clarity...
}
```

The configuration of the attendant `MessageListenerAdapter` would look like this

```
<object id="MessaggleHandler" type="MyApp.DefaultMessageHandler, MyApp"/>

<object id="MessageListenerAdapter" type="Spring.Messaging.Nms.Listener.Adapter.MessageListenerAdapter, Spring.Messaging.Nms"
    <property name="HandlerObject" ref="TextMessaggleHandler"/>
    <property name="DefaultHandlerMethod" value="Receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="MessageConverter">
        <null/>
    </property>
</object>
```

Please note that if the above 'MessageListener' receives a Message of a type other than `ITextMessage`, a `ListenerExecutionFailedException` will be thrown (and subsequently handled by the container by logging the exception).

If your `IMessageConverter` implementation will return multiple object types, overloading the handler method is perfectly acceptable, the most specific matching method will be used. A method with an object signature would be consider a 'catch-all' method of last resort. For example, you can have an handler interface as shown below.

```
public interface IMyHandler
{
    void DoWork(string text);
    void DoWork(OrderRequest orderRequest);
    void DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

Another of the capabilities of the `MessageListenerAdapter` class is the ability to automatically send back a response Message if a handler method returns a non-void value. The adapter's message converter will be used to convert the methods return value to a message. The resulting message will then be sent to the Destination defined in the JMS Reply-To property of the original Message (if one exists) , or the default Destination set on the `MessageListenerAdapter` (if one has been configured). If no Destination is found then an `InvalidDestinationException` will be thrown (and please note that this exception will not be swallowed and will propagate up the call stack).

An interface that is typical when used with a message converter that supports multiple object types and has return values is shown below.

```
public interface IMyHandler
{
    string DoWork(string text);
    OrderResponse DoWork(OrderRequest orderRequest);
    InvoiceResponse DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

29.5.5. Processing messages within a messaging transaction

Invoking a message listener within a transaction only requires reconfiguration of the listener container. Local message transactions can be activated by setting the property `SessionAcknowledgeMode` which for NMS is of the enum type `AcknowledgementMode`, to `AcknowledgementMode.Transactionnal`. Each message listener invocation will then operate within an active messaging transaction, with message reception rolled back in case of listener execution failure.

Sending a response message (via `ISessionAwareMessageListener`) will be part of the same local transaction, but any other resource operations (such as database access) will operate independently. This usually requires duplicate message detection in the listener implementation, covering the case where database processing has committed but message processing failed to commit. See the discussion on the ActiveMQ web site [here](#) for more information combining local database and messaging transactions.

29.5.6. Messaging Namespace support

To use the NMS namespace elements you will need to reference the NMS schema. When using TIBCO EMS you should refer to the TIBCO EMS Schema. For information on how to set this up refer to Section B.2.6, “The nms messaging schema”. The namespace consists of one top level elements: `<listener-container/>` which can contain one or more `<listener/>` child elements. Here is an example of a basic configuration for two listeners.

```
<nms:listener-container>

    <nms:listener destination="queue.orders" ref="OrderService" method="PlaceOrder"/>

    <nms:listener destination="queue.confirmations" ref="ConfirmationLogger" method="Log"/>

</nms:listener-container>
```

The example above is equivalent to creating two distinct listener container object definitions and two distinct `MessageListenerAdapter` object definitions as demonstrated in the section entitled Section 29.5.4, “`MessageListenerAdapater`”. In addition to the attributes shown above, the listener element may contain several optional ones. The following table describes all available attributes:

Table 29.1. Attributes of the NMS `<listener>` element

Attribute	Description
id	A object name for the hosting listener container. If not specified, a object name will be automatically generated.
destination (required)	The destination name for this listener, resolved through the <code>IDestinationResolver</code> strategy.
ref (required)	The object name of the handler object.
method	The name of the handler method to invoke. If the <code>ref</code> points to a <code>IMessageListener</code> or Spring <code>ISessionAwareMessageListener</code> , this attribute may be omitted.

Attribute	Description
response-destination	The name of the default response destination to send response messages to. This will be applied in case of a request message that does not carry a "NMSReplyTo" field. The type of this destination will be determined by the listener-container's "destination-type" attribute. Note: This only applies to a listener method with a return value, for which each result object will be converted into a response message.
subscription	The name of the durable subscription, if any.
selector	An optional message selector for this listener.
pubsub-domain	An optional boolean value. Set to true for the publish-subscribe domain (Topics) or false (the default) for point-to-point domain (Queues). This is useful when using the default implementation for destination resolvers.

The `<listener-container/>` element also accepts several optional attributes. This allows for customization of the various strategies (for example, `DestinationResolver`) as well as basic messaging settings and resource references. Using these attributes, it is possible to define highly-customized listener containers while still benefiting from the convenience of the namespace.

```
<nms:listener-container connection-factory="MyConnectionFactory"
    destination-resolver="MyDestinationResolver"
    concurrency="10">

  <nms:listener destination="queue.orders" ref="OrderService" method="PlaceOrder"/>

  <nms:listener destination="queue.confirmations" ref="ConfirmationLogger" method="Log"/>

</nms:listener-container>
```

The following table describes all available attributes. Consult the class-level SDK documentation of the `AbstractMessageListenerContainer` and its subclass `SimpleMessageListenerContainer` for more detail on the individual properties.

Table 29.2. Attributes of the NMS `<listener-container>` element

Attribute	Description
connection-factory	A reference to the NMS <code>ConnectionFactory</code> object (the default object name is 'ConnectionFactory').
destination-resolver	A reference to the <code>IDestinationResolver</code> strategy for resolving JMS Destinations.
message-converter	A reference to the <code>IMessageConverter</code> strategy for converting NMS Messages to listener method arguments. Default is a <code>SimpleMessageConverter</code> .
destination-type	The NMS destination type for this listener: <code>queue</code> , <code>topic</code> or <code>durableTopic</code> . The default is <code>queue</code> .
client-id	The NMS client id for this listener container. Needs to be specified when using durable subscriptions.
acknowledge	The native NMS acknowledge mode: <code>auto</code> , <code>client</code> , <code>dups-ok</code> or <code>transacted</code> . A value of <code>transacted</code> activates a locally transacted

Attribute	Description
	Session. As an alternative, specify the <code>transaction-manager</code> attribute described below. Default is <code>auto</code> .
concurrency	The number of concurrent sessions/consumers to start for each listener. Default is 1; keep concurrency limited to 1 in case of a topic listener or if queue ordering is important; consider raising it for general queues.
recovery-interval	The time interval between connection recovery attempts. The default is 5 seconds. Specify as a <code>TimeSpan</code> value using Spring's <code>TimeSpanConverter</code> (e.g. 10s, 10m, 3h, etc)
max-recovery-time	The maximum time try reconnection attempts. The default is 10 minutes. Specify as a <code>TimeSpan</code> value using Spring's <code>TimeSpanConverter</code> (e.g. 10s, 10m, 3h, etc)
auto-startup	Set whether to automatically start the listeners after initialization. Default is true, optionally set to false.
error-handler	A reference to a <code>ErrorHandler</code> that will handle any uncaught exceptions other than those of the type <code>NMSEException</code> (in the case of ActiveMQ or <code>EMSEException</code> in the case of TIBCO EMS) that may occur during the execution of the message listener. By default no <code>ErrorHandler</code> is registered and that error-level logging is the default behavior.
exception-listener	A reference to an <code>Spring.Messaging.Nms.Core.IExceptionListener</code> or <code>TIBCO.EMS.IExceptionListener</code> as appropriate. Is invoked in case of a <code>NMSEException</code> or <code>EMSEException</code> .

Chapter 30. Message Oriented Middleware - TIBCO EMS

30.1. Introduction

The bulk of the documentation for Spring's JMS support, independent of vendor, is described in the chapter [Chapter 29, Message Oriented Middleware - Apache ActiveMQ and TIBCO EMS](#). While that chapter refers to classes that are part of Spring's ActiveMQ integration, those classes have counter parts as part of Spring's TIBCO EMS integration. For example, `Spring.Messaging.Nms.Core.NmsTemplate` and `Spring.Messaging.Ems.Core.EmsTemplate`. This chapter fills in some of the gaps in taking that approach by describing Spring.NET features that are specific to its integration with TIBCO EMS and showing some examples using the TIBCO EMS integration.



Note

A complete sample application using Spring's EMS integration classes is in the distribution under the directory `examples\Spring\Spring.EmsQuickStart`. Documentation for the Quickstart is available [here](#).

30.2. Interface based APIs

The TIBCO EMS APIs are not interface based. What this means is that the class [TIBCO.EMS.Session](#) does not inherit from an [ISession](#) interface. The lack of interfaces makes it impossible to apply traditional approaches to support caching of Connections, Sessions, MessageProducers, and MessageProducers. Also, in some cases Java like setter methods were used instead of standard .NET properties making it difficult to configure those classes using dependency injection. (For example, see [EmssslSystemStoreInfo.SetCertificateStoreLocation\(\)](#)). For these reasons it was decided to create a 'mirror' API of the TIBCO EMS API that is interface based. In the namespace `Spring.Messaging.Ems.Common` are interfaces such as [IConnectionFactory](#), [IConnection](#), [ISession](#), [IMessageProducer](#), etc as well as their implementation classes [EmsConnectionFactory](#), [EmsConnection](#), [EmsSession](#), etc. The interfaces mirror all the operations that are on the standard TIBCO EMS classes so you should feel right as home when programming against these classes.

Typically users of Spring.NET do not need to programmatically interact with these classes, instead using methods of [EmsTemplate](#) to synchronously send and consume messages and a [SimpleMessageListenerContainer](#) to asynchronously consume messages. It will be common to configure an [Spring.Messaging.Ems.Common.ConnectionFactory](#) using dependency injection. The following sections show some example usage. You can also set or get the underlying 'native' TIBCO EMS object, such as the `TIBCO.EMS.ConnectionFactory` using a property 'NativeConnectionFactory'. Each class in the `Spring.Messaging.Ems.Common` namespace has a similar 'Native' property, for example `NativeSession`, `NativeMessageProducer` if you need access the raw TIBCO EMS class.

30.3. Using Spring's EMS based Messaging

30.3.1. Overview

In the namespace `Spring.Messaging.Ems.Core` is the class `EmsTemplate`. This is the main class you will use to send messages and to receive messages synchronously. In the namespace `Spring.Messaging.Ems.Listener` is the class `SimpleMessageListenerContainer`. This is the main class you will use to receive messages asynchronously.

30.3.2. Connections

To create a `Spring.Messaging.Ems.Common.ConnectionFactory` use the following object definition

```
<object id="emsConnectionFactory" type="Spring.Messaging.Ems.Common.EmsConnectionFactory, Spring.Messaging.Ems">
  <constructor-arg name="serverUrl" value="tcp://localhost:7222"/>
  <constructor-arg name="clientId" value="SpringEMSClient"/>
  <property name="ConnAttemptCount" value="10" />
  <property name="ConnAttemptDelay" value="100" />
  <property name="ConnAttemptTimeout" value="1000" />
</object>
```

Please refer to the API documentation for other properties you may want to set, in particular for those relating to SSL.

30.3.3. Caching Messaging Resources

While TIBCO EMS provides thread safe access to EMS Sessions (above and beyond what is specified in the JMS specification), Spring provides two implementations of the `ConnectionFactory` infrastructure to manage the use of intermediate objects when following the 'standard' API walk of

```
ConnectionFactory->Connection->Session->MessageProducer->Send
```

30.3.3.1. SingleConnectionFactory

`Spring.Messaging.Ems.Connections.SingleConnectionFactory` will return the same connection on all calls to `CreateConnection` and ignore calls to `Close`.

You can configure a `SingleConnectionFactory` as you would an `EmsConnectionFactory`.

30.3.3.2. CachingConnectionFactory

`Spring.Messaging.Ems.Connections.CachingConnectionFactory` extends the functionality of `SingleConnectionFactory` and adds the caching of Sessions, MessageProducers, and MessageConsumers. See the documentation for ActiveMQ `CachingConnectionFactory` for some additional information here.

An example configuration is shown below

```
<object id="connectionFactory" type="Spring.Messaging.Ems.Connections.CachingConnectionFactory, Spring.Messaging.Ems">
  <property name="SessionCacheSize" value="10" />
  <property name="TargetConnectionFactory" ref="emsConnectionFactory" />
</object>
```

Notice that the property `TargetConnectionFactory` refers to 'emsConnectionFactory' defined in the previous section. This connection factory implementation also set the `ReconnectOnException` property to true by default allowing for automatic recovery of the underlying Connection.



Note

The `CachingConnectionFactory` requires explicit closing of all Sessions obtained from its shared Connection. This is the usual recommendation for native EMS access code anyway and Spring EMS code follows this recommendation. However, with the `CachingConnectionFactory`, its use is mandatory in order to actually allow for Session reuse.



Note

MessageConsumers obtained from a cached Session won't get closed until the Session will eventually be removed from the pool. This may lead to semantic side effects in some cases. For a durable subscriber, the logical `Session.Close()` call will also close the subscription. Re-registering a durable consumer for the same subscription on the same Session handle is not supported; close and reobtain a cached Session first.

To avoid accidentally referring to the `ConnectionFactory` that does not support caching, (`emsConnectionFactory`), you should use an inner object definition as shown below.

```
<object id="connectionFactory" type="Spring.Messaging.Ems.Connections.CachingConnectionFactory, Spring.Messaging.Ems">
  <property name="SessionCacheSize" value="10" />
  <property name="TargetConnectionFactory">
    <object type="Spring.Messaging.Ems.Common.EmsConnectionFactory, Spring.Messaging.Ems">
      <constructor-arg name="serverUrl" value="tcp://localhost:7222"/>
      <constructor-arg name="clientId" value="SpringEMSClient"/>
      <property name="ConnAttemptCount" value="10" />
      <property name="ConnAttemptDelay" value="100" />
      <property name="ConnAttemptTimeout" value="1000" />
    </object>
  </property>
</object>
```

30.3.4. Dynamic Destination Management

The section in the ActiveMQ documentation covers the use of Dynamic Destination management for TIBCO as well.

30.3.5. Accessing Adminstrated objects via JNDI

TIBCO provides an implementation of JNDI to retrieve administrative objects in .NET. You can retrieve TIBCO [Destinations](#) and `ConnectionFactories` from the JNDI registry. To provide ease of access to these JNDI managed objects in a Spring application context the class `JndiFactoryObject` is used. This allows you look configure the location of the JNDI registry and to retrieve objects by name. The objects are retrieved from JNDI at application startup.

These retrieved objects from JNDI in turn can be dependency injected into other collaborating objects such as Spring's `CachingConnectionFactory` (for connections) or `EmsTemplate` (for destinations). Here is an example to retrieve a TIBCO `ConnectionFactory` object from the JNDI registry.

```
<object id="jndiEmsConnectionFactory" type="Spring.Messaging.Ems.Jndi.JndiLookupFactoryObject, Spring.Messaging.Ems">
  <property name="JndiName" value="TopicConnectionFactory"/>
  <property name="JndiProperties[LookupContext.PROVIDER_URL]" value="tibjmsnaming://localhost:7222"/>
</object>
```

`JndiLookupFactory` object implements the `IFactoryObject` interface, so the type that is associated with the name 'jndiConnectionFactory' is not `JndiLookupFactoryObject`, but the type returned from this factory's 'GetType' method, in this case the type of what was retrieved from JNDI.



Note

The dictionary `JndiProperties` is set using Spring Expression language syntax for the property name. This provides a shortcut to the more verbose `<dictionary/>` element. To enable this functionality a the `TIBCO.EMS.LookupContext` was registered under the name 'LookupContext' in Spring's `TypeRegistry`.

The use of this object retrieved from JNDI to configure Spring's `CachingConnectionFactory` set the property `TargetConnectionFactory` as shown below

```
<object id="cachingJndiConnectionFactory" type="Spring.Messaging.Ems.Connections.CachingConnectionFactory, Spring.Messaging.Ems.Connections.CachingConnectionFactory" />
<property name="SessionCacheSize" value="10" />
<property name="TargetConnectionFactory">
  <object type="Spring.Messaging.Ems.Common.EmsConnectionFactory, Spring.Messaging.Ems.Common.EmsConnectionFactory" />
  <constructor-arg ref="jndiEmsConnectionFactory" />
</object>
</property>
</object>
```

Other useful properties and features of `JndiLookupFactoryObject` are

- `JndiContextType` : This is an enumeration that can have either the value `JMS` or `LDAP`. These translate to configuring JNDI context with the constants `LookupContextFactory.TIBJMS_NAMING_CONT` or `LookupContextFactory.LDAP_CONTEXT` for use with EMS's own JNDI registry or an LDAP directory respectively. The default is set use `LookupContextFactory.TIBJMS_NAMING_CONT`. The type `JndiContextType` is also registered in Spring's `TypeRegistry` so that you can use a SpEL expression to set the value as shown below.

```
<object id="jndiEmsConnectionFactory" type="Spring.Messaging.Ems.Jndi.JndiLookupFactoryObject, Spring.Messaging.Ems.Jndi.JndiLookupFactoryObject" />
<property name="JndiName" value="TopicConnectionFactory" />
<property name="JndiProperties[LookupContext.PROVIDER_URL]" value="tibjmsnaming://localhost:7222" />
<property name="JndiContextType" expression="JndiContextType.JMS" />
<property name="ExpectedType" value="TIBCO.EMS.ConnectionFactory" />
</object>
```



Note

The `TargetConnectionFactory` is of the Spring wrapper type `Spring.Messaging.Ems.Common.IConnectionFactory`. You can pass into Spring's implementation of that interface, `Spring.Messaging.Ems.Common.EmsConnectionFactory`, the 'raw' TIBCO EMS type, `TIBCO.EMS.ConnectionFactory`.

- `ExpectedType`: This is a property of the type `System.Type`. You can set the type that the located JNDI object is supposed to be assignable to, if any. It's use is shown in the previous XML configuraiton listing.
- `JndiLookupContext`: This is a property of the type `TIBCO.EMS.ILookupContext`. If you create a custom implementation of `ILookupContext` (for example one that performs lazy caching), assign this property instead of configuring the property `JndiContextType`.
- `DefaultObject`: Sets a reference to an instance of an object to fall back to if the JNDI lookup fails. The default is not to have a fallback object.

30.3.6. MessageListenerContainers

Spring's `MessageListenerContainer`'s are used to process messages asynchronously and concurrently. `MessageListenerContainers` are described more in this section.

30.3.7. Transaction Management

Spring provides an implementation of the `IPlatformTransactionManager` interface for managing TIBCO messaging transactions. The class is `EmsTransactionManager` and it manages transactions for a single `ConnectionFactory`. Please refer to this section for additional information on messaging based transaction managers.

30.3.8. Sending a Message

The class `Spring.Messaging.Ems.Core.EmsTemplate` contains several convenience methods to send a message. These methods are identical to those described in the ActiveMQ documentation section aside from the use of type `TIBCO.EMS.Destination` instead of `Apache.NMS.IDestination` and switching of the namespace from `Apache.NMS` to `Spring.Messaging.Ems.Common`.

Shown below is the code example for a `SimplePublisher` using Spring's TIBCO EMS classes. This does now show the 'one-liner' send methods but one that gives you direct access to the `ISession` to create the message however you wish.

```
using Spring.Messaging.Ems.Common;
using TIBCO.EMS;

namespace Spring.Messaging.Ems.Core
{
    public class SimplePublisher
    {
        private EmsTemplate emsTemplate;

        public SimplePublisher()
        {
            emsTemplate = new EmsTemplate(new EmsConnectionFactory("tcp://localhost:7222"));
        }

        public void Publish(string ticker, double price)
        {
            emsTemplate.SendWithDelegate("APP.STOCK.MARKETDATA",
                delegate(ISession session)
                {
                    MapMessage message = session.CreateMapMessage();
                    message.SetString("TICKER", ticker);
                    message.SetDouble("PRICE", price);
                    message.Priority = 5;
                    return message;
                });
        }
    }
}
```

A more DI friendly implementation would be to expose a `EmsTemplate` property or to inherit from Spring's `EmsGatewaySupport` base class which provides a `ICConnectionFactory` property that will instantiate a `EmsTemplate` instance that is made available via the property `EmsTemplate`.

```
using Spring.Messaging.Ems.Common;
using TIBCO.EMS;

namespace Spring.Messaging.Ems.Core
{
    public class SimpleGateway : EmsGatewaySupport
    {
        public void Publish(string ticker, double price)
        {
            EmsTemplate.SendWithDelegate("APP.STOCK.MARKETDATA",
                delegate(ISession session)
                {
                    MapMessage message = session.CreateMapMessage();
                    message.SetString("TICKER", ticker);
                });
        }
    }
}
```

```

        message.SetDouble("PRICE", price);
        message.Priority = 5;
        return message;
    });
}
}
}

```

Where the `ConnectionFactory` is injected using the configuration.

```

<object id="simpleGateway" type="Spring.Messaging.Ems.Core.SimpleGateway, Spring.Messaging.Ems.Integration.Tests">
  <property name="ConnectionFactory" ref="connectionFactory" />
</object>

```

30.4. Using MessageConverters

In order to facilitate the sending of domain model objects, the `EmsTemplate` has various send methods that take a .NET object as an argument for a message's data content. The overloaded methods `ConvertAndSend` and `ReceiveAndConvert` in `NmsTemplate` delegate the conversion process to an instance of the `IMessageConverter` interface. Please refer to this section for more information on `MessageConverters`.

Example code that uses the `EmsTemplate`'s `ConvertAndSendWithDelegate`, which allows access to the message after it has been converted but before it has been sent is shown below. For examples of using other `ConvertAndSend` methods see the section referred to in the previous paragraph.

```

public void PublishUsingDict(string ticker, double price)
{
    IDictionary marketData = new Hashtable();
    marketData.Add("TICKER", ticker);
    marketData.Add("PRICE", price);
    EmsTemplate.ConvertAndSendWithDelegate("APP.STOCK.MARKETDATA", marketData,
        delegate(Message message)
        {
            message.Priority = 5;
            message.CorrelationID = new Guid().ToString();
            return message;
        });
}

```

30.5. Session and Producer Callback

Please refer to this section for more information on Session and Producer Callbacks.

30.6. Receiving a messages

There are two ways to receive messages, synchronously and asynchronously. To receive messages synchronously use `EmsTemplate`, to receive asynchronously use a `MessageListenerContainer`.

30.6.1. Synchronous Reception

Please refer to this section for using `EmsTemplate`'s overloaded Receive methods.

30.6.2. Asynchronous Reception

Please refer to this section for an introduction to Spring's `MessageListenerContainers`. The TIBCO EMS namespace to create an instance of a message listener container is shown below.

```

using Common.Logging;
using TIBCO.EMS;

```

```
namespace Spring.Messaging.Ems.Core
{
    public class SimpleMessageListener : IMessageListener
    {
        private static readonly ILog LOG = LogManager.GetLogger(typeof(SimpleMessageListener));

        private int messageCount;

        public int MessageCount
        {
            get { return messageCount; }
        }

        public void OnMessage(Message message)
        {
            messageCount++;
            LOG.Debug("Message listener count = " + messageCount);
            TextMessage textMessage = message as TextMessage;
            if (textMessage != null)
            {
                LOG.Info("Message Text = " + textMessage.Text);
            } else
            {
                LOG.Warn("Can not process message of type " + message.GetType());
            }
        }
    }
}
```

And the configuration to create 10 threads that process message off the queue named "APP.STOCK.REQUEST". See this section for more details about the message listener container.

```
<objects xmlns="http://www.springframework.net"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ems="http://www.springframework.net/ems">

    <object id="connectionFactory" type="Spring.Messaging.Ems.Connections.CachingConnectionFactory, Spring.Messaging.Ems">
        <property name="SessionCacheSize" value="10" />
        <property name="TargetConnectionFactory">
            <object type="Spring.Messaging.Ems.Common.EmsConnectionFactory, Spring.Messaging.Ems">
                <constructor-arg name="serverUrl" value="tcp://localhost:7222"/>
                <constructor-arg name="clientId" value="SpringEMSClient"/>
            </object>
        </property>
    </object>

    <object name="simpleMessageListener"
        type="Spring.Messaging.Ems.Core.SimpleMessageListener, Spring.Messaging.Ems.Integration.Tests"/>

    <ems:listener-container connection-factory="connectionFactory" concurrency="10">
        <ems:listener ref="simpleMessageListener" destination="APP.STOCK.REQUEST" />
    </ems:listener-container>

</objects>
```

30.6.3. The `ISessionAwareMessageListener` interface

Refer to this section for more information on the use of this interface.

30.6.4. `MessageListenerAdapter`

Refer to this section for more information on this feature and change code/XML references of 'Nms' to 'Ems'.

30.6.5. Processing messages within a messaging transaction

Refer to this section for more information about this type of message processing.

30.6.6. Messaging Namespace support

To use the EMS namespace you will need to reference the Ems schema. Please refer to this section for more information on configuring message listener containers. Change references of 'Nms' to 'Ems' in that section.

Chapter 31. Message Oriented Middleware - MSMQ

31.1. Introduction

The goals of Spring's MSMQ 3.0 messaging support is to raise the level of abstraction when writing MSMQ applications. The `System.Messaging` API is a low-level API that provides the basis for creating a messaging application. However, 'Out-of-the-box', `System.Messaging` leaves the act of creating sophisticated multi-threaded messaging servers and clients as an infrastructure activity for the developer. Spring fills this gap by providing easy to use helper classes that makes creating an enterprise messaging application easy. These helper classes take into account the nuances of the `System.Messaging` API, such as its lack of thread-safety in many cases, the handling of so-called 'poison messages' (messages that are endlessly redelivered due to an unrecoverable exception during message processing), and combining database transactions with message transactions. Other goals of Spring's MSMQ messaging support are to support messaging best practices, in particular encouraging a clean architectural layering that separates the messaging middleware specifics from the core business processing.

Spring's approach to distributed computing has always been to promote a plain old .NET object approach or a PONO programming model. In this approach plain .NET objects are those that are devoid of any reference to a particular middleware technology. Spring provides the 'adapter' classes that converts between the middleware world, in this case MSMQ, and the oo-world of your business processing. This is done through the use of Spring's `MessageListenerAdapter` class and `IMessageConverters`.

The namespace `Spring.Messaging` provides the core functionality for messaging. It contains the class `MessageQueueTemplate` that simplifies the use of `System.Messaging.MessageQueue` by handling the lack of thread-safety in most of `System.Messaging.MessageQueue`'s methods (for example `Send`). A single instance of `MessageQueueTemplate` can be used throughout your application and Spring will ensure that a different instance of a `MessageQueue` class is used per thread when using `MessageQueueTemplate`'s methods. This per-thread instance of a `System.Messaging.MessageQueue` is also available via its property `MessageQueue`. The `MessageQueueTemplate` class is also aware of the presence of either an 'ambient' `System.Transaction`'s transaction or a local `System.Messaging.MessageQueueTransaction`. As such if you use `MessageQueueTemplate`'s `send` and `receive` methods, unlike with plain use of `System.Messaging.MessageQueue`, you do not need to keep track of this information yourself and call the correct overloaded `System.Messaging.MessageQueue` method for a specific transaction environment. When using a `System.Messaging.MessageQueueTransaction` this would usually require you as a developer to come up with your own mechanism for passing around a `MessageQueueTransaction` to multiple classes and layers in your application. `MessageQueueTemplate` manages this for you, so you don't have to do so yourself. These resource management and transaction features of `MessageQueueTemplate` are quite analogous to the transactional features of Spring's `AdoTemplate` in case you are already familiar with that functionality.

For asynchronous reception Spring provides several multi-threaded message listener containers. You can pick and configure the container that matches your message transactional processing needs and configure poison-message handling policies. The message listener container leverages Spring's support for managing transactions. Both DTC, local messaging transactions, and local database transactions are supported. In particular, you can easily coordinate the commit and rollback of a local `MessageQueueTransaction` and a local database transaction when they are used together.

From a programming perspective, Spring's MSMQ support involves you *configuring* message listener containers and *writing a callback function* for message processing. On the sending side, it involves you learning

how to use `MessageQueueTemplate`. In both cases you will quite likely want to take advantage of using `MessageListenerConverters` so you can better structure the translation from the `System.Messaging.Message` data structure to your business objects. After the initial learning hurdle, you should find that you will be much more productive leveraging Spring's helper classes to write enterprise MSMQ applications than rolling your own infrastructure. Feedback and new feature requests are always welcome.

The `Spring.MsmqQuickstart` application located in the examples directory of the distribution shows this functionality in action.

31.2. A quick tour for the impatient

Here is a quick example of how to use Spring's MSMQ support to create a client that sends a message and a multi-threaded server application that receives the message. (The client code could also be used as-is in a multi-threaded environment but this is not demonstrated).

On the client side you create an instance of the `MessageQueueTemplate` class and configure it to use a `MessageQueue`. This can be done programmatically but it is common to use dependency injection and Spring's XML configuration file to configure your client class as shown below.

```
<object id='questionTxQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\questionTxQueue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
</object>

<object id="messageQueueTemplate" type="Spring.Messaging.Core.MessageQueueTemplate, Spring.Messaging">
  <property name="MessageQueueObjectName" value="questionTxQueue" />
</object>

<!-- Class you write -->
<object id="questionService" type="MyNamespace.QuestionService, MyAssembly">
  <property name="MessageQueueTemplate" ref="messageQueueTemplate" />
</object>
```

The `MessageQueue` object is created via an instance of `MessageQueueFactoryObject` and the `MessageQueueTemplate` refers to this factory object by name and not by reference. The `SimpleSender` class looks like this

```
public class QuestionService : IQuestionService
{
    private MessageQueueTemplate messageQueueTemplate;

    public MessageQueueTemplate {
        get { return messageQueueTemplate; }
        set { messageQueueTemplate = value; }
    }

    public void SendQuestion(string question)
    {
        messageQueueTemplate.ConvertAndSend(question);
    }
}
```

This class can be shared across multiple threads and the `MessageQueueTemplate` will take care of managing thread local access to a `System.Messaging.MessageQueue` as well as any `System.Messaging.IMessageFormatter` instances.

Furthermore, since this is a transactional queue (only the name gives it away), the message will be sent using a single local messaging transaction. The conversion from the string to the underlying message is managed by an instance of the `IMessageConverter` class. By default an implementation that uses an `XmlMessageFormatter` with a `TargetType` of `System.String` is used. You can configure the `MessageQueueTemplate` to use other

`IMessageConveter` implementations that do conversions above and beyond what the 'stock' `IMessageFormatters` do. See the section on `MessageConverters` for more details.

On the receiving side we would like to consume the messages transactionally from the queue. Since no other database operations are being performed in our server side processing, we select the `TransactionalMessageListenerContainer` and configure it to use the `MessageQueueTransactionManager`. The `MessageQueueTransactionManager` an implementation of Spring's `IPlatformTransactionManager` abstraction that provides a uniform API on top of various transaction manager (ADO.NET, NHibernate, MSMQ, etc). Spring's `MessageQueueTransactionManager` is responsible for createing, committing, and rolling back a `MSMQ MessageQueueTransaction`.

While you can create the message listener container programmatically, we will show the declarative configuration approach below

```
<!-- Queue to receive from -->
<object id='questionTxQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\questionTxQueue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
</object>

<!-- MSMQ Transaction Manager -->
<object id="messageQueueTransactionManager" type="Spring.Messaging.Core.MessageQueueTransactionManager, Spring.Messaging"/>

<!-- Message Listener Container that uses MSMQ transactional for receives -->
<object id="transactionalMessageListenerContainer" type="Spring.Messaging.Listener.TransactionalMessageListenerContainer, Spring.Messaging" />
  <property name="MessageQueueObjectName" value="questionTxQueue" />
  <property name="PlatformTransactionManager" ref="messageQueueTransactionManager" />
  <property name="MaxConcurrentListeners" value="10" />
  <property name="MessageListener" ref="messageListenerAdapter" />
</object>

<!-- Adapter to call a PONO as a messaging callback -->
<object id="messageListenerAdapter" type="Spring.Messaging.Listener.MessageListenerAdapter, Spring.Messaging">
  <property name="HandlerObject" ref="questionHandler" />
</object>

<!-- The PONO class that you write -->
<object id="questionHandler" type="MyNamespace.QuestionHandler, MyAssembly" />
```

We have specified the queue to listen, that we want to consume the messages transactionally, process messages from the queue using 10 threads, and that our plain object that will handle the business processing is of the type `QuestionHandler`. The only class you need to write, `QuestionHandler`, looks like

```
public class QuestionHandler : IQuestionHandler
{
    public void HandleObject(string question)
    {
        // perform message processing here

        Console.WriteLine("Received question: " + question);

        // use an instance of MessageQueueTemplate and have other MSQM send operations
        // partake in the same local message transaction used to receive
    }
}
```

That is general idea. You write the sender class using `MessageQueueTemplate` and the consumer class which does not refer to any messaging specific class. The rest is configuration of Spring provided helper classes.

Note that if the `HandleObject` method has returned a string value a reply message would be sent to a response queue. The response queue would be taken from the Message's own `ResponseQueue` property or can be specified explicitly using `MessageListenerAdapter`'s `DefaultResponseQueueName` property.

If an exception is thrown inside the `QuestionHandler`, then the MSMQ transaction is rolled back, putting the message back on the queue for redelivery. If the exception is not due to a transient error in the system, but a logical processing exception, then one would get endless redelivery of the message - clearly not a desirable situation. These messages are so called 'poison messages' and a strategy needs to be developed to deal with them. This is left as a development task if you when using the `System.Messaging` APIs but Spring provides a strategy for handling poison messages, both for DTC based message reception as well as for local messaging transactions.

In the last part this 'quick tour' we will configure the message listener container to handle poison messages. This is done by creating an instance of `SendToQueueExceptionHandler` and setting the property `MaxRetry` to be the number of exceptions or retry attempts we are willing to tolerate before taking corrective actions. In this case, the corrective action is to send the message to another queue. We can then create other message listener containers to read from those queues and handle the messages appropriately or perhaps you will avoid automated processing of these messages and take manual corrective actions.

```
<!-- The 'error' queue to send poison messages -->
<object id='errorQuestionTxQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\errorQuestionTxQueue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
</object>

<!-- Message Listener Container that uses MSMQ transactional for receives -->
<object id="transactionalMessageListenerContainer" type="Spring.Messaging.Listener.TransactionalMessageListenerContainer, Spring.Messaging.Listener.TransactionalMessageListenerContainer">
  <!-- as before but adding -->
  <property name="MessageTransactionExceptionHandler" ref="messageTransactionExceptionHandler" />
</object>

<!-- Poison message handling policy -->
<object id="messageTransactionExceptionHandler" type="Spring.Messaging.Listener.SendToQueueExceptionHandler, Spring.Messaging.Listener.SendToQueueExceptionHandler">
  <property name="MaxRetry" value="5" />
  <property name="MessageQueueObjectName" value="errorQuestionTxQueue" />
</object>
```

In the event of an exception while processing the message, the message transaction will be rolled back (putting the message back on the queue `questionTxQueue` for redelivery). If the same message causes an exception in processing 5 times, then it will be sent transactionally to the `errorQuestionTxQueue` and the message transaction will commit (removing it from the queue `questionTxQueue`). You can also specify that certain exceptions should commit the transaction (remove from the queue) but this is not shown here, see below for more information on this functionality. The `SendToQueueExceptionHandler` implements the interface `IMessageTransactionExceptionHandler` (discussed below) so you can write your own implementations should the provided ones not meet your needs.

That's the quick tour folks. Hopefully you got a general feel for how things work, what requires configuration, and what is the code you need to write. The following sections describe each of Spring's helper classes in more detail. The sample application that ships with Spring is also a good place to get started.

31.3. Using Spring MSMQ

31.3.1. MessageQueueTemplate

The `MessageQueueTemplate` is used for synchronously sending and receiving messages. A single instance can be shared across multiple threads, unlike the standard `System.Messaging.MessageQueue` class. (One less resource management issue to worry about!) A thread-local instance of the `MessageQueue` class is available via `MessageQueueTemplate`'s property `MessageQueue`. A `MessageQueueTemplate` is created by passing a reference to the name of a `MessageQueueFactoryObject`, you can think of it as a friendly name for your `MessagingQueue`.

and the recipe of how to create an instance of it. See the following section on `MessageQueueFactoryObject` for more information.

The `MessageQueueTemplate` also provides several convenience methods for sending and receiving messages. A family of overloaded `ConvertAndSend` and `ReceiveAndConvert` methods allow you to send and receive an object. The default message queue to send and receive from is specified using the `MessageQueueTemplate`'s property `MessageQueueObjectName`. The responsibility of converting the object to a `Message` and vice versa is given to the template's associated `IMessageConverter` implementation. This can be set using the property `MessageConverter`. The default implementation, `XmlMessageConverter`, uses an `XmlMessageFormatter` with its `TargetType` set to `System.String`. Note that `System.Messaging.IMessageFormatter` classes are also not thread safe, so `MessageQueueTemplate` ensures that thread-local instances of `IMessageConverter` are used (as they generally wrap `IMessageFormatter`'s that are not thread-safe).

You can use the `MessageQueueTemplate` to send messages to other `MessageQueues` by specifying their queue 'object name', the name of the `MessageQueueFactoryObject`.

The family of overloaded `ConvertAndSend` and `ReceiveAndConvert` methods are shown below

```
void ConvertAndSend(object obj);

void ConvertAndSend(object obj, MessagePostProcessorDelegate messagePostProcessorDelegate);

void ConvertAndSend(string messageQueueObjectName, object message);

void ConvertAndSend(string messageQueueObjectName, object obj, MessagePostProcessorDelegate messagePostProcessorDelegate);

object ReceiveAndConvert();

object ReceiveAndConvert(string messageQueueObjectName);
```

The transactional settings of the underlying overloaded `System.Messaging.MessageQueue Send` method that are used are based on the following algorithm.

1. If the message queue is transactional and there is an ambient `MessageQueueTransaction` in thread local storage (put there via the use of Spring's `MessageQueueTransactionManager` or `TransactionalMessageListenerContainer`), the message will be sent transactionally using the `MessageQueueTransaction` object in thread local storage.



Note

This lets you group together multiple messaging operations within the same transaction without having to explicitly pass around the `MessageQueueTransaction` object.

2. If the message queue is transactional but there is no ambient `MessageQueueTransaction`, then a single message transaction is created on each messaging operation. (`MessageQueueTransactionType = Single`).
3. If there is an ambient `System.Transactions` transaction then that transaction will be used (`MessageQueueTransactionType = Automatic`).
4. If the queue is not transactional, then a non-transactional send (`MessageQueueTransactionType = None`) is used.

The delegate `MessagePostProcessorDelegate` has the following signature

```
public delegate Message MessagePostProcessorDelegate(Message message);
```

This lets you modify the message after it has been converted from an object to a message using the `IMessageConverter` but before it is sent. This is useful for setting Message properties (e.g. `CorrelationId`, `AppSpecific`, `TimeToReachQueue`). Using anonymous delegates in .NET 2.0 makes this a very succinct coding task. If you have elaborate properties that need to be set, perhaps creating a custom `IMessageConverter` would be appropriate.

Overloaded `Send` and `Receive` operations that use the algorithm listed above to set transactional delivery options are also available. These are listed below

```
Message Receive();

Message Receive(string messageQueueObjectName);

void Send(Message message);

void Send(string messageQueueObjectName, Message message);

void Send(MessageQueue messageQueue, Message message);
```

Note that in the last `Send` method that takes a `MessageQueue` instance, it is the callers responsibility to ensure that this instance is not accessed from multiple threads. This `Send` method is commonly used when getting the `MessageQueue` from the `ResponseQueue` property of a `Message` during an asynchronous receive process. The receive timeout of the `Receive` operations is set using the `ReceiveTimeout` property of `MessageQueueTemplate`. The default value is `MessageQueue.InfiniteTimeout` (which is actually ~3 months).

The XML configuration snippet for defining a `MessageQueueTemplate` is shown in the previous section and also is located in the MSMQ quickstart application configuration file `Messaging.xml`

31.3.2. MessageQueueFactoryObject

The `MessageQueueFactoryObject` is responsible for creating `MessageQueue` instances. You configure the factory with some basic information, namely the constructor parameters you are familiar with already when creating a standard `MessageQueue` instance, and then setting `MessageQueue` properties, such as a `Label` etc. Some configuration tasks of a `MessageQueue` involve calling methods, for example to set which properties of the message to read. These are available as properties to set on the `MessageQueueFactoryObject`. An example declarative configuration is shown below

```
<object id='testqueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <!-- properties passed to the MessageQueue constructor -->
  <property name='Path' value='.\Private$\testqueue' />
  <property name='DenySharedReceive' value='true' />
  <property name='AccessMode' value='Receive' />
  <property name='EnableCache' value='true' />
  <!-- properties that call configuration methods on the MessageQueue -->
  <property name='MessageReadPropertyFilterSetAll' value='true' />
  <property name='ProductTemplate'>
    <object>
      <property name='Label' value='MyLabel' />
      <!-- other MessageQueue properties can be set here -->
    </object>
  </property>
</object>
```

Whenever an object reference is made to 'testqueue' a new instance of the `MessageQueue` class is created. This is Spring's so-called 'prototype' model, which differs from 'singleton' mode. In the singleton creation mode whenever an object reference is made to a 'testqueue' the same `MessageQueue` instance would be used. So that a new instance can be retrieved based on need, the message listener containers take as an argument the name of the `MessageQueueFactoryObject` and not a reference. (i.e. use of 'value' instead of 'ref' in the XML).



Note

The `MessageQueueFactoryObject` class is an ideal candidate for use of a custom namespace. This will be provided in the future. This will allow you to use VS.NET IntelliSense to configure this commonly used object. An example of the potential syntax is shown below

```
<mq:messageQueue id="testqueue" path=".\\Private$\\testqueue" MessageReadPropertyFilterSetAll="true">
  <mq:properties label="MyLabel"/>
</mq:messageQueue>
```

31.3.3. MessageQueue and IMessageConverter resource management

`MessageQueues` and `IMessageFormatters` (commonly used in `IMessageConverter` implementations) are not thread-safe. For example, only the following methods on `MessageQueue` are thread-safe, `BeginPeek`, `BeginReceive`, `EndPeek`, `EndReceive`, `GetAllMessages`, `Peek`, and `Receive`.

To isolate the creation logic of these classes, the factory interface `IMessageQueueFactory` is used. The interface is shown below

```
public interface IMessageQueueFactory
{
    MessageQueue CreateMessageQueue(string messageQueueObjectName);

    IMessageConverter CreateMessageConverter(string messageConverterObjectName);
}
```

A provided implementation, `DefaultMessageQueueFactory` will create an instance of each class per-thread. It delegates the creation of the `MessageQueue` instance to the Spring container. The argument, `messageConverterObjectName`, must be the id/name of a `MessageQueueFactoryObject` defined in the Spring container.

`DefaultMessageQueueFactory` leverages Spring's local thread storage support so it will work correctly in stand alone and web applications.

You can use the `DefaultMessageQueueFactory` independent of the rest of Spring's MSMQ support should you need only the functionality it offers. `MessageQueueTemplate` and the listener containers create an instance of `DefaultMessageQueueFactory` by default. Should you want to share the same instance across these two classes, or provide your own custom implementation, use the property `MessageQueueFactory` on either `MessageQueueTemplate` or the message listener classe.s

31.3.4. Message Listener Containers

One of the most common uses of MSMQ is to concurrently process messages delivered asynchronously. This support is provided in Spring by message listener containers. A message listener container is the intermediary between an `IMessageListener` and a `MessageQueue`. (Note, message listener containers are conceptually different than Spring's Inversion of Control container, though it integrates and leverages the IoC container.) The message listener container takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion and suchlike. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegate boilerplate MSMQ infrastructure concerns to the framework.

A subclass of `AbstractMessageListenerContainer` is used to receive messages from a `MessageQueue`. Which subclass you pick depends on your transaction processing requirements. The following subclasses are available in the namespace `Spring.Messaging.Listener`

- `NonTransactionalMessageListenerContainer` - does not surround the receive operation with a transaction

- `TransactionalMessageListenerContainer` - surrounds the receive operation with local (non-DTC) based transaction(s).
- `DistributedTxMessageListenerContainer` - surrounds the receive operation with a distributed (DTC) transaction

Each of these containers use an implementation in which is based on Peeking for messages on a `MessageQueue`. Peeking is the only resource efficient approach that can be used in order to have `MessageQueue` receipt in conjunction with transactions, either local MSMQ transactions, local ADO.NET based transactions, or DTC transactions. Each container can specify the number of threads that will be created for processing messages after the Peek occurs via the property `MaxConcurrentListeners`. Each processing thread will continue to listen for messages up until the timeout value specified by `ListenerTimeLimit` or until there are no more messages on the queue (whichever comes first). The default value of `ListenerTimeLimit` is `TimeSpan.Zero`, meaning that only one attempt to receive a message from the queue will be performed by each listener thread. The current implementation uses the standard .NET thread pool. Future implementations will use a custom (and pluggable) thread pool.

31.3.4.1. NonTransactionalMessageListenerContainer

This container performs a Receive operation on the `MessageQueue` without any transactional settings. As such messages will not be redelivered if an exception is thrown during message processing. Exceptions during message processing can be handled via an implementation of the interface `ExceptionHandler`. This can be set via the property `ExceptionHandler` on the listener. The `ExceptionHandler` interface is shown below

```
public interface IExceptionHandler
{
    void OnException(Exception exception, Message message);
}
```

An example of configuring a `NonTransactionalMessageListenerContainer` with an `ExceptionHandler` is shown below

```
<!-- Queue to receive from -->
<object id='msmqTestQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\testqueue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
  <property name='ProductTemplate'>
    <object>
      <property name='Label' value='MyTestQueueLabel' />
    </object>
  </property>
</object>

<!-- Queue to respond to -->
<object id='msmqTestResponseQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\testresponsequeue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
  <property name='ProductTemplate'>
    <object>
      <property name='Label' value='MyTestResponseQueueLabel' />
    </object>
  </property>
</object>

<!-- Listener container -->
<object id='nonTransactionalMessageListenerContainer' type='Spring.Messaging.Listener.NonTransactionalMessageListenerContainer'>
  <property name='MessageQueueObjectName' value='msmqTestQueue' />
  <property name='MaxConcurrentListeners' value='2' />
  <property name='ListenerTimeLimit' value='20s' /> <!-- 20 seconds -->
  <property name='MessageListener' ref='messageListenerAdapter' />
  <property name='ExceptionHandler' ref='exceptionHandler' />
</object>
```



```
<!-- Delegate to plain .NET object for message handling -->
<object id="messageListenerAdapter" type="Spring.Messaging.Listener.MessageListenerAdapter, Spring.Messaging">
  <property name="DefaultResponseQueueName" value="msmqTestResponseQueue"/>
  <property name="HandlerObject" ref="simpleHandler"/>
</object>

<!-- Classes you need to write -->
<object id="simpleHandler" type="MyNamespace.SimpleHandler, MyAssembly"/>

<object id="exceptionHandler" type="MyNamespace.SimpleExceptionHandler, MyAssembly"/>
```

The SimpleHandler class would look something like this

```
public class SimpleHandler : ISimpleHandler
{
    public void HandleObject(string txt)
    {
        // perform message processing...
        Console.WriteLine("Received text: " + txt);
    }
}
```

31.3.4.2. TransactionalMessageListenerContainer

This message listener container performs receive operations within the context of local transaction. This class requires an instance of Spring's `IPlatformTransactionManager`, either `AdoPlatformTransactionManager`, `HibernateTransactionManager`, or `MessageQueueTransactionManager`.

If you specify a `MessageQueueTransactionManager` then a `MessageQueueTransaction` will be started before receiving the message and used as part of the container's receive operation. As with other `IPlatformTransactionManager` implementation's, the transactional resources (in this case an instance of the `MessageQueueTransaction` class) is bound to thread local storage. `MessageQueueTemplate` will look in thread-local storage and use this 'ambient' transaction if found for its send and receive operations. The message listener is invoked and if no exception occurs, then the `MessageQueueTransactionManager` will commit the `MessageQueueTransaction`.

The message listener implementation can call into service layer classes that are made transactional using standard Spring declarative transactional techniques. In case of exceptions in the service layer, the database operation will be rolled back (nothing new here), and the `TransactionalMessageListenerContainer` will call its `IMessageTransactionExceptionHandler` implementation to determine if the `MessageQueueTransaction` should commit (removing the message from the queue) or rollback (leaving the message on the queue for redelivery).



Note

The use of a transactional service layer in combination with a `MessageQueueTransactionManager` is a powerful combination that can be used to achieve "exactly one" transaction message processing with database operations. This requires a little extra programming effort and is a more efficient alternative than using distributed transactions which are commonly associated with this functionality since both the database and the message transaction commit or rollback together.

The additional programming logic needed to achieve this is to keep track of the `Message.Id` that has been processed successfully within the transactional service layer. This is needed as there may be a system failure (e.g. power goes off) between the 'inner' database commit and the 'outer' messaging commit, resulting in message redelivery. The transactional service layer needs logic to detect if incoming message was processed successfully. It can do this by checking the database for an indication of successful processing, perhaps by recording the `Message.Id` itself in a status table. If the transactional service layer determines that the message has already been processed, it can throw a specific exception for this case. The container's exception handler will recognize this exception type

and vote to commit (remove from the queue) the 'outer' messaging transaction. Spring provides an exception handler with this functionality, see `SendToQueueExceptionHandler` described below.

An example of configuring the `TransactionalMessageListenerContainer` using a `MessageQueueTransactionManager` is shown below

```
<!-- Queue to receive from -->
<object id='msmqTestQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\testqueue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
  <property name='ProductTemplate'>
    <object>
      <property name='Label' value='MyTestQueueLabel' />
    </object>
  </property>
</object>

<!-- Queue to respond to -->
<object id='msmqTestResponseQueue' type='Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging'>
  <property name='Path' value='.\Private$\testresponsequeue' />
  <property name='MessageReadPropertyFilterSetAll' value='true' />
  <property name='ProductTemplate'>
    <object>
      <property name='Label' value='MyTestResponseQueueLabel' />
    </object>
  </property>
</object>

<!-- Transaction Manager for MSMQ Messaging -->
<object id="messageQueueTransactionManager" type="Spring.Messaging.Core.MessageQueueTransactionManager, Spring.Messaging" />

<!-- The transaction message listener container -->
<object id="transactionalMessageListenerContainer" type="Spring.Messaging.Listener.TransactionalMessageListenerContainer, Spring.Messaging" />
  <property name="MessageQueueObjectName" value="msmqTestQueue" />
  <property name="PlatformTransactionManager" ref="messageQueueTransactionManager" />
  <property name="MaxConcurrentListeners" value="5" />
  <property name="ListenerTimeLimit" value="20s" />
  <property name="MessageListener" ref="messageListenerAdapter" />
  <property name="MessageTransactionExceptionHandler" ref="messageTransactionExceptionHandler" />
</object>

<!-- Delegate to plain .NET object for message handling -->
<object id="messageListenerAdapter" type="Spring.Messaging.Listener.MessageListenerAdapter, Spring.Messaging" />
  <property name="DefaultResponseQueueName" value="msmqTestResponseQueue" />
  <property name="HandlerObject" ref="simpleHandler" />
</object>

<!-- Poison message handling -->
<object id="messageTransactionExceptionHandler" type="Spring.Messaging.Listener.SendToQueueExceptionHandler, Spring.Messaging" />
  <property name="MaxRetry" value="5" />
  <property name="MessageQueueObjectName" value="testTxErrorQueue" />
</object>

<!-- Classes you need to write -->
<object id="simpleHandler" type="MyNamespace.SimpleHandler, MyAssembly" />
```

If you specify either `AdoPlatformTransactionManager` Or `HibernateTransactionManager` then a local database transaction will be started before the receiving the message. By default, the container will also start a local `MessageQueueTransaction` after the local database transaction has started, but before the receiving the message. This `MessageQueueTransaction` will be used to receive the message. By default the `MessageQueueTransaction` will be bound to thread local storage so that any `MessageQueueTemplate` send or receive operations will participate transparently in the same `MessageQueueTransaction`. If you do not want this behavior set the property `ExposeContainerManagedMessageQueueTransaction` to false.

In case of exceptions during `IMessageListener` processing when using either either `AdoPlatformTransactionManager` Or `HibernateTransactionManager` the container's

`IMessageTransactionExceptionHandler` will determine if the `MessageQueueTransaction` should commit (removing it from the queue) or rollback (placing it back on the queue for redelivery). The listener exception will always trigger a rollback in the 'outer' database transaction.

Poison message handling, that is, the endless redelivery of a message due to exceptions during processing, can be detected using implementations of the `IMessageTransactionExceptionHandler`. This interface is shown below

```
public interface IMessageTransactionExceptionHandler
{
    TransactionAction OnException(Exception exception, Message message, MessageQueueTransaction messageQueueTransaction);
}
```

The return value is an enumeration with the values `Commit` and `Rollback`. A specific implementation is provided that will move the poison message to another queue after a maximum number of redelivery attempts. See `SendToQueueExceptionHandler` described below. You can set a specific implementation to by setting `TransactionalMessageListenerContainer`'s property `MessageTransactionExceptionHandler`

The `IMessageTransactionExceptionHandler` implementation `SendToQueueExceptionHandler` keeps track of the `Message`'s `Id` property in memory with a count of how many times an exception has occurred. If that count is greater than the handler's `MaxRetry` count it will be sent to another queue using the provided `MessageQueueTransaction`. The queue to send the message to is specified via the property `MessageQueueObjectName`.

31.3.4.3. DistributedTxMessageListenerContainer

This message listener container performs receive operations within the context of distributed transaction. A distributed transaction is started before a message is received. The receive operation participates in this transaction using by specifying `MessageQueueTransactionType = Automatic`. The transaction that is started is automatically promoted to two-phase-commit to avoid the default behavior of transaction promotion since the only reason to use this container is to use two different resource managers (messaging and database typically).

The commit and rollback semantics are simple, if the message listener does not throw an exception the transaction is committed, otherwise it is rolled back.

Exceptions in message listener processing are handled by implementations of the `IDistributedTransactionExceptionHandler` interface. This interface is shown below

```
public interface IDistributedTransactionExceptionHandler
{
    bool IsPoisonMessage(Message message);

    void HandlePoisonMessage(Message poisonMessage);

    void OnException(Exception exception, Message message);
}
```

the `IsPoisonMessage` method determines whether the incoming message is a poison message. This method is called before the `IMessageListener` is invoked. The container will call `HandlePoisonMessage` if `IsPoisonMessage` returns true and will then commit the distributed transaction (removing the message from the queue). Typical implementations of `HandlePoisonMessage` will move the poison message to another queue (under the same distributed transaction used to receive the message). The class `SendToQueueDistributedTransactionExceptionHandler` detects poison messages by tracking the `Message Id` property in memory with a count of how many times an exception has occurred. If that count is greater than the handler's `MaxRetry` count it will be sent to another queue. The queue to send the message to is specified via the property `MessageQueueObjectName`.

31.4. MessageConverters

31.4.1. Using MessageConverters

In order to facilitate the sending of business model objects, the `MessageQueueTemplate` has various send methods that take a .NET object as an argument for a message's data content. The overloaded methods `ConvertAndSend` and `ReceiveAndConvert` in `MessageQueue` delegate the conversion process to an instance of the `IMessageConverter` interface. This interface defines a simple contract to convert between .NET objects and JMS messages. The interface is shown below

```
public interface IMessageConverter : ICloneable
{
    Message ToMessage(object obj);

    object FromMessage(Message message);
}
```

There are a standard implementations provided the simply wrap existing `IMessageFormatter` implementations.

- `XmlMessageConverter` - uses a `XmlMessageFormatter`.
- `BinaryMessageConverter` - uses a `BinaryMessageFormatter`
- `ActiveXMessageConverter` - uses a `ActiveXMessageFormatter`

The default implementation used in `MessageQueueTemplate` and the message listener containers is an instance of `XmlMessageConverter` configured with a `TargetType` to be `System.String`. You specify the types that the `XmlMessageConverter` can convert though either the array property `TargetTypes` or `TargetTypeNames`. Here is an example taken from the QuickStart application

```
<object id="xmlMessageConverter" singleton="false" type="Spring.Messaging.Support.Converters.XmlMessageConverter, Spring.M
  <property name="TargetTypes">
    <list>
      <value>Spring.MsmqQuickStart.Common.Data.TradeRequest, Spring.MsmqQuickStart.Common</value>
      <value>Spring.MsmqQuickStart.Common.Data.TradeResponse, Spring.MsmqQuickStart.Common</value>
      <value>System.String, mscorlib</value>
    </list>
  </property>
</object>
```

You can specify other `IMessageConverter` implementations using the `MessageConverterObjectName` property on the `MessageQueueTemplate` and `MessageListenerAdapter`.



Note

The scope of the object definition is set to `singleton="false"`, meaning that a new instance of the `MessageConverter` will be created each time you ask the container for an object of the name 'xmlMessageConverter'. This is important to ensure that a new instance will be used for each thread. If you forget, a warning will be logged and `IMessageConverter`'s `Clone()` method will be called to create an independent instance.

Other implementations provided are

- `XmlDocumentConverter` - loads and saves an `XmlDocument` to the message `BodyStream`. This lets you manipulate directly the XML data independent of type serialization issues. This is quite useful if you use XPath expressions to pick out the relevant information to construct your business objects.

Other potential implementations:

- `RawBytesMessageConverter` - directly write raw bytes to the message stream, compress
- `CompressedMessageConverter` - compresses the message payload
- `EncryptedMessageConverter` - encrypt the message (standard MSMQ encryption has several limitations)
- `SoapMessageConverter` - use soap formatting.

31.5. Interface based message processing

31.5.1.1. MessageListenerAdapter

The `MessageListenerAdapter` allows methods of a class that does not implement the `IMessageListener` interface to be invoked upon message delivery. Lets call this class the 'message handler' class. To achieve this goal the `MessageListenerAdapter` implements the standard `IMessageListener` interface to receive a message and then delegates the processing to the message handler class. Since the message handler class does not contain methods that refer to MSMQ artifacts such as `Message`, the `MessageListenerAdapter` uses a `IMessageConverter` to bridge the MSMQ and 'plain object' worlds. As a reminder, the default `XmlMessageConverter` used in `MessageQueueTemplate` and the message listener containers converts from `Message` to string. Once the incoming message is converted to an object (string for example) a method with the name 'HandleMessage' is invoked via reflection passing in the string as an argument.

Using the default configuration of `XmlMessageConverter` in the message listeners, a simple string based message handler would look like this.

```
public class MyHandler
{
    public void HandleMessage(string text)
    {
        ...
    }
}
```

The next example has a similar method signature but the name of the handler method name has been changed to "DoWork", by setting the adapter's property `DefaultHandlerMethod`.

```
public interface IMyHandler
{
    void DoWork(string text);
}
```

If your `IMessageConverter` implementation will return multiple object types, overloading the handler method is perfectly acceptable, the most specific matching method will be used. A method with an object signature would be consider a 'catch-all' method of last resort.

```
public interface IMyHandler
{
    void DoWork(string text);
    void DoWork(OrderRequest orderRequest);
    void DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

Another of the capabilities of the `MessageListenerAdapter` class is the ability to automatically send back a response `Message` if a handler method returns a non-void value. Any non-null value that is returned from the execution of the handler method will (in the default configuration) be converted to a string. The resulting string

will then be sent to the `ResponseQueue` defined in the `Message`'s `ResponseQueue` property of the original `Message`, or the `DefaultResponseQueueName` on the `MessageListenerAdapter` (if one has been configured) will be used. If not `ResponseQueue` is found then an `Spring MessagingException` will be thrown. Please note that this exception will not be swallowed and will propagate up the call stack.

Here is an example of Handler signatures that have various return types.

```
public interface IMyHandler
{
    string DoWork(string text);
    OrderResponse DoWork(OrderRequest orderRequest);
    InvoiceResponse DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

The following configuration shows how to hook up the adapter to process incoming MSMQ messages using the default message converter.

```
<!-- Delegate to plain .NET object for message handling -->
<object id="messageListenerAdapter" type="Spring.Messaging.Listener.MessageListenerAdapter, Spring.Messaging">
    <property name="DefaultResponseQueueName" value="msmqTestResponseQueue" />
    <property name="HandlerObject" ref="myHandler" />
</object>
```

31.6. Comparison with using WCF

The goals of Spring's MSMQ messaging support are quite similar to those of WCF with its MSMQ related bindings, in as much as a WCF service contract is a PONO (minus the attributes if you really picky about what you call a PONO). Spring's messaging support can give you the programming convenience of dealing with PONO contracts for message receiving but does not (at the moment) provide a similar PONO contract for sending, instead relying on explicit use of the `MessageQueueTemplate` class. This feature exists - some question whether it should for messaging - in the Java version of the Spring framework, see `JmsInvokerServiceExporter` and `JmsInvokerProxyFactoryBean`.

The good news is that if and when it comes time to move from a Spring MSMQ solution to WCF, you will be in a great position as the PONO interface used for business processing when receiving in a Spring based MSMQ application can easily be adapted to a WCF environment. There may also be some features unique to MSMQ and/or Spring's MSMQ support that you may find appealing over WCF. Many messaging applications still need to be 'closer to the metal' and this is not possible using the WCF bindings, for example Peeking and Label, AppSpecific properties, multicast.. An interesting recent quote by Yoel Arnon (MSMQ guru) *"With all the respect to WCF, System.Messaging is still the major programming model for MSMQ programmers, and is probably going to remain significant for the foreseeable future. The message-oriented programming model is different from the service-oriented model of WCF, and many real-world solutions would always prefer it."*

Chapter 32. Scheduling and Thread Pooling

32.1. Introduction

The Spring Framework features integration classes for scheduling support. Currently, Spring supports the Quartz Scheduler (<http://quartznet.sourceforge.net/>). The scheduler is set up using a `IFactoryObject` with optional references to `Trigger` instances, respectively. Furthermore, a convenience class for the Quartz Scheduler is available that allows you to invoke a method of an existing target object.



Note

There is a Quartz Quickstart application that is shipped with Spring.NET. It is documented [here](#).

32.2. Using the Quartz.NET Scheduler

Quartz uses `Trigger`, `Job` and `JobDetail` objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at <http://quartznet.sourceforge.net/>. For convenience purposes, Spring offers a couple of classes that simplify the usage of Quartz within Spring-based applications.

32.2.1. Using the JobDetailObject

`JobDetail` objects contain all information needed to run a job. The Spring Framework provides a `JobDetailObject` that makes the `JobDetail` easier to configure and with sensible defaults. Let's have a look at an example:

```
<object name="ExampleJob" type="Spring.Scheduling.Quartz.JobDetailObject, Spring.Scheduling.Quartz">
  <property name="JobType" value="Example.Quartz.ExampleJob, Example.Quartz" />
  <property name="JobDataAsMap">
    <dictionary>
      <entry key="Timeout" value="5" />
    </dictionary>
  </property>
</object>
```

The job detail object has all information it needs to run the job (`ExampleJob`). The timeout is specified in the job data dictionary. The job data dictionary is available through the `JobExecutionContext` (passed to you at execution time), but the `JobDetailObject` also maps the properties from the job data map to properties of the actual job. So in this case, if the `ExampleJob` contains a property named `Timeout`, the `JobDetailObject` will automatically apply it:

```
namespace Example.Quartz;

public class ExampleJob : QuartzJobObject {

    private int timeout;

    /// <summary>
    /// Setter called after the ExampleJob is instantiated
    /// with the value from the JobDetailObject (5)
    /// </summary>
    public int Timeout {
        set { timeout = value; };
    }
}
```



```
protected override void ExecuteInternal(JobExecutionContext context) {
    // do the actual work
}
}
```

All additional settings from the job detail object are of course available to you as well.

Note: Using the `name` and `group` properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the object name of the job detail object (in the example above, this is `ExampleJob`).

32.2.2. Using the `MethodInvokingJobDetailFactoryObject`

Often you just need to invoke a method on a specific object. Using the `MethodInvokingJobDetailFactoryObject` you can do exactly this:

```
<object id="JobDetail" type="Spring.Scheduling.Quartz.MethodInvokingJobDetailFactoryObject, Spring.Scheduling.Quartz">
  <property name="TargetObject" ref="ExampleBusinessObject" />
  <property name="TargetMethod" value="DoIt" />
</object>
```

The above example will result in the `doIt` method being called on the `exampleBusinessObject` method (see below):

```
public class ExampleBusinessObject {

    // properties and collaborators

    public void DoIt() {
        // do the actual work
    }
}
```

```
<object id="ExampleBusinessObject" type="Examples.BusinessObjects.ExampleBusinessObject, Examples.BusinessObjects"/>
```

Using the `MethodInvokingJobDetailFactoryObject`, you don't need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same `JobDetail`, it might be possible that before the first job has finished, the second one will start. If `JobDetail` classes implement the `Stateful` interface, this won't happen. The second job will not start before the first one has finished. To make jobs resulting from the `MethodInvokingJobDetailFactoryObject` non-concurrent, set the `concurrent` flag to `false`.

```
<object id="JobDetail" type="Spring.Scheduling.Quartz.MethodInvokingJobDetailFactoryObject, Spring.Scheduling.Quartz">
  <property name="TargetObject" ref="ExampleBusinessObject" />
  <property name="TargetMethod" value="DoIt" />
  <property name="Concurrent" value="false" />
</object>
```



Note

By default, jobs will run in a concurrent fashion.

Also note that when using `MethodInvokingJobDetailFactoryObject` you can't use database persistence for Jobs. See the class documentation for additional details.

32.2.3. Wiring up jobs using triggers and the `SchedulerFactoryObject`

We've created job details and jobs. We've also reviewed the convenience class that allows you to invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using triggers and a `SchedulerFactoryObject`. Several triggers are available within Quartz. Spring offers two subclassed triggers with convenient defaults: `CronTriggerObject` and `SimpleTriggerObject`

Triggers need to be scheduled. Spring offers a `SchedulerFactoryObject` that exposes triggers to be set as properties. `SchedulerFactoryObject` schedules the actual jobs with those triggers.

Find below a couple of examples:

```
<object id="SimpleTrigger" type="Spring.Scheduling.Quartz.SimpleTriggerObject, Spring.Scheduling.Quartz">
  <!-- see the example of method invoking job above -->
  <property name="JobDetail" ref="ExampleJob" />

  <!-- 10 seconds -->
  <property name="StartDelay" value="10s" />

  <!-- repeat every 50 seconds -->
  <property name="RepeatInterval" value="50s" />
</object>

<object id="CronTrigger" type="Spring.Scheduling.Quartz.CronTriggerObject, Spring.Scheduling.Quartz">
  <property name="JobDetail" ref="ExampleJob" />

  <!-- run every morning at 6 AM -->
  <property name="CronExpressionString" value="0 0 6 * * ?" />
</object>
```

Now we've set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the `SchedulerFactoryObject`:

```
<object id="quartzSchedulerFactory" type="Spring.Scheduling.Quartz.SchedulerFactoryObject, Spring.Scheduling.Quartz">
  <property name="triggers">
    <list>
      <ref object="CronTrigger" />
      <ref object="SimpleTrigger" />
    </list>
  </property>
</object>
```

More properties are available for the `SchedulerFactoryObject` for you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the [SchedulerFactoryObject SDK docs](#) for more information.

Chapter 33. Template Engine Support

33.1. Introduction

The Spring Framework features integration classes for templating engine support. Spring 1.3 provides support for the [NVelocity](#) templating engine.

33.2. Dependencies

The Spring NVelocity support depends on the Castle project's NVelocity implementation which is located in the lib directory of the Spring release.

33.3. Configuring a VelocityEngine

The NVelocity template engine is set up using a `IFactoryObject` with optional configuration parameters to define where templates reside, define logging and more. For more information on `IFactoryObjects` see Section 5.9.3, “Customizing instantiation logic using `IFactoryObjects`”. A custom namespace parser is provided to simplify the configuration of a NVelocity template engine. For more information on custom namespace parser see Section 5.11.1, “Registering custom parsers”.

33.3.1. Simple file based template engine definition

You create a simple definition of the template engine that uses the default resource loader as follows:

```
<objects xmlns="http://www.springframework.net" xmlns:nv="http://www.springframework.net/nvelocity">

    <!-- Simple no arg file based configuration use's NVeclocity default file resource loader -->
    <nv:engine id="velocityEngine" />

</objects>
```

The velocity engine could then be used to load and merge a local template using a simple relative path (the default resource loader path is the current execution directory):

```
StringWriter stringWriter = new StringWriter();
Hashtable modelTable = new Hashtable();
modelTable.Add("var1", TEST_VALUE);
VelocityContext velocityContext = new VelocityContext(modelTable);
velocityEngine.MergeTemplate("Template/Velocity/MyTemplate.vm", Encoding.UTF8.WebName, velocityContext, stringWriter);
string mergedContent = stringWriter.ToString();
```

To disable the use of NVelocity's file loader that tracks runtime changes, set the element `prefer-file-system-access` of `<engine/>` to `false`.

33.3.2. Configuration Options

You can define several attributes on the `<engine>` element to control how the factory is configured:

Table 33.1. Engine Factory Configuration Options

Attribute	Description	Required	Default Value
config-file	A uri of a properties file defining the NVelocity configuration. This value accepts all spring resource	no	N/A

Attribute	Description	Required	Default Value
	loader uri (e.g., file://, http://). See Section 33.3.7, “Using a custom configuration file”		
prefer-file-system-access	Instructs the NVelocity engine factory to attempt use NVelocity's file loader. When set to false the provided <code>SpringResourceLoader</code> will be used (and the <code>ResourceLoaderPath</code> property must be set)	no	true
override-logging	Instructs the NVelocity engine factory to use the provided spring commons logging based logging system. See Section 33.3.8, “Logging”	no	true

33.3.3. Assembly based template loading

When templates are packaged in an assembly, NVelocity's assembly resource loader can be used to define where templates reside:

```
<nv:engine id="velocityEngine" >
  <nv:resource-loader>
    <nv:assembly name="MyAssembly" />
  </nv:resource-loader>
</nv:nvelocity>
```

Using the example above the template would be loaded using a namespace syntax for the template resource:

```
velocityEngine.MergeTemplate("MyAssembly.MyNamespace.MyTemplate.vm", Encoding.UTF8.WebName, velocityContext, stringWriter);
```

33.3.4. Using Spring's `IResourceLoader` to load templates

In some cases Spring's `IResource` abstraction can be beneficial to load templates from a variety of resources. A Spring `IResource` loader extension to the NVelocity resource loader implementation is provided for this use case. The following object definition loads the NVelocity templates from a single path

```
<nv:engine id="velocityEngine">
  <nv:resource-loader>
    <nv:spring uri="file://Template/Velocity/" />
  </nv:resource-loader>
</nv:engine>
```

Or with multiple locations

```
<nv:engine id="velocityEngine">
  <nv:resource-loader>
    <nv:spring uri="file://Template/Velocity/" />
    <nv:spring uri="assembly://MyAssembly/MyNameSpace" />
  </nv:resource-loader>
</nv:engine>
```



Note

By default spring will attempt to load resources using NVelocity's file based template loading (useful for detection of template changes at runtime). If this is not desirable you set the `prefer-file-system-access` property of the factory object to `false` which will cause the factory to utilize the supplied spring resource loader.

Using the example above when resource loader paths are defined templates can be loaded using their name:

```
string mergedTemplate = VelocityEngineUtils.MergeTemplateIntostring(velocityEngine, "MyFileTemplate.vm", Encoding.UTF8.WebName
string mergedTemplate = VelocityEngineUtils.MergeTemplateIntostring(velocityEngine, "MyAsemblyTemplate.vm", Encoding.UTF8.W
```

33.3.5. Defining a custom resource loader

The following defines a custom resource loader (the type is an extension of NVelocity's `ResourceLoader` class):

```
<nv:engine id="velocityEngine">
  <nv:resource-loader>
    <nv:custom name="myResourceLoader"
      description="A custom resource loader"
      type="MyNamespace.MyResourceLoader, MyAssembly"
      path="Template/Velocity/" />
  </nv:resource-loader>
</nv:engine>
```

33.3.6. Resource Loader configuration options

The `<nv:resource-loader>` element has additional attributes which define how NVelocity's resource manager and resource loader behave.

Table 33.2. Resource Loader Configuration Options

Attribute	Description	Required	Default Value
default-cache-size	defines resource manager global cache size, applies when caching is turned on. This maps to NVelocity's resource manager <code>resource.manager.defaultcache.size</code> property	no	89
template-caching	Enables template caching for the defined resource loader. This maps to NVelocity's resource loader <code><name>.resource.loader.cache</code> property	no	false
modification-check-interval	The modification check interval value (seconds) of the resource loader, applies only to resource loader with change detection capabilities (file or custom). This maps to NVelocity's resource loader <code><name>.resource.loader.modificationCheckInterval</code> property	no	2

33.3.7. Using a custom configuration file

If so desired one could provide a custom configuration resource to customize the NVelocity configuration:

```
<nv:engine id="velocityEngine" config-file="file://Template/Velocity/config.properties"/>
```

You can override specific properties by providing the `velocityProperties` property to the NVelocity factory object (shown above)

```
<nv:engine id="velocityTemplate" >
  <nv:nvelocity-properties>
    <entry key="input.encoding" value="ISO-8859-1"/>
    <entry key="output.encoding" value="ISO-8859-1"/>
  </nv:nvelocity-properties>
</nv:engine>
```

33.3.8. Logging

By default Spring will override NVelocity's default `ILogSystem` implementation with its own `CommonsLoggingLogSystem` implementation so that the logging stream of NVelocity will go to the same logging subsystem that Spring uses. If this is not desirable, you can specify the following property of the NVelocity factory object:

```
<template:nvelocity id="velocityEngine" override-logging="false" />
```

33.4. Merging a template

Spring provides the `VelocityEngineUtils` utility for merging templates using an engine instance:

```
string mergedTemplate = VelocityEngineUtils.MergeTemplateIntostring(velocityEngine, "MyTemplate.vm", Encoding.UTF8.WebName, ...)
```

33.5. Configuring a VelocityEngine without a custom namespace

While most users will prefer to use the NVelocity custom namespace to configure a `VelocityEngine`, you can also use standard `<object/>` definition syntax as shown below:

To create a `VelocityEngine` using the default file resource loader use the definition:

```
<!-- Simple no arg file based configuration use's NVelocity default file resource loader -->
<object id="velocityEngine" type="Spring.Template.Velocity.VelocityEngineFactoryObject, Spring.Template.Velocity" />
```

For convenience in defining NVelocity engine instances a custom namespace is provided, for example the resource loader definition could be done this way:

```
<objects xmlns="http://www.springframework.net" xmlns:nv="http://www.springframework.net/nvelocity">

  <nv:nvelocity id="velocityEngine" >
    <nv:resource-loader>
      <nv:file path="Template/Velocity/" />
    </nv:resource-loader>
  </nv:nvelocity>

</objects>
```

When templates are packaged in an assembly, NVelocity's assembly resource loader can be used to define where templates reside:

```
<!-- Assembly based template loading with NVelocity assembly resource loader -->
<object id="velocityEngine" type="Spring.Template.Velocity.VelocityEngineFactoryObject, Spring.Template.Velocity">
  <property name="VelocityProperties">
    <dictionary key-type="string" value-type="object">
      <entry key="resource.loader" value="assembly"/>
      <entry key="assembly.resource.loader.class" value="NVelocity.Runtime.Resource.Loader.AssemblyResourceLoader"/>
      <entry key="assembly.resource.loader.assembly" value="MyAssembly"/>
    </dictionary>
  </property>
</object>
```

To load NVelocity templates from a single path use the definition:

```
<object id="velocityEngine" type="Spring.Template.Velocity.VelocityEngineFactoryObject, Spring.Template.Velocity" >
  <property name="ResourceLoaderPath" value="file://MyTemplateFolder/AnotherFolder/" />
</object>
```

To load NVelocity templates from multiple paths use the definition:

```
<object id="velocityEngine" type="Spring.Template.Velocity.VelocityEngineFactoryObject, Spring.Template.Velocity" >
  <property name="ResourceLoaderPaths" >
    <list>
      <value>file://MyTemplateFolder/</value>
      <value>file://MyOtherTemplateFolder/</value>
    </list>
  </property>
</object>
```



Note

By default spring will attempt to load resources using NVelocity's file based template loading (useful for detection of template changes at runtime). If this is not desirable you set the `preferFileSystemAccess` property of the factory object to `false` which will cause the factory to utilize the supplied spring resource loader.

To refer to a property file based configuration of the TemplateEngine use the definition:

```
<object id="velocityEngine" type="Spring.Template.Velocity.VelocityEngineFactoryObject, Spring.Template.Velocity" >
  <property name="ConfigLocation" value="file://Template/Velocity/config.properties" />
</object>
```



Note

You can override specific properties by providing the `velocityProperties` property.

To not integrate with the Common.Logging subsystem, set the `OverrideLogging` property to false:

```
<object id="velocityEngine" type="Spring.Template.Velocity.VelocityEngineFactoryObject, Spring.Template.Velocity" >
  <property name="OverrideLogging" value="false" />
</object>
```

Part VI. VS.NET Integration

This part of the reference documentation covers the Spring Framework's integration with VS.NET

- Chapter 34, *Visual Studio.NET Integration*

Chapter 34. Visual Studio.NET Integration

34.1. XML Editing and Validation

Most of this section is well travelled territory for those familiar with editing XML files in their favorite XML editor. The XML configuration data that defines the objects that Spring will manage for you are validated against the Spring.NET XML Schema at runtime. The location of the XML configuration data to create an `IApplicationContext` can be any of the resource locations supported by Spring's `IResource` abstraction. (See Section 7.1, "Introduction" for more information.) To create an `IApplicationContext` using a "standalone" XML configuration file the custom configuration section in the standard .NET application configuration would read:

```
<spring>

  <context>
    <resource uri="file://objects.xml"/>
  </context>

</spring>
```

The VS.NET 2005 or later, the XML editor uses the attribute `xsi:schemaLocation` as a hint to associate the physical location of a schema file with the XML document being edited. VS.NET 2002/2003 do not recognize the `xsi:schemaLocation` element. If you reference the Spring.NET XML schema as shown below, you can get intellisense and validation support while editing a Spring configuration file in VS.NET 2005/2008. In order to get this functionality in VS.NET 2002/2003 you will need to register the schema with VS.NET or include the schema as part of your application project.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects.xsd">
  <object id="..." type="...">
    ...
  </object>
  <object id="..." type="...">
    ...
  </object>
  ...
</objects>
```

It is typically more convenient to install the schema in VS.NET, even for VS.NET 2005/2008, as it makes the xml a little less verbose and you don't need to keep copying the XSD file for each project you create. For VS.NET 2003 the schema directory is

Table 34.1.

Visual Studio Version	Directory to put Spring .XSD files
VS.NET 2003	C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml
VS.NET 2005	C:\Program Files\Microsoft Visual Studio 8\Xml\Schemas
VS.NET 2008	C:\Program Files\Microsoft Visual Studio 9.0\Xml\Schemas

Spring's .xsd schemas are located in the directory doc/schema. In that directory is also a NAnt build file to help copy over the .xsd files to the appropriate VS.NET locations. To execute this script simply type 'nant' in the doc/schema directory.

Once you have registered the schema with VS.NET you can adding only the namespace declaration to the objects element,

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net">
  <object id="..." type="...">
    ...
  </object>
  <object id="..." type="...">
    ...
  </object>
  ...
</objects>
```

Once registered, the namespace declaration alone is sufficient to get intellisense and validation of the configuration file from within VS.NET. Alternatively, you can select the .xsd file to use by setting the targetSchema property in the Property Sheet for the configuration file.

As shown in the section Section 5.2.3, “Using the container” Spring.NET supports using .NET's application configuration file as the location to store the object definitions that will be managed by the object factory.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>

  <spring>

    <context>
      <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
      ...
    </objects>

  </spring>

</configuration>
```

In this case VS.NET 2003 will still provide you with intellisense help but you will not be able to fully validate the document as the entire schema for App.config is not known. To be able to validate this document one would need to install the [.NET Configuration File schema](#) and an additional schema that incorporates the <spring> and <context> section in addition to the <objects> would need to be created.

Validating schema is a new feature in VS 2005 or later. It is validating all the time while you edit, you will see any errors that it finds in the Error List window.

Keep these trade offs in mind as you decide where to place the bulk of your configuration information. Conventional wisdom is do quick prototyping with App.config and use another IResource location, file or embedded assembly resource, for serious development.

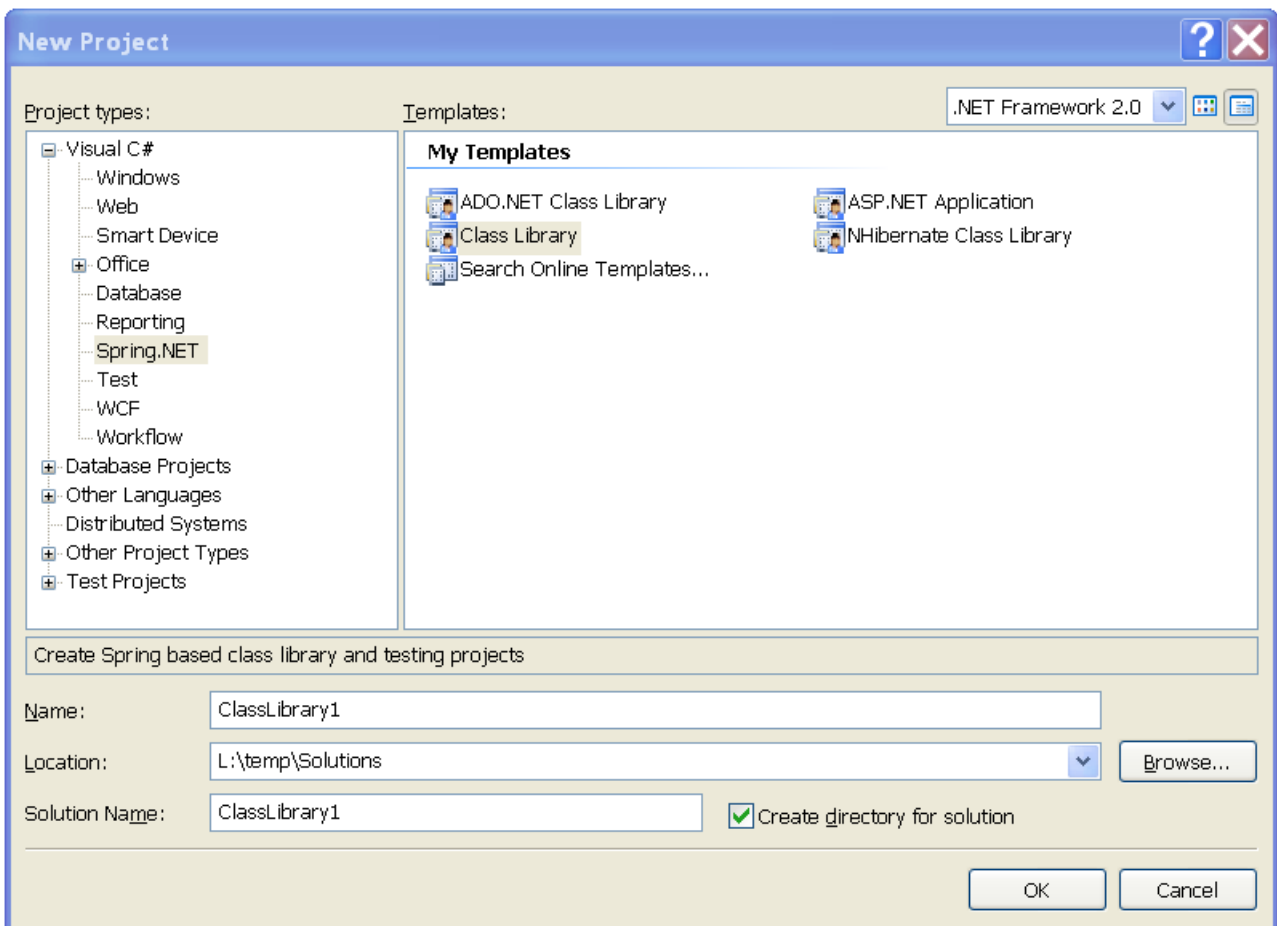
34.2. Solution Templates

Solution templates for VS.NET 2008 are provided to get you up and running quickly with a Spring.NET based application or library. Four templates are provided and there are plans for more. All the templates aside from the web template have been created using SolutionFactory VS.NET Add-in [<http://solutionfactory.codeplex.com/>]. The source to creating the templates is not included in the distribution now, so please download the source from the subversion repository [<https://src.springframework.org/svn/spring-net/trunk>] if you are interested in making modifications.

To install the templates

1. Add the registry key [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\AssemblyFolders\MyAssemblies] and set the value to be the directory <spring.net-install-directory\bin\net\2.0
2. In the directory <spring.net-install-directory>\dev-support\vs.net-2008 run the batch file install-templates.bat

In VS.NET 2008 when you create a new project you will see the category Spring.NET and the four solution templates as shown below

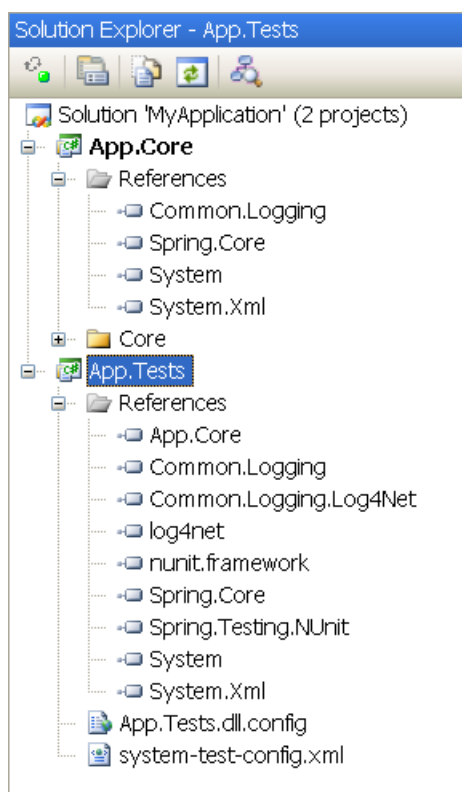


All of the templates have the required Spring dependencies set and Spring application configuration files are present and ready for you to add object definitions.

34.2.1. Class Library

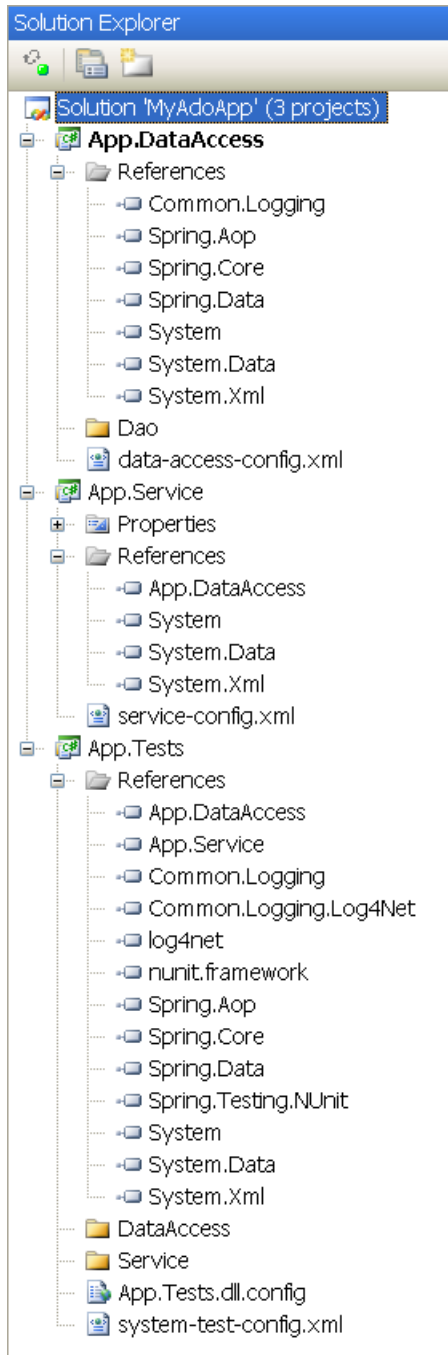
The simplest of the solution templates is the Spring Class Library. This creates a solution with two class library projects, one for you application classes that will be managed by Spring and another testing project. The projects

have starter files to write XML based object definitions and also refer to Spring.NET .dlls as needed. The testing project refers to Spring.Testing.NUnit which provides integration testing support. A screen shot of the generated Class Library solution is shown below.



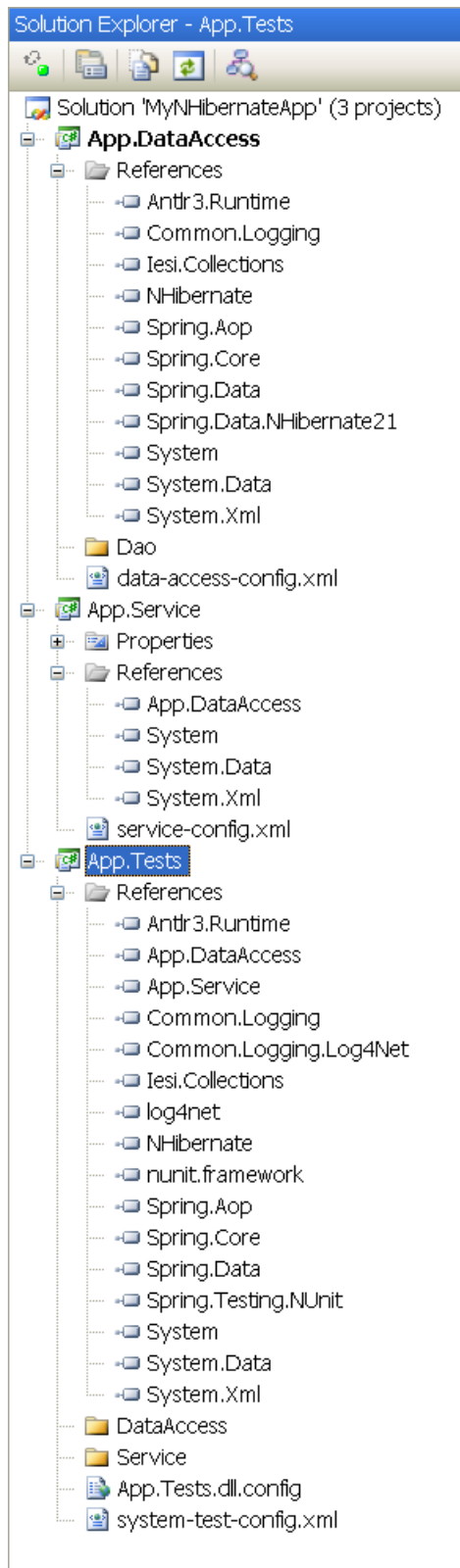
34.2.2. ADO.NET based application library

This solution template provides a service layer project, ADO.NET based data access layer and an unit/integration testing project.



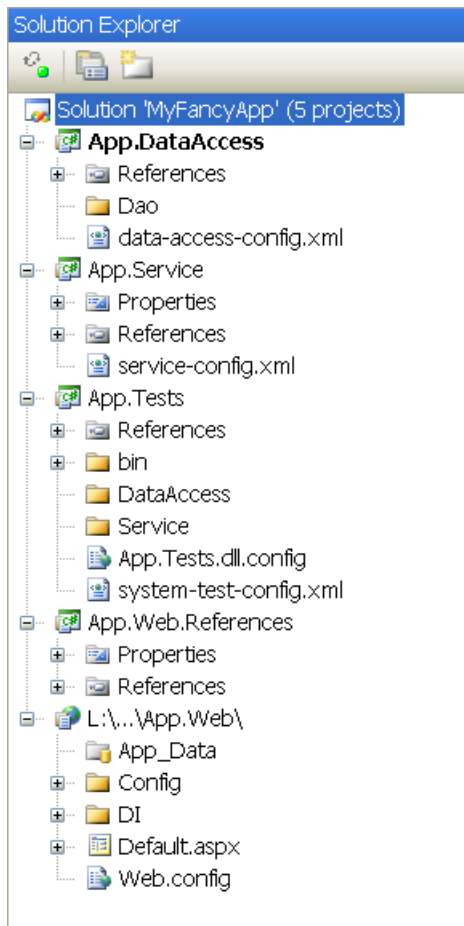
34.2.3. NHibernate based application library

This solution template provides a service layer project, NHibernate based data access layer and an unit/integration testing project.



34.2.4. Spring based web application

This solution template provides a Spring based web layer project, service layer project, ADO.NET based data access layer project and an unit/integration testing project. You will need to set the reference of the App.Web project to refer to the App.Web.References project manually.



34.3. Resharper Type Completion

Resharper supports intellisense completion for the value of the type attribute when editing Spring's XML files. The key combination is Shift+Alt+Space. This is shown below for the case of specifying the type of a DAO object in the NHibernate sample application

```

<!-- Exception translation object post processor -->
<object type="Spring.Dao.Attributes.PersistenceExceptionTranslationPostProcessor, Spring

<!-- Data Access Objects -->
] <object id="CustomerDao" type="Spring.Northwind.Dao.NHibernate.HibernateCustomerDao, Spr
  <property name="SessionFactory" ref="NHibernateSessionFactory"/>
- </object>

<object id="OrderDao" type=""
Start typing prefix to show type name completion
-

```

You start to type the name of the class and will get a filter list. In this case we are typing HibernateOrderDao.

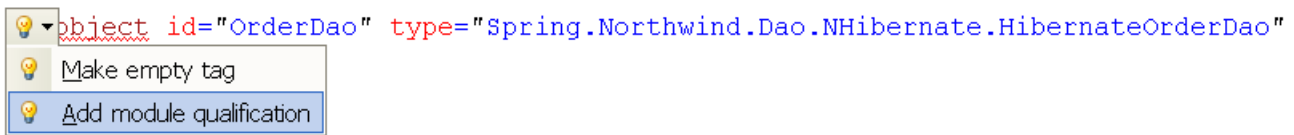
```

<object id="OrderDao" type="HibernateO"

```

Class Spring.Northwind.
Data access object for

Hitting 'enter' will then insert the fully qualified type name with the namespace but not the assembly reference. To add the assembly reference either hit 'CTRL+ENTER' or select the yellow 'light bulb' to and select 'add module qualification'.



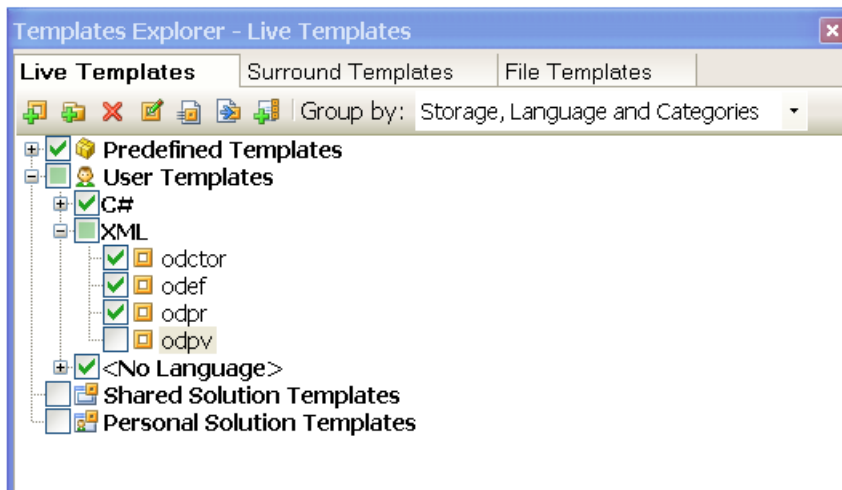
You will need to remove the extraneous 'Verstion' information. This will leave you with the following object definition.

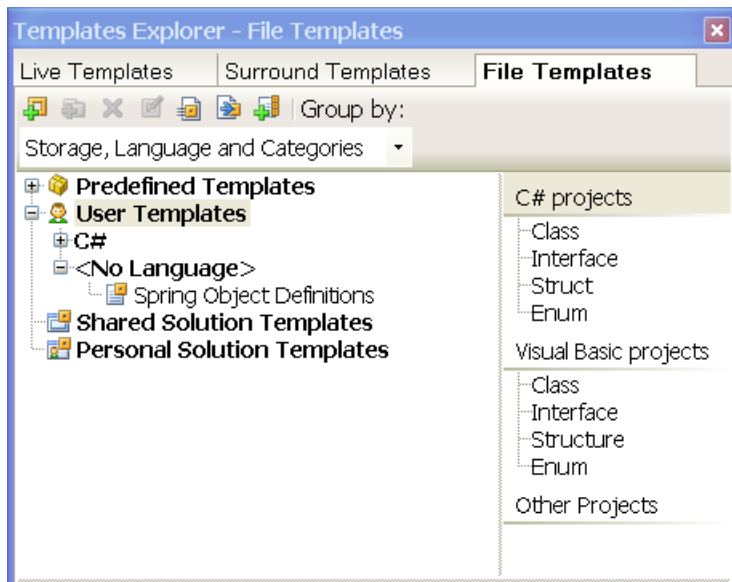
```
<object id="OrderDao" type="Spring.Northwind.Dao.NHibernate.HibernateOrderDao, Spring.Nor
|
</object>
```

If you use Spring's autowiring functionality, then you can even avoid having to type the property information when referring to collaborating objects. See Section 5.3.6, “Autowiring collaborators”. for more information on autowiring.

34.4. Resharper templates

Resharper offers live templates for assistance while coding as well as file templates. Spring 1.3 provides a few of each type to help you be more efficient when performing common configuration related tasks. To install the templates follow the directions in the 'dev-support' directory. One installed the following templates are available





For example, to set a property reference for the object definition from the previous chapter, type 'odpr' (Object Definition Property Reference) and you will be prompted to hit 'tab' to complete the XML fragment.

```
<object id="OrderDao" type="Spring.Northwind.Dao.NHibernate.HibernateOrderDao, Spring.Nor
  odref
</object>
```

Press Tab to expand 'Object Definition Reference' template

Hitting tab will generate the XML to use for an object property values

```
<object id="OrderDao" type="Spring.Northwind.Dao.NHibernate.HibernateOrderDao, Spring.Nor
  <property name="NAME" ref="" />
</object>
```

You will need to type the name of the property and name of the reference. Unfortunately, intellisense for property completion and ref completion is not available. Typing the missing information in then leaves the completed object definition.

```
<object id="OrderDao" type="Spring.Northwind.Dao.NHibernate.HibernateOrderDao, Spring.Nor
  <property name="SessionFactory" ref="NHibernateSessionFactory" />
</object>
```

There are similar live templates for object property values (odpv), object constructors (odctor) and object definitions (odef)

34.5. Versions of XML Schema

The latest version of the schema will always be located under <http://www.springframework.net/xsd/>. The filename of the .xsd files contains the first Spring.NET version to which they apply.

34.6. API documentation

Spring provides API documentation that can be integrated within Visual Studio. There are two versions of the documentation, one for .NET 1.1 and one for .NET 2.0/3.0. They differ only in the format applied and the versions of VS.NET that supported. There is also standalone HTMLHELP format API documentation. You will need to download the help file separately from the distribution.

Part VII. Quickstart applications

This part of the reference documentation covers the quickstart applications included with Spring that demonstrate features in a code centric manner.

- Chapter 35, *IoC Quickstarts*
- Chapter 36, *AOP QuickStart*
- Chapter 37, *Portable Service Abstraction Quick Start*
- Chapter 38, *Web Quickstarts*
- Chapter 39, *SpringAir - Reference Application*
- Chapter 40, *Data Access QuickStart*
- Chapter 41, *Transactions QuickStart*
- Chapter 42, *NHibernate QuickStart*
- Chapter 43, *Quartz QuickStart*
- Chapter 44, *NMS QuickStart*
- Chapter 45, *TIBCO EMS QuickStart*
- Chapter 46, *MSMQ QuickStart*
- Chapter 47, *WCF QuickStart*

Chapter 35. IoC Quickstarts

35.1. Introduction

This chapter includes a grab bag of quickstart examples for using the Spring.NET framework.

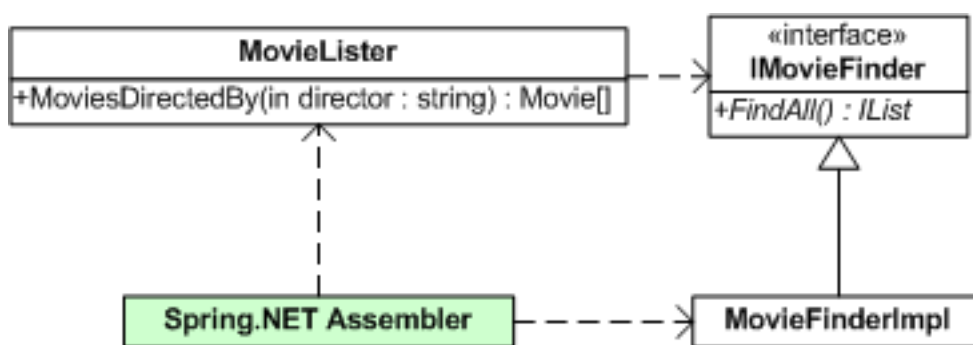
35.2. Movie Finder

The source material for this simple demonstration of Spring.NET's IoC features is lifted straight from Martin Fowler's article that discussed the ideas underpinning the IoC pattern. See [Inversion of Control Containers and the Dependency Injection pattern](#) for more information. The motivation for basing this quickstart example on said article is because the article is pretty widely known, and most people who are looking at IoC for the first time typically will have read the article (at the time of writing a [simple Google search for 'IoC'](#) yields the article in the first five results).

Fowler's article used the example of a search facility for movies to illustrate IoC and Dependency Injection (DI). The article described how a `MovieLister` object might receive a reference to an implementation of the `IMovieFinder` interface (using DI).

The `IMovieFinder` returns a list of all movies and the `MovieLister` filters this list to return an array of `Movie` objects that match a specified directors name. This example demonstrates how the Spring.NET IoC container can be used to supply an appropriate `IMovieFinder` implementation to an arbitrary `MovieLister` instance.

The C# code listings for the MovieFinder application can be found in the `examples/Spring/Spring.Examples.MovieFinder` directory off the top level directory of the Spring.NET distribution.



35.2.1. Getting Started - Movie Finder

The startup class for the MovieFinder example is the `MovieApp` class, which is an ordinary .NET class with a single application entry point...

```
using System;
namespace Spring.Examples.MovieFinder
{
    public class MovieApp
    {
        {
            public static void Main ()
            {
            }
        }
    }
}
```

What we want to do is get a reference to an instance of the `MovieLister` class... since this is a Spring.NET example we'll get this reference from Spring.NET's IoC container, the `IApplicationContext`. There are a

number of ways to get a reference to an `IApplcationContext` instance, but for this example we'll be using an `IApplcationContext` that is instantiated from a custom configuration section in a standard .NET application config file...

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>
  <spring>
    <context>
      <resource uri="config://spring/objects"/>
    </context>
    <objects xmlns="http://www.springframework.net">
      <description>An example that demonstrates simple IoC features.</description>
    </objects>
  </spring>
</configuration>
```

The objects that will be used in the example application will be configured as XML `<object/>` elements nested inside the `<objects/>` element.

The body of the `Main` method in the `MovieApp` class can now be fleshed out a little further...

```
using System;
using Spring.Context;
...
public static void Main ()
{
  IApplicationContext ctx = ContextRegistry.GetContext();
}
...
```

As can be seen in the above C# snippet, a `using` statement has been added to the `MovieApp` source. The `Spring.Context` namespace gives the application access to the `IApplcationContext` class that will serve as the primary means for the application to access the IoC container. The line of code...

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

... retrieves a fully configured `IApplcationContext` implementation that has been configured using the named `<objects/>` section from the application config file.

35.2.2. First Object Definition

As yet, no objects have been defined in the application config file, so let's do that now. The very minimal XML definition for the `MovieLister` instance that we are going to use in the application can be seen in the following XML snippet...

```
<objects xmlns="http://www.springframework.net">
  <object name="MyMovieLister"
    type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
  </object>
</objects>
```

Notice that the full, assembly-qualified name of the `MovieLister` class has been specified in the `type` attribute of the object definition, and that the definition has been assigned the (unique) id of `MyMovieLister`. Using this id, an instance of the object so defined can be retrieved from the `IApplcationContext` reference like so...

```
...
public static void Main ()
```

```

{
    IApplicationContext ctx = ContextRegistry.GetContext();
    MovieLister lister = (MovieLister) ctx.GetObject ("MyMovieLister");
}
...

```

The `lister` instance has not yet had an appropriate implementation of the `IMovieFinder` interface injected into it. Attempting to use the `MoviesDirectedBy` method will most probably result in a nasty `NullReferenceException` since the `lister` instance does not yet have a reference to an `IMovieFinder`. The XML configuration for the `IMovieFinder` implementation that is going to be injected into the `lister` instance looks like this...

```

<objects xmlns="http://www.springframework.net">
    <object name="MyMovieFinder">
        type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
    </object>
</objects>

```

35.2.3. Setter Injection

What we want to do is inject the `IMovieFinder` instance identified by the `MyMovieFinder` id into the `MovieLister` instance identified by the `MyMovieLister` id, which can be accomplished using Setter Injection and the following XML...

```

<objects xmlns="http://www.springframework.net">
    <object name="MyMovieLister">
        type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
            <!-- using setter injection... -->
            <property name="movieFinder" ref="MyMovieFinder"/>
        </object>
    <object name="MyMovieFinder">
        type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
    </object>
</objects>

```

When the `MyMovieLister` object is retrieved from (i.e. instantiated by) the `IApplicationContext` in the application, the Spring.NET IoC container will inject the reference to the `MyMovieFinder` object into the `MovieFinder` property of the `MyMovieLister` object. The `MovieLister` object that is referenced in the application is then fully configured and ready to be used in the application to do what it does best... list movies by director.

```

...
public static void Main ()
{
    IApplicationContext ctx = ContextRegistry.GetContext();
    MovieLister lister = (MovieLister) ctx.GetObject ("MyMovieLister");
    Movie[] movies = lister.MoviesDirectedBy("Roberto Benigni");
    Console.WriteLine ("\nSearching for movie...\n");
    foreach (Movie movie in movies)
    {
        Console.WriteLine (
            string.Format ("Movie Title = '{0}', Director = '{1}'.",
                movie.Title, movie.Director));
    }
    Console.WriteLine ("\nMovieApp Done.\n\n");
}
...

```

To help ensure that the XML configuration of the `MovieLister` class must specify a value for the `MovieFinder` property, you can add the `[Required]` attribute to the `MovieLister`'s `MovieFinder` property. The example code shows uses this attribute. For more information on using and configuring the `[Required]` attribute, refer to this section of the reference documentation.

35.2.4. Constructor Injection

Let's define another implementation of the `IMovieFinder` interface in the application config file...

```
...
<object name="AnotherMovieFinder"
    type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Spring.Examples.MovieFinder">
</object>
...
```

This XML snippet describes an `IMovieFinder` implementation that uses a colon delimited text file as its movie source. The C# source code for this class defines a single constructor that takes a `System.IO.FileInfo` as its single constructor argument. As this object definition currently stands, attempting to get this object out of the `IApplicationContext` in the application with a line of code like so...

```
IMovieFinder finder = (IMovieFinder) ctx.GetObject ("AnotherMovieFinder");
```

will result in a fatal `Spring.Objects.Factory.ObjectCreationException`, because the `Spring.Examples.MovieFinder.ColonDelimitedMovieFinder` class does not have a default constructor that takes no arguments. If we want to use this implementation of the `IMovieFinder` interface, we will have to supply an appropriate constructor argument...

```
...
<object name="AnotherMovieFinder"
    type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Spring.Examples.MovieFinder">
    <constructor-arg index="0" value="movies.txt" />
</object>
...
```

Unsurprisingly, the `<constructor-arg/>` element is used to supply constructor arguments to the constructors of managed objects. The Spring.NET IoC container uses the functionality offered by `System.ComponentModel.TypeConverter` specializations to convert the `movies.txt` string into an instance of the `System.IO.FileInfo` that is required by the single constructor of the `Spring.Examples.MovieFinder.ColonDelimitedMovieFinder` (see Section 6.3, “Type conversion” for a more in depth treatment concerning the automatic type conversion functionality offered by Spring.NET).

So now we have two implementations of the `IMovieFinder` interface that have been defined as distinct object definitions in the config file of the example application; if we wanted to, we could switch the implementation that the `MyMovieLister` object uses like so...

```
...
<object name="MyMovieLister"
    type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
    <!-- lets use the colon delimited implementation instead -->
    <property name="movieFinder" ref="AnotherMovieFinder" />
</object>
<object name="MyMovieFinder"
    type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder" />
</object>
<object name="AnotherMovieFinder"
    type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Spring.Examples.MovieFinder">
    <constructor-arg index="0" value="movies.txt" />
</object>
...
```

Note that there is no need to recompile the application to effect this change of implementation... simply changing the application config file and then restarting the application will result in the Spring.NET IoC container injecting the colon delimited implementation of the `IMovieFinder` interface into the `MyMovieLister` object.

35.2.5. Summary

This example application is quite simple, and admittedly it doesn't do a whole lot. It does however demonstrate the basics of wiring together an object graph using an intuitive XML format. These simple features will get you through pretty much 80% of your object wiring needs. The remaining 20% of the available configuration options

are there to cover corner cases such as factory methods, lazy initialization, and suchlike (all of the configuration options are described in detail in the Chapter 5, *The IoC container*).

35.2.6. Logging

Often enough the first use of Spring.NET is also a first introduction to log4net. To kick start your understanding of log4net this section gives a quick overview. The authoritative place for information on log4net is the [log4net website](#). Other good online tutorials are [Using log4net \(OnDotNet article\)](#) and [Quick and Dirty Guide to Configuring Log4Net For Web Applications](#). Spring.NET is using version 1.2.9 whereas most of the documentation out there is for version 1.2.0. There have been some changes between the two so always double check at the log4net web site for definitive information. Also note that we are investigating using a "commons" logging library so that Spring.NET will not be explicitly tied to log4net but will be able to use other logging packages such as NLog and Microsoft enterprise logging application block.

The general usage pattern for log4net is to configure your loggers, (either in App/Web.config or a separate file), initialize log4net in your main application, declare some loggers in code, and then log log log. (Sing along...) We are using App.config to configure the loggers. As such, we declare the log4net configuration section handler as shown below

```
<section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
```

The corresponding configuration section looks like this

```
<log4net>
  <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%date [%thread] %-5level %logger - %message%newline" />
    </layout>
  </appender>

  <!-- Set default logging level to DEBUG -->
  <root>
    <level value="DEBUG" />
    <appender-ref ref="ConsoleAppender" />
  </root>

  <!-- Set logging for Spring to INFO. Logger names in Spring correspond to the namespace -->
  <logger name="Spring">
    <level value="INFO" />
  </logger>
</log4net>
```

The appender is the output sink - in this case the console. There are a large variety of output sinks such as files, databases, etc. Refer to the log4net [Config Examples](#) for more information. Of interest as well is the PatternLayout which defines exactly the information and format of what gets logged. Usually this is the date, thread, logging level, logger name, and then finally the log message. Refer to [PatternLayout Documentation](#) for information on how to customize.

The logging name is up to you to decide when you declare the logger in code. In the case of this example we used the convention of giving the logging name the name of the fully qualified class name.

```
private static readonly ILog LOG = LogManager.GetLogger(typeof (MovieApp));
```

Other conventions are to give the same logger name across multiple classes that constitute a logical component or subsystem within the application, for example a data access layer. One tip in selecting the pattern layout is to shorten the logging name to only the last 2 parts of the fully qualified name to avoid the message sneaking off to the right too much (where can't see it) because of all the other information logged that precedes it. Shortening the logging name is done using the format %logger{2}.

To initialize the logging system add the following to the start of your application

```
XmlConfigurator.Configure();
```

Note that if you are using or reading information on version 1.2.0 this used to be called `DOMConfigurator.Configure()`;

The logger sections associate logger names with logging levels and appenders. You have great flexibility to mix and match names, levels, and appenders. In this case we have defined the root logger (using the special tag `root`) to be at the debug level and have an console sink. We can then specialize other loggers with different setting. In this case, loggers that start with "Spring" in their name are logged at the info level and also sent to the console. Setting the value of this logger from INFO to DEBUG will show you detailed logging information as the Spring container goes about its job of creating and configuring your objects. Coincidentally, the example code itself uses Spring in the logger name, so this logger also controls the output level you see from running MainApp. Finally, you are ready to use the simple logger api to log, i.e.

```
LOG.Info("Searching for movie...");
```

Logging exceptions is another common task, which can be done using the error level

```
try {
    //do work
}
catch (Exception e)
{
    LOG.Error("Movie Finder is broken.", e);
}
```

35.3. ApplicationContext and IMessageSource

35.3.1. Introduction

The example program `Spring.Examples.AppContext` shows the use of the application context for text localization, retrieving objects contained in `ResourceSets`, and applying the values of embedded resource properties to an object. The values that are retrieved are displayed in a window.

The application context configuration file contains an object definition with the name `messageSource` of the type `Spring.Context.Support.ResourceSetMessageSource` which implements the interface `IMessageSource`. This interface provides various methods for retrieving localized resources such as text and images as described in Section 5.12.2, "Using IMessageSource". When creating an instance of `IApplicationContext`, an object with the name 'messageSource' is searched for and used as the implementation for the context's `IMessageSource` functionality.

The `ResourceSetMessageSource` takes a list of `ResourceManagers` to define the collection of culture-specific resources. The `ResourceManager` can be constructed in two ways. The first way is to specifying a two part string consisting of the base resource name and the containing assembly. In this example there is an embedded resource file, `Images.resx` in the project. The second way is to use helper factory class `ResourceManagerFactoryObject` that takes a resource base name and assembly name as properties. This second way of specifying a `ResourceManager` is useful if you would like direct access to the `ResourceManager` in other parts of your application. In the example program an embedded resource file, `MyResource.resx` and a Spanish specific resource file, `MyResources.es.resx` are declared in this manner. The corresponding XML fragment is shown below

```
...
<object name="messageSource" type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
  <property name="resourceManagers">
    <list>
```

```

        <value>Spring.Examples.AppContext.Images, Spring.Examples.AppContext</value>
        <ref object="myResourceManager"/>
    </list>
</property>
</object>

<object name="myResourceManager" type="Spring.Objects.Factory.Config.ResourceManagerFactoryObject, Spring.Core">
    <property name="baseName">
        <value>Spring.Examples.AppContext.MyResource</value>
    </property>
    <property name="assemblyName">
        <value>Spring.Examples.AppContext</value>
    </property>
</object>
...

```

The main application creates the application context and then retrieves various resources via their key names. In the code all the key names are declared as static fields in the class `Keys`. The resource file `Images.resx` contains image data under the key name `bubblechamber` (aka `Keys.BUBBLECHAMBER`). The code `Image image = (Image)ctx.GetResourceObject(Keys.BUBBLECHAMBER);` is used to retrieve the image from the context. The resource file `MyResource.resx` contains a text resource, `Hello {0} {1}` under the key name `HelloMessage` (aka `Keys.HELLO_MESSAGE`) that can be used for string text formatting purposes. The example code

```

string msg = ctx.GetMessage(Keys.HELLO_MESSAGE,
    CultureInfo.CurrentCulture,
    "Mr.", "Anderson");

```

retrieves the text string and replaces the placeholders in the string with the passed argument values resulting in the text, "Hello Mr. Anderson". The current culture is used to select the resource file `MyResource.resx`. If instead the Spanish culture is specified

```

CultureInfo spanishCultureInfo = new CultureInfo("es");
string esMsg = ctx.GetMessage(Keys.HELLO_MESSAGE,
    spanishCultureInfo,
    "Mr.", "Anderson");

```

Then the resource file `MyResource.es.resx` is used instead as in standard .NET localization. Spring is simply delegating to .NET `ResourceManager` to select the appropriate localized resource. The Spanish version of the resource differs from the English one in that the text under the key `HelloMessage` is `Hola {0} {1}` resulting in the text "Hola Mr. Anderson".

As you can see in this example, the title "Mr." should not be used in the case of the spanish localization. The title can be abstracted out into a key of its own, called `FemaleGreeting` (aka `Keys.FEMALE_GREETING`). The replacement value for the message argument `{0}` can then be made localization aware by wrapping the key in a convenience class `DefaultMessageResolvable`. The code

```

string[] codes = {Keys.FEMALE_GREETING};
DefaultMessageResolvable dmr = new DefaultMessageResolvable(codes, null);

msg = ctx.GetMessage(Keys.HELLO_MESSAGE,
    CultureInfo.CurrentCulture,
    dmr, "Anderson");

```

will assign `msg` the value, Hello Mrs. Anderson, since the value for the key `FemaleGreeting` in `MyResource.resx` is 'Mrs.' Similarly, the code

```

esMsg = ctx.GetMessage(Keys.HELLO_MESSAGE,
    spanishCultureInfo,
    dmr, "Anderson");

```


will assign `esMsg` the value, `Hola Senora Anderson`, since the value for the key `FemaleGreeting` in `MyResource.es.resx` is `'Senora'`.

Localization can also apply to objects and not just strings. The .NET 1.1 framework provides the utility class `ComponentResourceManager` that can apply multiple resource values to object properties in a performant manner. (VS.NET 2005 makes heavy use of this class in the code it generates for winform applications.) The example program has a simple class, `Person`, that has an integer property `Age` and a string property `Name`. The resource file, `Person.resx` contains key names that follow the pattern, `person.<PropertyName>`. In this case it contains `person.Name` and `person.Age`. The code to assign these resource values to an object is shown below

```
Person p = new Person();
ctx.ApplyResources(p, "person", CultureInfo.CurrentUICulture);
```

While you could also use the `Spring` itself to set the properties of these objects, the configuration of simple properties using `Spring` will not take into account localization. It may be convenient to combine approaches and use `Spring` to configure the `Person`'s object references while using `IApplicationContext` inside an `AfterPropertiesSet` callback (see `IInitializingObject`) to set the `Person`'s culture aware properties.

35.4. ApplicationContext and IEventRegistry

35.4.1. Introduction

The example program `Spring.Examples.EventRegistry` shows how to use the application context to wire .NET events in a loosely coupled manner.

Loosely coupled eventing is normally associated with Message Oriented Middleware (MOM) where a daemon process acts as a message broker between other independent processes. Processes communicate indirectly with each other by sending messages through the message broker. The process that initiates the communication is known as a publisher and the process that receives the message is known as the subscriber. By using an API specific to the middleware these processes register themselves as either publishers or subscribers with the message broker. The communication between the publisher and subscriber is considered loosely coupled because neither the publisher nor subscriber has a direct reference to each other, the messages broker acts as an intermediary between the two processes. The `IEventRegistry` is the analogue of the message broker as applied to .NET events. Publishers are classes that invoke a .NET event, subscribers are the classes that register interest in these events, and the messages sent between them are instances of `System.EventArgs`. The implementation of `IEventRegistry` determines the exact semantics of the notification style and coupling between subscribers and publishers.

The `IApplicationContext` interface extends the `IEventRegistry` interface and implementations of `IApplicationContext` delegate the event registry functionality to an instance of `Spring.Objects.Events.Support.EventRegistry`. `IEventRegistry` is a simple interface with one `publish` method and two `subscribe` methods. Refer to Section 5.12.4, “Loosely coupled events” for a reminder of their signatures. The `Spring.Objects.Events.Support.EventRegistry` implementation is essentially a convenience to decouple the event wiring process between publisher and subscribers. In this implementation, after the event wiring is finished, publishers are directly coupled to the subscribers via the standard .NET eventing mechanisms. Alternate implementations could increase the decoupling further by having the event registry subscribe to the events and be responsible for then notifying the subscribers.

In this example the class `MyClientEventArgs` is a subclass of `System.EventArgs` that defines a string property `EventMessage`. The class `MyEventPublisher` defines a public event with the delegate signature `void SimpleClientEvent(object sender, MyClientEventArgs args)`. The method `void`

`ClientMethodThatTriggersEvent1()` fires this event. On the subscribing side, the class `MyEventSubscriber` contains a method, `HandleClientEvents` that matches the delegate signature and has a boolean property which is set to true if this method is called.

The publisher and subscriber classes are defined in an application context configuration file but that is not required in order to participate with the event registry. The main program, `EventRegistryApp` creates the application context and asks it for an instance of `MyEventPublisher`. The publisher is registered with the event registry via the call, `ctx.PublishEvents(publisher)`. The event registry keeps a reference to this publisher for later use to register any subscribers that match its event signature. Two subscribers are then created and one of them is wired to the publisher by calling the method `ctx.Subscribe(subscriber, typeof(MyEventPublisher))`. Specifying the type indicates that the subscriber should be registered only to events from objects of the type `MyEventPublisher`. This acts as a simple filtering mechanism on the subscriber.

The publisher then fires the event using normal .NET eventing semantics and the subscriber is called. The subscriber prints a message to the console and sets a state variable to indicate it has been called. The program then simply prints the state variable of the two subscribers, showing that only one of them (the one that registered with the event registry) was called.

35.5. Pooling example

The idea is to build an executor backed by a pool of `QueuedExecutor`: this will show how Spring.NET provides some useful low-level/high-quality reusable threading and pooling abstractions. This executor will provide parallel executions (in our case `grep`-like file scans). *Note: This example is not in the 1.0.0 release to its use of classes in the `Spring.Threading` namespace scheduled for release in Spring 1.1. To access this example please get the code from CVS ([instructions](#)) or from the download section of the Spring.NET website that contains an .zip with the full CVS tree.*

Some information on `QueuedExecutor` is helpful to better understand the implementation and to possibly disagree with it. Keep in mind that the point is to show how to develop your own object-pool.

A `QueuedExecutor` is an executor where `IRunnable` instances are run serially by a worker thread. When you `Execute` with a `QueuedExecutor`, your request is queued; at some point in the future your request will be taken and executed by the worker thread; in case of error the thread is terminated. However this executor recreates its worker thread as needed.

Last but not least, this executor can be shut down in a few different ways (please refer to the Spring.NET SDK documentation). Given its simplicity, it is very powerful.

The example project `Spring.Examples.Pool` provides an implementation of a pooled executor, backed by `n` instances of `Spring.Threading.QueuedExecutor`: please ignore the fact that `Spring.Threading` includes already a very different implementation of a `PooledExecutor`: here we want to use a pool of `QueuedExecutors`.

This executor will be used to implement a parallel recursive `grep`-like console executable.

35.5.1. Implementing `spring.Pool.IPoolableObjectFactory`

In order to use the `SimplePool` implementation, the first thing to do is to implement the `IPoolableObjectFactory` interface. This interface is intended to be implemented by objects that can create the type of objects that should be pooled. The `SimplePool` will call the lifecycle methods on `IPoolableObjectFactory` interface (`MakeObject`, `ActivateObject`, `ValidateObject`, `PassivateObject`, and `DestroyObject`) as appropriate when the pool is created, objects are borrowed and returned to the pool, and when the pool is destroyed.

In our case, as already said, we want to implement a pool of `QueuedExecutor`. Ok, here the declaration:

```
public class QueuedExecutorPoolableFactory : IPoolableObjectFactory
{
```

the first task a factory should do is to create objects:

```
object IPoolableObjectFactory.MakeObject()
{
    // to actually make this work as a pooled executor
    // use a bounded queue of capacity 1.
    // If we don't do this one of the queued executors
    // will accept all the queued IRunnable as, by default
    // its queue is unbounded, and the PooledExecutor
    // will happen to always run only one thread ...
    return new QueuedExecutor(new BoundedBuffer(1));
}
```

and should be also able to destroy them:

```
void IPoolableObjectFactory.DestroyObject(object o)
{
    // ah, self documenting code:
    // Here you can see that we decided to let the
    // executor process all the currently queued tasks.
    QueuedExecutor executor = o as QueuedExecutor;
    executor.ShutdownAfterProcessingCurrentlyQueuedTasks();
}
```

When an object is taken from the pool, to satisfy a client request, may be the object should be activated. We can possibly implement the activation like this:

```
void IPoolableObjectFactory.ActivateObject(object o)
{
    QueuedExecutor executor = o as QueuedExecutor;
    executor.Restart();
}
```

even if a `QueuedExecutor` restarts itself as needed and so a valid implementation could leave this method empty.

After activation, and before the pooled object can be successfully returned to the client, it is validated (should the object be invalid, it will be discarded: this can lead to an empty unusable pool ¹). Here we check that the worker thread exists:

```
bool IPoolableObjectFactory.ValidateObject(object o)
{
    QueuedExecutor executor = o as QueuedExecutor;
    return executor.Thread != null;
}
```

Passivation, symmetrical to activation, is the process a pooled object is subject to when the object is returned to the pool. In our case we simply do nothing:

```
void IPoolableObjectFactory.PassivateObject(object o)
{
}
```

At this point, creating a pool is simply a matter of creating an `SimplePool` as in:

```
pool = new SimplePool(new QueuedExecutorPoolableFactory(), size);
```

¹You may think that we can provide a smarter implementation and you are probably right. However, it is not so difficult to create a new pool in case the old one became unusable. It could not be your preferred choice but surely it leverages simplicity and object immutability

35.5.2. Being smart using pooled objects

Taking advantage of the `using` keyword seems to be very important in these c# days, so we implement a very simple helper (`PooledObjectHolder`) that can allow us to do things like:

```
using (PooledObjectHolder holder = PooledObjectHolder.UseFrom(pool))
{
    QueuedExecutor executor = (QueuedExecutor) holder.Pooled;
    executor.Execute(runnable);
}
```

without worrying about obtaining and returning an object from/to the pool.

Here is the implementation:

```
public class PooledObjectHolder : IDisposable
{
    IObjectPool pool;
    object pooled;

    /// <summary>
    /// Builds a new <see cref="PooledObjectHolder"/>
    /// trying to borrow an object form it
    /// </summary>
    /// <param name="pool"></param>
    private PooledObjectHolder(IObjectPool pool)
    {
        this.pool = pool;
        this.pooled = pool.BorrowObject();
    }

    /// <summary>
    /// Allow to access the borrowed pooled object
    /// </summary>
    public object Pooled
    {
        get
        {
            return pooled;
        }
    }

    /// <summary>
    /// Returns the borrowed object to the pool
    /// </summary>
    public void Dispose()
    {
        pool.ReturnObject(pooled);
    }

    /// <summary>
    /// Creates a new <see cref="PooledObjectHolder"/> for the
    /// given pool.
    /// </summary>
    public static PooledObjectHolder UseFrom(IObjectPool pool)
    {
        return new PooledObjectHolder(pool);
    }
}
```

Please don't forget to destroy all the pooled instances once you have finished! How? Well using something like this in `PooledQueuedExecutor`:

```
public void Stop ()
{
    // waits for all the grep-task to have been queued ...
    foreach (ISync sync in syncs)
    {
        sync.Acquire();
    }
}
```

```
pool.Close();
}
```

35.5.3. Using the executor to do a parallel grep

The use of the just built executor is quite straightforward but a little tricky if we want to really exploit the pool.

```
private PooledQueuedExecutor executor;

public ParallelGrep(int size)
{
    executor = new PooledQueuedExecutor(size);
}

public void Recurse(string startPath, string filePattern, string regexPattern)
{
    foreach (string file in Directory.GetFiles(startPath, filePattern))
    {
        executor.Execute(new Grep(file, regexPattern));
    }
    foreach (string directory in Directory.GetDirectories(startPath))
    {
        Recurse(directory, filePattern, regexPattern);
    }
}

public void Stop()
{
    executor.Stop();
}
```

```
public static void Main(string[] args)
{
    if (args.Length < 3)
    {
        Console.WriteLine("usage: {0} regex directory file-pattern [pool-size]", Assembly.GetEntryAssembly().CodeBase);
        Environment.Exit(1);
    }

    string regexPattern = args[0];
    string startPath = args[1];
    string filePattern = args[2];
    int size = 10;
    try
    {
        size = Int32.Parse(args[3]);
    }
    catch
    {
    }
    Console.WriteLine ("pool size {0}", size);

    ParallelGrep grep = new ParallelGrep(size);
    grep.Recurse(startPath, filePattern, regexPattern);
    grep.Stop();
}
```

35.6. AOP

Refer to Chapter 36, *AOP QuickStart*.

Chapter 36. AOP QuickStart

36.1. Introduction

This is an introductory guide to Aspect Oriented Programming (AOP) with Spring.NET.

This guide assumes little to no prior experience of having *used* Spring.NET AOP on the part of the reader. However, it *does* assume a certain familiarity with the terminology of AOP in general. It is probably better if you have read (or at least have skimmed through) the AOP section of the reference documentation beforehand, so that you are familiar with a) just what AOP is, b) what problems AOP is addressing, and c) what the AOP concepts of advice, pointcut, and joinpoint actually mean... this guide spends absolutely zero time defining those terms. Having said all that, if you are the kind of developer who learns best by example, then by all means follow along... you can always consult the reference documentation as the need arises (see Section 13.1.1, “AOP concepts”).

*The examples in this guide are **intentionally** simplistic. One of the core aims of this guide is to get you up and running with Spring.NET's flavor of AOP in as short a time as possible. Having to comprehend even a simple object model in order to understand the AOP examples would not be conducive to learning Spring.NET AOP. It is left as an exercise for the reader to take the concepts learned from this guide and apply them to his or her own code base. Again, having said all of that, this guide concludes with a number of cookbook-style AOP 'recipes' that illustrate the application of Spring.NET's AOP offering in a real world context; additionally, the Spring.NET reference application contains a number of Spring.NET AOP aspects particular to it's own domain model (see Chapter 39, SpringAir - Reference Application).*



Note

To follow this AOP QuickStart load the solution file found in the directory <spring-install-dir>\examples\Spring\Spring.AopQuickStart

36.2. The basics

This initial section introduces the basics of defining and then applying some simple advice.

36.2.1. Applying advice

Lets see (a very basic) example of using Spring.NET AOP. The following example code simply applies advice that writes the details of an advised method call to the system console. Admittedly, this is not a particularly compelling or even useful application of AOP, but having worked through the example, you will then hopefully be able to see how to apply your own custom advice to perform useful work (transaction management, auditing, security enforcement, thread safety, etc).

Before looking at the AOP code proper lets quickly look at the domain classes that are the target of the advice (in Spring.NET AOP terminology, an instance of the following class is going to be the *advised object*).

```
public interface ICommand
{
    object Execute(object context);
}

public class ServiceCommand : ICommand
{
    public object Execute(object context)
    {
    }
```

```

        Console.Out.WriteLine("Service implementation : [{0}]", context);
        return null;
    }
}

```

Find below the advice that is going to be applied to the `Execute(object context)` method of the `ServiceCommand` class. As you can see, this is an example of *around advice* (see Section 13.3.2, “Advice types”).

```

public class ConsoleLoggingAroundAdvice : IMethodInterceptor
{
    public object Invoke(IMethodInvocation invocation)
    {
        Console.Out.WriteLine("Advice executing; calling the advised method..."); ❶
        object returnValue = invocation.Proceed(); ❷ ❸
        Console.Out.WriteLine("Advice executed; advised method returned " + returnValue); ❹
        return returnValue; ❺
    }
}

```

- ❶ Some simple code that merely prints out the fact that the advice is executing.
- ❷ The advised method is invoked.
- ❸ The return value is captured in the `returnValue` variable.
- ❹ The value of the captured `returnValue` is printed out.
- ❺ The previously captured `returnValue` is returned.

So thus far we have three artifacts: an interface (`ICommand`); an implementation of said interface (`ServiceCommand`); and some (trivial) advice (encapsulated by the `ConsoleLoggingAroundAdvice` class). All that remains is to actually apply the `ConsoleLoggingAroundAdvice` advice to the invocation of the `Execute()` method of the `ServiceCommand` class. Lets look at how to effect this programmatically...

```

ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingAroundAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute("This is the argument");

```

The result of executing the above snippet of code will look something like this...

```

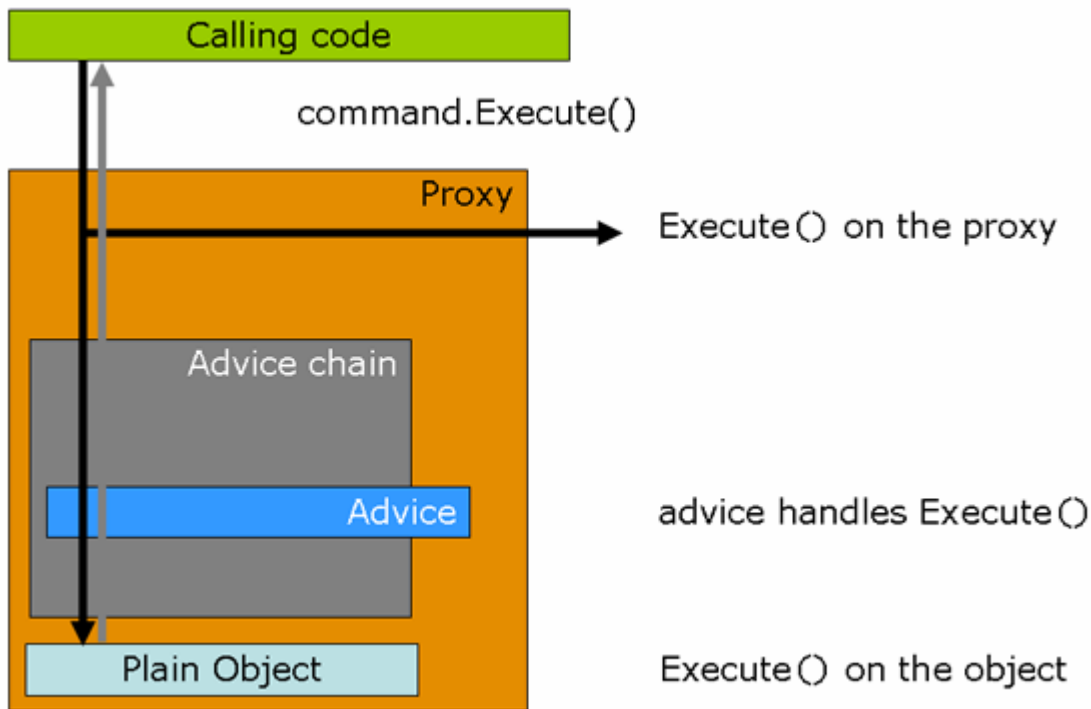
Advice executing; calling the advised method...
Service implementation : [This is the argument]
Advice executed; advised method returned

```

The output shows that the advice (the `Console.Out` statements from the `ConsoleLoggingAroundAdvice` was applied *around* the invocation of the advised method.

So what is happening here? The fact that the preceding code used a class called `ProxyFactory` may have clued you in. The constructor for the `ProxyFactory` class took as an argument the object that we wanted to advise (in this case, an instance of the `ServiceCommand` class). We then added some advice (a `ConsoleLoggingAroundAdvice` instance) using the `AddAdvice()` method of the `ProxyFactory` instance. We then called the `GetProxy()` method of the `ProxyFactory` instance which gave us a proxy... an (AOP) proxy that proxied the target object (the `ServiceCommand` instance), and called the advice (a single instance of the `ConsoleLoggingAroundAdvice` in this case). When we invoked the `Execute(object context)` method of the proxy, the advice was 'applied' (executed), as can be seen from the attendant output.

The following image shows a graphical view of the flow of execution through a Spring.NET AOP proxy.



One thing to note here is that the AOP proxy that was returned from the call to the `getProxy()` method of the `ProxyFactory` instance was cast to the `ICommand` interface that the `ServiceCommand` target object implemented. This is very important... currently, Spring.NET's AOP implementation mandates the use of an interface for advised objects. In short, this means that in order for your classes to leverage Spring.NET's AOP support, those classes that you wish to use with Spring.NET AOP **must** implement at least one interface. In practice this restriction is not as onerous as it sounds... in any case, it is *generally* good practice to program to interfaces anyway (support for applying advice to classes that do not implement any interfaces is planned for a future point release of Spring.NET AOP).

The remainder of this guide is concerned with fleshing out some of the finer details of Spring.NET AOP, but basically speaking, that's about it.

As a first example of fleshing out one of those finer details, find below some Spring.NET XML configuration that does *exactly* the same thing as the previous example; it should also be added that this declarative style approach to Spring.NET AOP is preferred to the programmatic style.

```
<object id="consoleLoggingAroundAdvice"
    type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>
<object id="myServiceObject" type="Spring.Aop.Framework.ProxyFactoryObject">
    <property name="target">
        <object id="myServiceObjectTarget"
            type="Spring.Examples.AopQuickStart.ServiceCommand"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>consoleLoggingAroundAdvice</value>
        </list>
    </property>
</object>
```

```
ICommand command = (ICommand) ctx["myServiceObject"];
command.Execute();
```

Some comments are warranted concerning the above XML configuration snippet. Firstly, note that the `ConsoleLoggingAroundAdvice` is itself a plain vanilla object, and is eligible for configuration just like any other

class... if the advice itself needed to be injected with any dependencies, any such dependencies could be injected as normal.

Secondly, notice that the object definition corresponding to the object that is retrieved from the IoC container is a `ProxyFactoryObject`. The `ProxyFactoryObject` class is an implementation of the `IFactoryObject` interface; `IFactoryObject` implementations are treated specially by the Spring.NET IoC container... in this specific case, it is not a reference to the `ProxyFactoryObject` instance itself that is returned, but rather the object that the `ProxyFactoryObject` produces. In this case, it will be an advised instance of the `ServiceCommand` class.

Thirdly, notice that the target of the `ProxyFactoryObject` is an instance of the `ServiceCommand` class; this is the object that is going to be advised (i.e. invocations of its methods are going to be intercepted). This object instance is defined as an inner object definition... this is the preferred idiom for using the `ProxyFactoryObject`, as it means that other objects cannot acquire a reference to the raw object, but rather only the advised object.

Finally, notice that the advice that is to be applied to the target object is referred to by its object name in the list of the names of interceptors for the `ProxyFactoryObject`'s `interceptorNames` property. In this particular case, there is only one instance of advice being applied... the `ConsoleLoggingAroundAdvice` defined in an object definition of the same name. The reason for using a list of object names as opposed to references to the advice objects themselves is explained in the reference documentation...

'... if the `ProxyFactoryObject`'s `singleton` property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it is necessary to be able to obtain an instance of the prototype from the context; holding a reference isn't sufficient.'

36.2.2. Using Pointcuts - the basics

The advice that was applied in the previous section was rather indiscriminate with regard to which methods on the advised object were to be advised... the `ConsoleLoggingAroundAdvice` simply intercepted **all** methods (that were part of an interface implementation) on the target object.

This is great for simple examples and suchlike, but not so great when you only want certain methods of an object to be advised. For example, you may only want those methods beginning with 'Start' to be advised; or you may only want those methods that are called with specific runtime argument values to be advised; or you may only want those methods that are decorated with a `Lockable` attribute to be advised.

The mechanism that Spring.NET AOP uses to discriminate about where advice is applied (i.e. which method invocations are intercepted) is encapsulated by the `IPointcut` interface (see Section 13.2, "Pointcut API in Spring.NET"). Spring.NET provides many out-of-the-box implementations of the `IPointcut` interface... the implementation that is used if none is explicitly supplied (as was the case with the first example) is the canonical `TruePointcut`: as the name suggests, this pointcut always matches, and hence **all** methods that can be advised will be advised.

So let's change the configuration of the advice such that it is only applied to methods that contain the letters 'Do'. We'll change the `ICommand` interface (and it's attendant implementation) to accommodate this...

```
public interface ICommand
{
    void Execute();

    void DoExecute();
}

public class ServiceCommand : ICommand
{
    public void Execute()
    {
        Console.Out.WriteLine("Service implementation : Execute()...");
    }
}
```

```

    }

    public void DoExecute()
    {
        Console.Out.WriteLine("Service implementation : DoExecute()...");
    }
}

```

Please note that the advice itself (encapsulated within the `ConsoleLoggingAroundAdvice` class) does not need to change; we are changing *where* this advice is applied, and not the advice itself.

Programmatic configuration of the advice, taking into account the fact that we only want methods that contain the letters 'Do' to be advised, looks like this...

```

ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvisor(new DefaultPointcutAdvisor(
    new SdkRegularExpressionMethodPointcut("Do"),
    new ConsoleLoggingAroundAdvice()));
ICommand command = (ICommand) factory.GetProxy();
command.DoExecute();

```

The result of executing the above snippet of code will look something like this...

```

Intercepted call : about to invoke next item in chain...
Service implementation...
Intercepted call : returned

```

The output indicates that the advice was applied around the invocation of the advised method, because the name of the method that was executed contained the letters 'Do'. Try changing the pertinent code snippet to invoke the `Execute()` method, like so...

```

ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvisor(
    new DefaultPointcutAdvisor(
        new SdkRegularExpressionMethodPointcut("Do"),
        new ConsoleLoggingAroundAdvice()));
ICommand command = (ICommand) factory.GetProxy();

// note that there is no 'Do' in this method name
command.Execute();

```

Run the code snippet again; you will see that the advice will not be applied : the pointcut is not matched (the method name does not contain the letters 'Do'), resulting in the following (unadvised) output...

```

Service implementation...

```

XML configuration that accomplishes exactly the same thing as the previous programmatic configuration example can be seen below...

```

<object id="consoleLoggingAroundAdvice"
    type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor">
    <property name="pattern" value="Do"/>
    <property name="advice">
        <object type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>
    </property>
</object>
<object id="myServiceObject"
    type="Spring.Aop.Framework.ProxyFactoryObject">
    <property name="target">
        <object id="myServiceObjectTarget"
            type="Spring.Examples.AopQuickStart.ServiceCommand"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>consoleLoggingAroundAdvice</value>
        </list>
    </property>

```

</object>

You'll will perhaps have noticed that this treatment of pointcuts introduced the concept of an `advisor` (see Section 13.4, “Advisor API in Spring.NET”). An advisor is nothing more the composition of a pointcut (i.e. *where* advice is going to be applied), and the advice itself (i.e. *what* is going to happen at the interception point). The `consoleLoggingAroundAdvice` object defines an advisor that will apply the advice to all those methods of the advised object that match the pattern `'Do'` (the pointcut). The pattern to match against is supplied as a simple string value to the `pattern` property of the `RegularExpressionMethodPointcutAdvisor` class.

36.3. Going deeper

The first section should (hopefully) have demonstrated the basics of firstly defining advice, and secondly, of choosing where to apply that advice using the notion of a pointcut. Of course, there is a great deal more to Spring.NET AOP than the aforementioned single advice type and pointcut. This section continues the exploration of Spring.NET AOP, and describes the various advice and pointcuts that are available for you to use (yes, there is more than one type of advice and pointcut).

36.3.1. Other types of Advice

The advice that was demonstrated and explained in the preceding section is what is termed '*around advice*'. The name '*around advice*' is used because the advice is applied *around* the target method invocation. In the specific case of the `ConsoleLoggingAroundAdvice` advice that was defined previously, the target was made available to the advice as an `IMethodInvocation` object... a call was made to the `Console` class before the target was invoked, and a call was made to the `Console` class after the target method invocation was invoked. The advice surrounded the target, one could even say that the advice was totally 'around' the target... hence the name, '*around advice*'.

'*around advice*' provides one with the opportunity to do things both **before** the target gets a chance to do anything, and **after** the target has returned: one even gets a chance to inspect (and possibly even totally change) the return value.

Sometimes you don't need all that power though. If we stick with the example of the `ConsoleLoggingAroundAdvice` advice, what if one just wants to log the fact that a method was called? In that case one doesn't need to do anything *after* the target method invocation is to be invoked, nor do you need access to the return value of the target method invocation. In fact, you only want to do something *before* the target is to be invoked (in this case, print out a message to the system `Console` detailing the name of the method). In the tradition of good programming that says one should use only what one needs and no more, Spring.NET has another type of advice that one can use... if one only wants to do something *before* the target method invocation is invoked, why bother with having to manually call the `Proceed()` method? The most expedient solution simply is to use '*before advice*'.

36.3.1.1. Before advice

'*before advice*' is just that... it is advice that runs *before* the target method invocation is invoked. One does not get access to the target method invocation itself, and one cannot return a value... this is a good thing, because it means that you cannot inadvertently forget to call the `Proceed()` method on the target, and it also means that you cannot inadvertently forget to return the return value of the target method invocation. If you don't need to inspect or change the return value, or even do anything after the successful execution of the target method invocation, then '*before advice*' is just what you need.

'*before advice*' in Spring.NET is defined by the `IMethodBeforeAdvice` interface in the `Spring.Aop` namespace. Lets just dive in with an example... we'll use the same scenario as before to keep things simple. Let's define the '*before advice*' implementation first.

```

public class ConsoleLoggingBeforeAdvice : IMethodBeforeAdvice
{
    public void Before(MethodInfo method, object[] args, object target)
    {
        Console.Out.WriteLine("Intercepted call to this method : " + method.Name);
        Console.Out.WriteLine("    The target is           : " + target);
        Console.Out.WriteLine("    The arguments are       : ");
        if(args != null)
        {
            foreach (object arg in args)
            {
                Console.Out.WriteLine("\t: " + arg);
            }
        }
    }
}

```

Let's apply a single instance of the `ConsoleLoggingBeforeAdvice` advice to the invocation of the `Execute()` method of the `ServiceCommand`. What follows is programmatic configuration; as you can see, its pretty much identical to the previous version... the only difference is that we're using our new '*before advice*' (encapsulated as an instance of the `ConsoleLoggingBeforeAdvice` class).

```

ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();

```

The result of executing the above snippet of code will look something like this...

```

Intercepted call to this method : Execute
The target is                   : Spring.Examples.AopQuickStart.ServiceCommand
The arguments are               :

```

The output clearly indicates that the advice was applied **before** the invocation of the advised method. Notice that in contrast to '*around advice*', with '*before advice*' there is no chance of forgetting to call the `Proceed()` method on the target, because one does not have access to the `IMethodInvocation` (as is the case with '*around advice*')... similarly, you cannot forget to return the return value either.

If you can use '*before advice*', then do so. The simpler programming model offered by '*before advice*' means that there is less to remember, and thus potentially less things to get wrong.

Here is the Spring.NET XML configuration for applying our '*before advice*' declaratively...

```

<object id="beforeAdvice"
    type="Spring.Examples.AopQuickStart.ConsoleLoggingBeforeAdvice"/>

<object id="myServiceObject"
    type="Spring.Aop.Framework.ProxyFactoryObject">
    <property name="target">
        <object id="myServiceObjectTarget"
            type="Spring.Examples.AopQuickStart.ServiceCommand"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>beforeAdvice</value>
        </list>
    </property>
</object>

```

36.3.1.2. After advice

Just as '*before advice*' defines advice that executes **before** an advised target, '*after advice*' is advice that executes **after** a target has been executed.

'*after advice*' in Spring.NET is defined by the `IAfterReturningAdvice` interface in the `Spring.Aop` namespace. Again, let's just fire on ahead with an example... again, we'll use the same scenario as before to keep things simple.

```
public class ConsoleLoggingAfterAdvice : IAfterReturningAdvice
{
    public void AfterReturning(
        object returnValue, MethodInfo method, object[] args, object target)
    {
        Console.Out.WriteLine("This method call returned successfully : " + method.Name);
        Console.Out.WriteLine("    The target was                : " + target);
        Console.Out.WriteLine("    The arguments were          : ");
        if(args != null)
        {
            foreach (object arg in args)
            {
                Console.Out.WriteLine("\t: " + arg);
            }
        }
        Console.Out.WriteLine("    The return value is          : " + returnValue);
    }
}
```

Let's apply a single instance of the `ConsoleLoggingAfterAdvice` advice to the invocation of the `Execute()` method of the `ServiceCommand`. What follows is programmatic configuration; as you can, it's pretty much identical to the '*before advice*' version (which in turn was pretty much identical to the original '*around advice*' version)... the only real difference is that we're using our new '*after advice*' (encapsulated as an instance of the `ConsoleLoggingAfterAdvice` class).

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingAfterAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
This method call returned successfully : Execute
The target was                        : Spring.Examples.AopQuickStart.ServiceCommand
The arguments were                     :
The return value is                   : null
```

The output clearly indicates that the advice was applied **after** the invocation of the advised method. Again, it bears repeating that your real world development will actually have an advice implementation that does something useful after the invocation of an advised method. Notice that in contrast to '*around advice*', with '*after advice*' there is no chance of forgetting to call the `Proceed()` method on the target, because just like '*before advice*' you don't have access to the `IMethodInvocation`... similarly, although you get access to the return value of the target, you cannot forget to return the return value either. You can however change the state of the return value, typically by setting some of its properties, or by calling methods on it.

The best-practice rule for '*after advice*' is much the same as it is for '*before advice*'; namely that if you can use '*after advice*', then do so (in preference to using '*around advice*'). The simpler programming model offered by '*after advice*' means that there is less to remember, and thus less things to get potentially wrong.

A possible use case for '*after advice*' would include performing access control checks on the return value of an advised method invocation; consider the case of a service that returns a list of document URI's... depending on the identity of the (Windows) user that is running the program that is calling this service, one could strip out those URI's that contain sensitive data for which the user does not have sufficient privileges to access. That is just one (real world) scenario... I'm sure you can think of plenty more that are a whole lot more relevant to your own development needs.

Here is the Spring.NET XML configuration for applying the '*after advice*' declaratively...

```

<object id="afterAdvice"
  type="Spring.Examples.AopQuickStart.ConsoleLoggingAfterAdvice"/>

<object id="myServiceObject"
  type="Spring.Aop.Framework.ProxyFactoryObject">
  <property name="target">
    <object id="myServiceObjectTarget"
      type="Spring.Examples.AopQuickStart.ServiceCommand"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>afterAdvice</value>
      </list>
    </property>
  </object>

```

36.3.1.3. Throws advice

So far we've covered '*around advice*', '*before advice*', and '*after advice*'... these advice types will see you through most if not all of your AOP needs. However, one of the remaining advice types that Spring.NET has in its locker is '*throws advice*'.

'*throws advice*' is advice that executes when an advised method invocation *throws* an exception.. hence the name. One basically applies the '*throws advice*' to a target object in much the same way as any of the previously mentioned advice types. If during the execution of ones application none of any of the advised methods throws an exception, then the '*throws advice*' will never execute. However, if during the execution of your application an advised method *does* throw an exception, then the '*throws advice*' will kick in and be executed. You can use '*throws advice*' to apply a common exception handling policy across the various objects in your application, or to perform logging of every exception thrown by an advised method, or to alert (perhaps via email) the support team in the case of particularly of critical exceptions... the list of possible uses cases is of course endless.

The '*throws advice*' type in Spring.NET is defined by the `IThrowsAdvice` interface in the `Spring.Aop` namespace... basically, one defines on one's '*throws advice*' implementation class what types of exception are going to be handled. Lets take a quick look at the `IThrowsAdvice` interface...

```

public interface IThrowsAdvice : IAdvice
{
}

```

Yes, that is really it... it is a marker interface that has no methods on it. You may be wondering how Spring.NET determines which methods to call to effect the running of one's '*throws advice*'. An example would perhaps be illustrative at this point, so here is some simple Spring.NET style '*throws advice*'...

```

public class ConsoleLoggingThrowsAdvice : IThrowsAdvice
{
    public void AfterThrowing(Exception ex)
    {
        Console.Out.WriteLine("Advised method threw this exception : " + ex);
    }
}

```

Lets also change the implementation of the `Execute()` method of the `ServiceCommand` class such that it throws an exception. This will allow the advice encapsulated by the above `ConsoleLoggingThrowsAdvice` to kick in.

```

public class ServiceCommand : ICommand
{
    public void Execute()
    {
        throw new UnauthorizedAccessException();
    }
}

```

Let's programmatically apply the *'throws advice'* (an instance of our `ConsoleLoggingThrowsAdvice`) to the invocation of the `Execute()` method of the above `ServiceCommand` class; to wit...

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
Advised method threw this exception : System.UnauthorizedAccessException:
Attempted to perform an unauthorized operation.
```

As can be seen from the output, the `ConsoleLoggingThrowsAdvice` kicked in when the advised method invocation threw an exception. There are a number of things to note about the `ConsoleLoggingThrowsAdvice` advice class, so let's take them each in turn.

In Spring.NET, *'throws advice'* means that you have to define a class that implements the `IThrowsAdvice` interface. Then, for each type of exception that your *'throws advice'* is going to handle, you have to define a method with this signature...

```
void AfterThrowing(Exception ex)
```

Basically, your exception handling method has to be named `AfterThrowing`. This name is important... your exception handling method(s) absolutely must be called `AfterThrowing`. If your handler method is not called `AfterThrowing`, then your *'throws advice'* will **never** be called, it's as simple as that. Currently, this naming restriction is not configurable (although it may well be opened up for configuration in the future).

Your exception handling method must (at the very least) declare a parameter that is an `Exception` type... this parameter can be the root `Exception` class (as in the case of the above example), or it can be an `Exception` subclass if you only want to handle certain types of exception. It is good practice to always make your exception handling methods have an `Exception` parameter that is the most specialized `Exception` type possible... i.e. if you are applying *'throws advice'* to a method that could only ever throw `ArgumentException`s, then declare the parameter of your exception handling method as...

```
void AfterThrowing(ArgumentException ex)
```

Note that your exception handling method can have any return type, but returning any value from a Spring.NET *'throws advice'* method would be a waste of time... the Spring.NET AOP infrastructure will simply ignore the return value, so always define the return type of your exception handling methods to be `void`.

Finally, here is the Spring.NET XML configuration for applying the *'throws advice'* declaratively...

```
<object id="throwsAdvice"
  type="Spring.Examples.AopQuickStart.ConsoleLoggingThrowsAdvice"/>

<object id="myServiceObject"
  type="Spring.Aop.Framework.ProxyFactoryObject">
  <property name="target">
    <object id="myServiceObjectTarget"
      type="Spring.Examples.AopQuickStart.ServiceCommand"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>throwsAdvice</value>
    </list>
  </property>
</object>
```


One thing that cannot be done using *'throws advice'* is exception swallowing. It is not possible to define an exception handling method in a *'throws advice'* implementation that will swallow any exception and prevent said exception from bubbling up the call stack. The nearest thing that one can do is define an exception handling method in a *'throws advice'* implementation that will wrap the handled exception in another exception; one would then throw the wrapped exception in the body of one's exception handling method. One can use this to implement some sort of exception translation or exception scrubbing policy, in which implementation specific exceptions (such as `SQLException` or `OracleException` exceptions being thrown by an advised data access object) get replaced with a business exception that has meaning to the service objects in one's business layer. A toy example of this type of *'throws advice'* can be seen below.

```
public class DataAccessExceptionScrubbingThrowsAdvice : IThrowsAdvice
{
    public void AfterThrowing (SQLException ex)
    {
        // business objects in higher level service layer need only deal with PersistenceException...
        throw new PersistenceException ("Cannot access persistent storage.", ex.StackTrace);
    }
}
```

Spring.NET's data access library already has this kind of functionality (and is a whole lot more sophisticated)... the above example is merely being used for illustrative purposes.

This treatment of *'throws advice'*, and of Spring.NET's implementation of it is rather simplistic. *'throws advice'* features that have been omitted include the fact that one can define exception handling methods that permit access to the original object, method, and method arguments of the advised method invocation that threw the original exception. This is a quickstart guide though, and is not meant to be exhaustive... do consult the *'throws advice'* section of the reference documentation, which describes how to declare an exception handling method that gives one access to the above extra objects, and how to declare multiple exception handling methods on the same `IThrowsAdvice` implementation class (see Section 13.3.2.3, "Throws advice").

36.3.1.4. Introductions (mixins)

In a nutshell, introductions are all about adding new state and behaviour to arbitrary objects... transparently and at runtime. Introductions (also called mixins) allow one to emulate multiple inheritance, typically with an eye towards applying crosscutting state and operations to a wide swathe of objects in your application that don't share the same inheritance hierarchy.

36.3.1.5. Layering advice

The examples shown so far have all demonstrated the application of a single advice instance to an advised object. Spring.NET's flavor of AOP would be pretty poor if one could only apply a single advice instance per advised object... it is perfectly valid to apply multiple advice to an advised object. For example, one might apply transactional advice to a service object, and also apply a security access checking advice to that same advised service object.

In the interests of keeping this section lean and tight, let's simply apply *all* of the advice types that have been previously described to a single advised object... in this first instance we'll just use the default pointcut which means that every possible joinpoint will be advised, and you'll be able to see that the various advice instances are applied in order.

Please do consult the class definitions for the following previously defined advice types to see exactly what each advice type implementation does... we're going to be using single instances of the `ConsoleLoggingAroundAdvice`, `ConsoleLoggingBeforeAdvice`, `ConsoleLoggingAfterAdvice`, and `ConsoleLoggingThrowsAdvice` advice to advise a single instance of the `ServiceCommand` class.

You can find the following listing and executable application in the AopQuickStart solution in the project `Spring.AopQuickStart.Step1`.

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
factory.AddAdvice(new ConsoleLoggingAfterAdvice());
factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
factory.AddAdvice(new ConsoleLoggingAroundAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

Here is the Spring.NET XML configuration for declaratively applying multiple advice.

You can find the following listing and executable application in the AopQuickStart solution in the project `Spring.AopQuickStart.Step2`.

```
<object id="throwsAdvice"
  type="Spring.Examples.AopQuickStart.ConsoleLoggingThrowsAdvice"/>
<object id="afterAdvice"
  type="Spring.Examples.AopQuickStart.ConsoleLoggingAfterAdvice"/>
<object id="beforeAdvice"
  type="Spring.Examples.AopQuickStart.ConsoleLoggingBeforeAdvice"/>
<object id="aroundAdvice"
  type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>

<object id="myServiceObject"
  type="Spring.Aop.Framework.ProxyFactoryObject">
  <property name="target">
    <object id="myServiceObjectTarget"
      type="Spring.Examples.AopQuickStart.ServiceCommand"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>throwsAdvice</value>
      <value>afterAdvice</value>
      <value>beforeAdvice</value>
      <value>aroundAdvice</value>
    </list>
  </property>
</object>
```

36.3.1.6. Configuring advice

In case it is not immediately apparent, remember that advice is just a plain old .NET object (a PONO); advice can have constructors that can take any number of parameters, and like any other .NET class, advice can have properties. What this means is that one can leverage the power of the Spring.NET IoC container to apply the IoC principle to one's advice, and in so doing reap all the benefits of Dependency Injection.

Consider the case of throws advice that needs to report (fatal) exceptions to a first line support centre. The throws advice could declare a dependency on a reporting service via a .NET property, and the Spring.NET container could dependency inject the reporting service dependency into the throws advice when it is being created; the reporting dependency might be a simple Log4NET wrapper, or a Windows EventLog wrapper, or a custom reporting exception reporting service that sends detailed emails concerning the fatal exception.

Also bear in mind the fact that Spring.NET's AOP implementation is quite independent of Spring.NET's IoC container. As you have seen, the various examples used in this have illustrated both programmatic and declarative AOP configuration (the latter being illustrated via Spring.NET's IoC XML configuration mechanism).

36.3.2. Using Attributes to define Pointcuts

36.4. The Spring.NET AOP Cookbook

The preceding treatment of Spring.NET AOP has (quite intentionally) been decidedly simple. The overarching aim was to convey the concepts of Spring.NET AOP... this section of the Spring.NET AOP guide contains a number of real world examples of the application of Spring.NET AOP.

36.4.1. Caching

This example illustrates one of the more common usages of AOP... caching.

Lets consider the scenario where we have some static reference data that needs to be kept around for the duration of an application. The data will almost never change over the uptime of an application, and it exists only in the database to satisfy referential integrity amongst the various relations in the database schema. An example of such static (and typically immutable) reference data would be a collection of `Country` objects (comprising a country name and a code). What we would like to do is suck in the collection of `Country` objects and then pin them in a cache. This saves us having to hit the back end database again and again every time we need to reference a country in our application (for example, to populate dropdown controls in a Windows Forms desktop application).

The Data Access Object (DAO) that will load the collection of `Country` objects is called `AdoCountryDao` (it is an implementation of the data-access-technology agnostic DAO interface called `ICountryDao`). The implementation of the `AdoCountryDao` is quite simple, in that every time the `FindAllCountries` instance method is called, an instance will query the database for an `IDataReader` and hydrate zero or more `Country` objects using the returned data.

```
public class AdoCountryDao : ICountryDao
{
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

Ideally, what we would like to have happen is for the results of the *first* call to the `FindAllCountries` instance method to be cached. We would also like to do this in a non-invasive way, because caching is something that we might want to apply at any number of points across the codebase of our application. So, to address what we have identified as a *cross cutting concern*, we can use Spring.NET AOP to implement the caching.

The mechanism that this example is going to use to identify (or pick out) areas in our application that we would like to apply caching to is a .NET `Attribute`. Spring.NET ships with a number of useful custom .NET `Attribute` implementations, one of which is the cunningly named `CacheAttribute`. In the specific case of this example, we are simply going to decorate the definition of the `FindAllCountries` instance method with the `CacheAttribute`.

```
public class AdoCountryDao : ICountryDao
{
    [Cache]
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

The SpringAir reference application that is packaged as part of the Spring.NET distribution comes with a working example of caching applied using Spring.NET AOP (see Chapter 39, *SpringAir - Reference Application*).

36.4.2. Performance Monitoring

This recipe show how easy it is to instrument the classes and objects in an application for performance monitoring. The performance monitoring implementation uses one of the (many) Windows performance counters to display and track the performance data.

36.4.3. Retry Rules

This final recipe describes a simple (but really quite useful) aspect... retry logic. Using Spring.NET AOP, it is quite easy to surround an operation such as a method that opens a connection to a database with a (configurable) aspect that tries to obtain a database connection any number of times in the event of a failure.

36.5. Spring.NET AOP Best Practices

Spring.NET AOP is an 80% AOP solution, in that it only tries to solve the 80% of those cases where AOP is a good fit in a typical enterprise application. This final section of the Spring.NET AOP guide describes where Spring.NET AOP is typically useful (the 80%), as well as where Spring.NET AOP is not a good fit (the 20%).

Chapter 37. Portable Service Abstraction Quick Start

37.1. Introduction

This quickstart demonstrates the basic usage of Spring.NET's portable service abstraction functionality. Sections 2-5 demonstrate the use of .NET Remoting, Section 6 shows the use of the `ServiceComponentExporter` for .NET Enterprise Services, and Section 7 shows the use of the `WebServiceExporter`.



Note

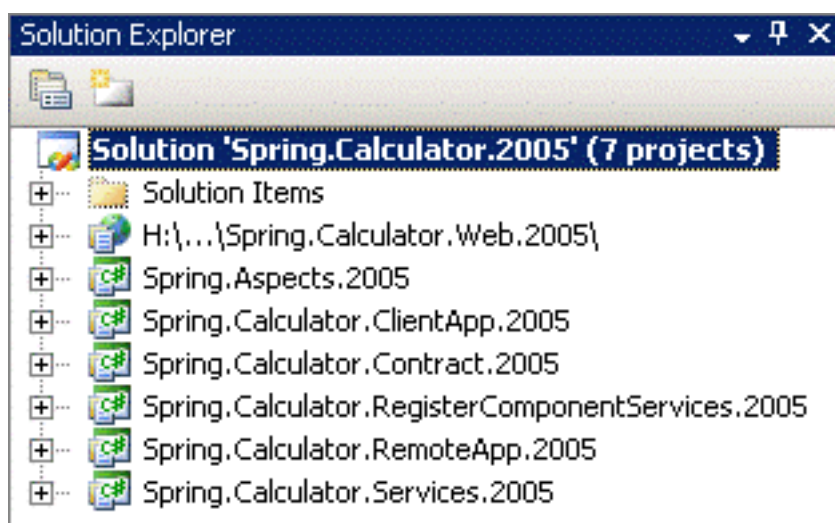
To follow this Quarts QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.Calculator`

37.2. .NET Remoting Example

The infrastructure classes are located in the `Spring.Services` assembly under the `Spring.Services.Remoting` namespace. The overall strategy is to export .NET objects on the server side as either CAO or SAO objects using `CaoExporter` or `SaoExporter` and obtain references to these objects on the client side using `CaoFactoryObject` and `SaoFactoryObject`. This quickstart does assume familiarity with .NET Remoting on the part of the reader. If you are new to .NET remoting you may find the links to introductory remoting material presented at the conclusion of this quickstart of some help.

As usual with quick start examples in Spring.NET, the classes used in the quickstart are intentionally simple. In the specific case of this remoting quickstart we are going to make a simple calculator that can be accessed remotely. The same calculator class will be exported in multiple ways reflecting the variety of .NET remoting options available (CAO, SAO-SingleCall, SAO-Singleton) and also the use of adding AOP advice to SAO hosted objects.

The example solution is located in the `examples\Spring\Spring.Calculator` directory and contains multiple projects.



The `Spring.Calculator.Contract` project contains the interface `ICalculator` that defines the basic operations of a calculator and another interface `IAdvancedCalculator` that adds support for memory

storage for results. (woo hoo - big feature - HP-12C beware!) These interfaces are shown below. The `Spring.Calculator.Services` project contains an implementation of the these interfaces, namely the classes `Calculator` and `AdvancedCalculator`. The purpose of the `AdvancedCalculator` implementation is to demonstrate the configuration of object state for SAO-singleton objects. Note that the calculator implementations *do not* inherit from the `MarshalByRefObject` class. The `Spring.Calculator.ClientApp` project contains the client application and the `Spring.Calculator.RemoteApp` project contains a console application that will host a `Remoted` instance of the `AdvancedCalculator` class. The `Spring.Aspects` project contains some logging advice that will be used to demonstrate the application of aspects to remoted objects. `Spring.Calculator.RegisterComponentServices` is related to enterprise service exporters and is not relevant for this quickstart. `Spring.Calculator.Web` is related to web services exporters and is not relevant for this quickstart.

```
public interface ICalculator
{
    int Add(int n1, int n2);

    int Subtract(int n1, int n2);

    DivisionResult Divide(int n1, int n2);

    int Multiply(int n1, int n2);
}

[Serializable]
public class DivisionResult
{
    private int _quotient = 0;
    private int _rest = 0;

    public int Quotient
    {
        get { return _quotient; }
        set { _quotient = value; }
    }

    public int Rest
    {
        get { return _rest; }
        set { _rest = value; }
    }
}
```

An extension of this interface that supports having a slot for calculator memory is shown below

```
public interface IAdvancedCalculator : ICalculator
{
    int GetMemory();

    void SetMemory(int memoryValue);

    void MemoryClear();

    void MemoryAdd(int num);
}
```

The structure of the VS.NET solution is a consequence of following the best practice of using interfaces to share type information between a .NET remoting client and server. The benefits of this approach are that the client does not need a reference to the assembly that contains the implementation class. Having the client reference the implementation assembly is undesirable for a variety of reasons. One reason being security since an untrusted client could potentially obtain the source code to the implementation since Intermediate Language (IL) code is easily reverse engineered. Another, more compelling, reason is to provide a greater decoupling between the client and server so the server can update its implementation of the interface in a manner that is quite transparent to the client; i.e. the client code need not change. Independent of .NET remoting best practices, using an interface to

provide a service contract is just good object-oriented design. This lets the client choose another implementation unrelated to .NET Remoting, for example a local, test-stub or a web services implementation. One of the major benefits of using Spring.NET is that it reduces the cost of doing 'interface based programming' to almost nothing. As such, this best practice approach to .NET remoting fits naturally into the general approach to application development that Spring.NET encourages you to follow. Ok, with that barrage of OO design ranting finished, on to the implementation!

37.3. Implementation

The implementation of the calculators contained in the `Spring.Calculator.Services` project is quite straightforward. The only interesting methods are those that deal with the memory storage, which is the state that we will be configuring explicitly using constructor injection. A subset of the implementation is shown below.

```
public class Calculator : ICalculator
{
    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }

    public int Subtract(int n1, int n2)
    {
        return n1 - n2;
    }

    public DivisionResult Divide(int n1, int n2)
    {
        DivisionResult result = new DivisionResult();
        result.Quotient = n1 / n2;
        result.Rest = n1 % n2;
        return result;
    }

    public int Multiply(int n1, int n2)
    {
        return n1 * n2;
    }
}

public class AdvancedCalculator : Calculator, IAdvancedCalculator
{
    private int memoryStore = 0;

    public AdvancedCalculator()
    {}

    public AdvancedCalculator(int initialMemory)
    {
        memoryStore = initialMemory;
    }

    public int GetMemory()
    {
        return memoryStore;
    }

    // other methods omitted in this listing...
}
```

The `Spring.Calculator.RemotedApp` project hosts remoted objects inside a console application. The code is also quite simple and shown below

```
public static void Main(string[] args)
```

```

{
    try
    {
        // initialization of Spring.NET's IoC container
        IApplicationContext ctx = ContextRegistry.GetContext();

        Console.Out.WriteLine("Server listening...");
    }
    catch (Exception e)
    {
        Console.Out.WriteLine(e);
    }
    finally
    {
        Console.Out.WriteLine("--- Press <return> to quit ---");
        Console.ReadLine();
    }
}

```

The configuration of the .NET remoting channels is done using the standard `system.runtime.remoting` configuration section inside the .NET configuration file of the application (`App.config`). In this case we are using the `tcp` channel on port 8005.

```

<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp" port="8005" />
    </channels>
  </application>
</system.runtime.remoting>

```

The objects created in Spring's application context are shown below. Multiple resource files are used to export these objects under various remoting configurations. The AOP advice used in this example is a simple Log4Net based around advice.

```

<configSections>
  <sectionGroup name="spring">
    <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
    <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core" />
  </sectionGroup>
  <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
</configSections>

<spring>
  <parsers>
    <parser type="Spring.Remoting.Config.RemotingNamespaceParser, Spring.Services" />
  </parsers>
  <context>
    <resource uri="config://spring/objects" />
    <resource uri="assembly://RemoteServer/RemoteServer.Config/cao.xml" />
    <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleCall.xml" />
    <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleCall-aop.xml" />
    <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleton.xml" />
    <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleton-aop.xml" />
  </context>
  <objects xmlns="http://www.springframework.net">
    <description>Definitions of objects to be exported.</description>

    <object type="Spring.Remoting.RemotingConfigurer, Spring.Services">
      <property name="Filename" value="Spring.Calculator.RemoteApp.exe.config" />
    </object>

    <object id="Log4NetLoggingAroundAdvice" type="Spring.Aspects.Logging.Log4NetLoggingAroundAdvice, Spring.Aspects">
      <property name="Level" value="Debug" />
    </object>

    <object id="singletonCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services">
      <constructor-arg type="int" value="217"/>
    </object>
  </objects>
</spring>

```

```

<object id="singletonCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="target" ref="singletonCalculator"/>
  <property name="interceptorNames">
    <list>
      <value>Log4NetLoggingAroundAdvice</value>
    </list>
  </property>
</object>

<object id="prototypeCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services" singleton="false">
  <constructor-arg type="int" value="217"/>
</object>

<object id="prototypeCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="targetSource">
    <object type="Spring.Aop.Target.PrototypeTargetSource, Spring.Aop">
      <property name="TargetObjectName" value="prototypeCalculator"/>
    </object>
  </property>
  <property name="interceptorNames">
    <list>
      <value>Log4NetLoggingAroundAdvice</value>
    </list>
  </property>
</object>

</objects>
</spring>

```

The declaration of the calculator instance, `singletonCalculator` for example, and the setting of any property values and / or object references is done as you would normally do for any object declared in the Spring.NET configuration file. To expose the calculator objects as .NET remoted objects the exporter `Spring.Remoting.CaoExporter` is used for CAO objects and `Spring.Remoting.SaoExporter` is used for SAO objects. Both exporters require the setting of a `TargetName` property that refers to the name of the object in Spring's IoC container that will be remoted. The semantics of SAO-SingleCall and CAO behavior are achieved by exporting a target object that is declared as a "prototype" (i.e. `singleton=false`). For SAO objects, the `ServiceName` property defines the name of the service as it will appear in the URL that clients use to locate the remote object. To set the remoting lifetime of the objects to be infinite, the property `Infinite` is set to true.

The configuration for the exporting a SAO-Singleton is shown below.

```

<objects
  xmlns="http://www.springframework.net"
  xmlns:r="http://www.springframework.net/remoting">

  <description>Registers the calculator service as a SAO in 'Singleton' mode.</description>

  <r:saoExporter
    targetName="singletonCalculator"
    serviceName="RemotedSaoSingletonCalculator" />
</objects>

```

The configuration shown above uses the Spring Remoting schema but you can also choose to use the standard 'generic' XML configuration shown below.

```

<object name="saoSingletonCalculator" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculator" />
  <property name="ServiceName" value="RemotedSaoSingletonCalculator" />
</object>

```

This will result in the remote object being identified by the URL `tcp://localhost:8005/RemotedSaoSingletonCalculator`. The use of `SaoExporter` and `CaoExporter` for other configuration are similar, look at the configuration files in the `Spring.Calculator.RemotedApp` project files for more information.

On the client side, the client application will connect a specific type of remote calculator service, object, ask it for it's current memory value, which is pre-configured to 217, then perform a simple addition. As in the case of the

server, the channel configuration is done using the standard .NET Remoting configuration section of the .NET application configuration file (App.config), as can be seen below.

```
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp"/>
    </channels>
  </application>
</system.runtime.remoting>
```

The client implementation code is shown below.

```
public static void Main(string[] args)
{
    try
    {
        Pause();

        IApplicationContext ctx = ContextRegistry.GetContext();

        Console.Out.WriteLine("Get Calculator...");
        IAdvancedCalculator firstCalc = (IAdvancedCalculator) ctx.GetObject("calculatorService");
        Console.WriteLine("Divide(11, 2) : " + firstCalc.Divide(11, 2));
        Console.Out.WriteLine("Memory = " + firstCalc.GetMemory());
        firstCalc.MemoryAdd(2);
        Console.Out.WriteLine("Memory + 2 = " + firstCalc.GetMemory());

        Console.Out.WriteLine("Get Calculator...");
        IAdvancedCalculator secondCalc = (IAdvancedCalculator) ctx.GetObject("calculatorService");
        Console.Out.WriteLine("Memory = " + secondCalc.GetMemory());
    }
    catch (Exception e)
    {
        Console.Out.WriteLine(e);
    }
    finally
    {
        Pause();
    }
}
```

Note that the client application code is not aware that it is using a remote object. The `Pause()` method simply waits until the `Return` key is pressed on the console so that the client doesn't make a request to the server before the server has had a chance to start. The standard configuration and initialization of the .NET remoting infrastructure is done before the creation of the Spring.NET IoC container. The configuration of the client application is constructed in such a way that one can easily switch implementations of the `calculatorService` retrieved from the application context. In more complex applications the calculator service would be a dependency on another object in your application, say in a workflow processing layer. The following listing shows a configuration for use of a local implementation and then several remote implementations. The same `Exporter` approach can be used to create Web Services and Serviced Components (Enterprise Services) of the calculator object but are not discussed in this QuickStart.

```
<spring>
  <context>
    <resource uri="config://spring/objects" />

    <!-- Only one at a time ! -->

    <!-- ===== -->
    <!-- In process (local) implementations -->
    <!-- ===== -->
    <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.InProcess/inProcess.xml" />

    <!-- ===== -->
    <!-- Remoting implementations -->
```

```

<!-- ===== -->
<!-- Make sure 'RemoteApp' console application is running and listening. -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.Remoting/cao.xml" /> -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.Remoting/cao-ctor.xml" /> -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.Remoting/saoSingleton.xml" /> -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.Remoting/saoSingleton-aop.xml" /> -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.Remoting/saoSingleCall.xml" /> -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.Remoting/saoSingleCall-aop.xml" /> -->

<!-- ===== -->
<!-- Web Service implementations -->
<!-- ===== -->
<!-- Make sure 'http://localhost/SpringCalculator/' web application is running -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.WebServices/webServices.xml" /> -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.WebServices/webServices-aop.xml" /> -->

<!-- ===== -->
<!-- EnterpriseService implementations -->
<!-- ===== -->
<!-- Make sure you register components with 'RegisterComponentServices' console application. -->
<!-- <resource uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientApp.Config.EnterpriseServices/enterpriseServices.xml" /> -->

</context>
</spring>

```

The `inProcess.xml` configuration file creates an instance of `AdvancedCalculator` directly

```

<objects xmlns="http://www.springframework.net">

    <description>inProcess</description>

    <object id="calculatorService" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services" />

</objects>

```

Factory classes are used to create a client side reference to the .NET remoting implementations. For SAO objects use the `SaoFactoryObject` class and for CAO objects use the `CaoFactoryObject` class. The configuration for obtaining a reference to the previously exported SAO singleton implementation is shown below

```

<objects xmlns="http://www.springframework.net">

    <description>saoSingleton</description>

    <object id="calculatorService" type="Spring.Remoting.SaoFactoryObject, Spring.Services">
        <property name="ServiceInterface" value="Spring.Calculator.Interfaces.IAdvancedCalculator, Spring.Calculator.Contract" />
        <property name="ServiceUrl" value="tcp://localhost:8005/RemotedSaoSingletonCalculator" />
    </object>

</objects>

```

You must specify the property `ServiceInterface` as well as the location of the remote object via the `ServiceUrl` property. The property replacement facilities of Spring.NET can be leveraged here to make it easy to configure the URL value based on environment variable settings, a standard .NET configuration section, or an external property file. This is useful to easily switch between test, QA, and production (yea baby!) environments. An example of how this would be expressed is...

```

<property name="ServiceUrl" value="${protocol}://${host}:${port}/RemotedSaoSingletonCalculator" />

```

The property values in this example are defined elsewhere; refer to Section 5.9.2.1, “Example: The `PropertyPlaceholderConfigurer`” for additional information. As mentioned previously, more important in terms of configuration flexibility is the fact that now you can swap out different implementations (.NET remoting based or otherwise) of this interface by making a simple change to the configuration file.

The configuration for obtaining a reference to the previously exported CAO implementation is shown below

```

<objects xmlns="http://www.springframework.net">

```

```

<description>cao</description>

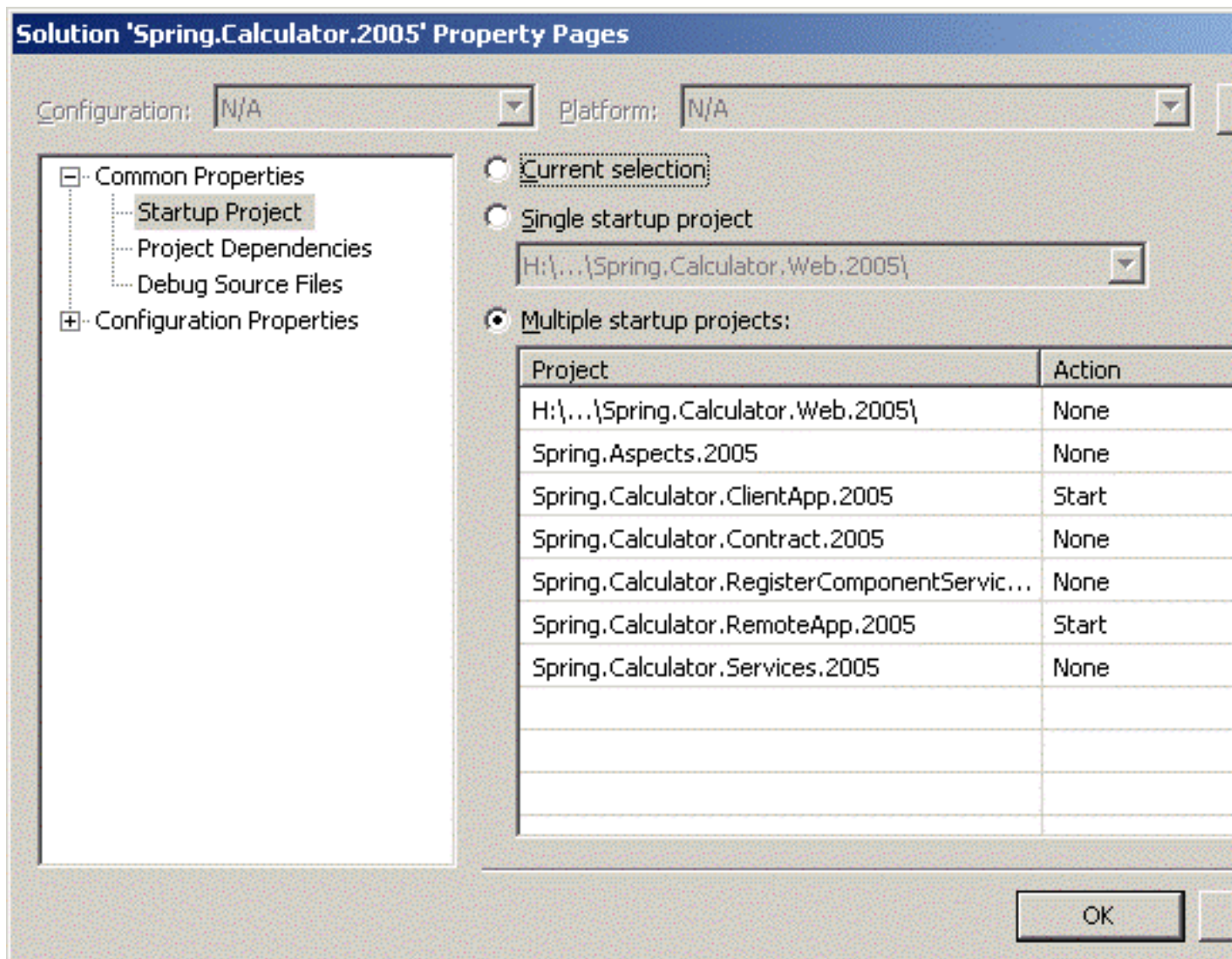
<object id="calculatorService" type="Spring.Remoting.CaoFactoryObject, Spring.Services">
  <property name="RemoteTargetName" value="prototypeCalculator" />
  <property name="ServiceUrl" value="tcp://localhost:8005" />
</object>

</objects>

```

37.4. Running the application

Now that we have had a walk through of the implementation and configuration it is finally time to run the application (if you haven't yet pulled the trigger). Be sure to set up VS.NET to run multiple applications on startup as shown below.



Running the solution yields the following output in the server and client window

SERVER WINDOW

```

Server listening...
--- Press <return> to quit ---

```

CLIENT WINDOW

```

--- Press <return> to continue ---      (hit return...)
Get Calculator...
Divide(11, 2) : Quotient: '5'; Rest: '1'
Memory = 0
Memory + 2 = 2
Get Calculator...
Memory = 2
--- Press <return> to continue ---

```

37.5. Remoting Schema

The spring-remoting.xsd file in the doc directory provides a short syntax to configure Spring.NET remoting features. To install the schema in the VS.NET environment run the install-schema NAnt script in the doc directory. Refer to the Chapter on VS.NET integration for more details.

The various configuration files in the RemoteServer and Client projects show the schema in action. Here is a condensed listing of those definitions which should give you a good feel for how to use the schema.

```

<!-- Calculator definitions -->
<object id="singletonCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services">
  <constructor-arg type="int" value="217" />
</object>

<object id="prototypeCalculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services" singleton=
  <constructor-arg type="int" value="217" />
</object>

<!-- CAO object -->
<r:caoExporter targetName="prototypeCalculator" infinite="false">
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m"/>
</r:caoExporter>

<!-- SAO Single Call -->
<r:saoExporter
  targetName="prototypeCalculator"
  serviceName="RemotedSaoSingleCallCalculator"/>

<!-- SAO Singleton -->
<r:saoExporter
  targetName="singletonCalculator"
  serviceName="RemotedSaoSingletonCalculator" />

```

Note that the singleton nature of the remoted object is based on the Spring object definition. The "PrototypeCalculator" has its singleton property set to false so that a new one will be created every time a method on the remoted object is invoked for the SAO case.

37.6. .NET Enterprise Services Example

The .NET Enterprise Services example is located in the project Spring.Calculator.RegisterComponentServices.2005.csproj or Spring.Calculator.RegisterComponentServices.2003.csproj, depending on the use of .NET 1.1 or 2.0. The example uses the previous AdvancedCalculator implementation and then imports the embedded configuration file 'enterpriseServices.xml' from the namespace Spring.Calculator.RegisterComponentServices.Config. The top level configuration is shown below

```

<spring>

  <context>
    <resource uri="config://spring/objects" />
    <resource uri="assembly://Spring.Calculator.RegisterComponentServices/Spring.Calculator.RegisterComponentServices.Config/" />
  </context>

  <objects xmlns="http://www.springframework.net">

```

```

<description>Definitions of objects to be registered.</description>

<object id="calculatorService" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services" />

</objects>

</spring>

```

The exporter that adapts the AdvancedCalculator for use as an Enterprise Service component is defined first in enterpriseServices.xml. Second is defined an exporter that will host the exported Enterprise Services component application by signing the assembly, registering it with the specified COM+ application name. If application does not exist it will create it and configure it using values specified for Description, AccessControl and Roles properties. The configuration file for enterpriseServices.xml is shown below

```

<objects xmlns="http://www.springframework.net">

  <description>enterpriseService</description>

  <object id="calculatorComponent" type="Spring.EnterpriseServices.ServicedComponentExporter, Spring.Services">
    <property name="TargetName" value="calculatorService" />
    <property name="TypeAttributes">
      <list>
        <object type="System.EnterpriseServices.TransactionAttribute, System.EnterpriseServices" />
      </list>
    </property>
    <property name="MemberAttributes">
      <dictionary>
        <entry key="*">
          <list>
            <object type="System.EnterpriseServices.AutoCompleteAttribute, System.EnterpriseServices" />
          </list>
        </entry>
      </dictionary>
    </property>
  </object>

  <object type="Spring.EnterpriseServices.EnterpriseServicesExporter, Spring.Services">
    <property name="ApplicationName">
      <value>Spring Calculator Application</value>
    </property>
    <property name="Description">
      <value>Spring Calculator application.</value>
    </property>
    <property name="AccessControl">
      <object type="System.EnterpriseServices.ApplicationAccessControlAttribute, System.EnterpriseServices">
        <property name="AccessChecksLevel">
          <value>ApplicationComponent</value>
        </property>
      </object>
    </property>
    <property name="Roles">
      <list>
        <value>Admin : Administrator role</value>
        <value>User : User role</value>
        <value>Manager : Administrator role</value>
      </list>
    </property>
    <property name="Components">
      <list>
        <ref object="calculatorComponent" />
      </list>
    </property>
    <property name="Assembly">
      <value>Spring.Calculator.EnterpriseServices</value>
    </property>
  </object>

</objects>

```

37.7. Web Services Example

The WebServices example shows how to export the AdvancedCalculator as a web service that is an AOP proxy of AdvancedCalculator that has logging advice applied to it. The main configuration file, Web.config, includes information from three locations as shown below

```
<context>
  <resource uri="config://spring/objects" />
  <resource uri="~/Config/webServices.xml" />
  <resource uri="~/Config/webServices-aop.xml" />
</context>
```

The config section 'spring/objects' in Web.config contains the definition for the 'plain' Advanced calculator, as well as the definitions to create an AOP proxy of an AdvancedCalculator that adds logging advice. These definitions are shown below

```
<objects xmlns="http://www.springframework.net">

  <!-- Aspect -->

  <object id="CommonLoggingAroundAdvice" type="Spring.Aspects.Logging.CommonLoggingAroundAdvice, Spring.Aspects">
    <property name="Level" value="Debug" />
  </object>

  <!-- Service -->

  <!-- 'plain object' for AdvancedCalculator -->
  <object id="calculator" type="Spring.Calculator.Services.AdvancedCalculator, Spring.Calculator.Services"/>

  <!-- AdvancedCalculator object with AOP logging advice applied. -->
  <object id="calculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
    <property name="target" ref="calculator" />
    <property name="interceptorNames">
      <list>
        <value>CommonLoggingAroundAdvice</value>
      </list>
    </property>
  </object>

</objects>
```

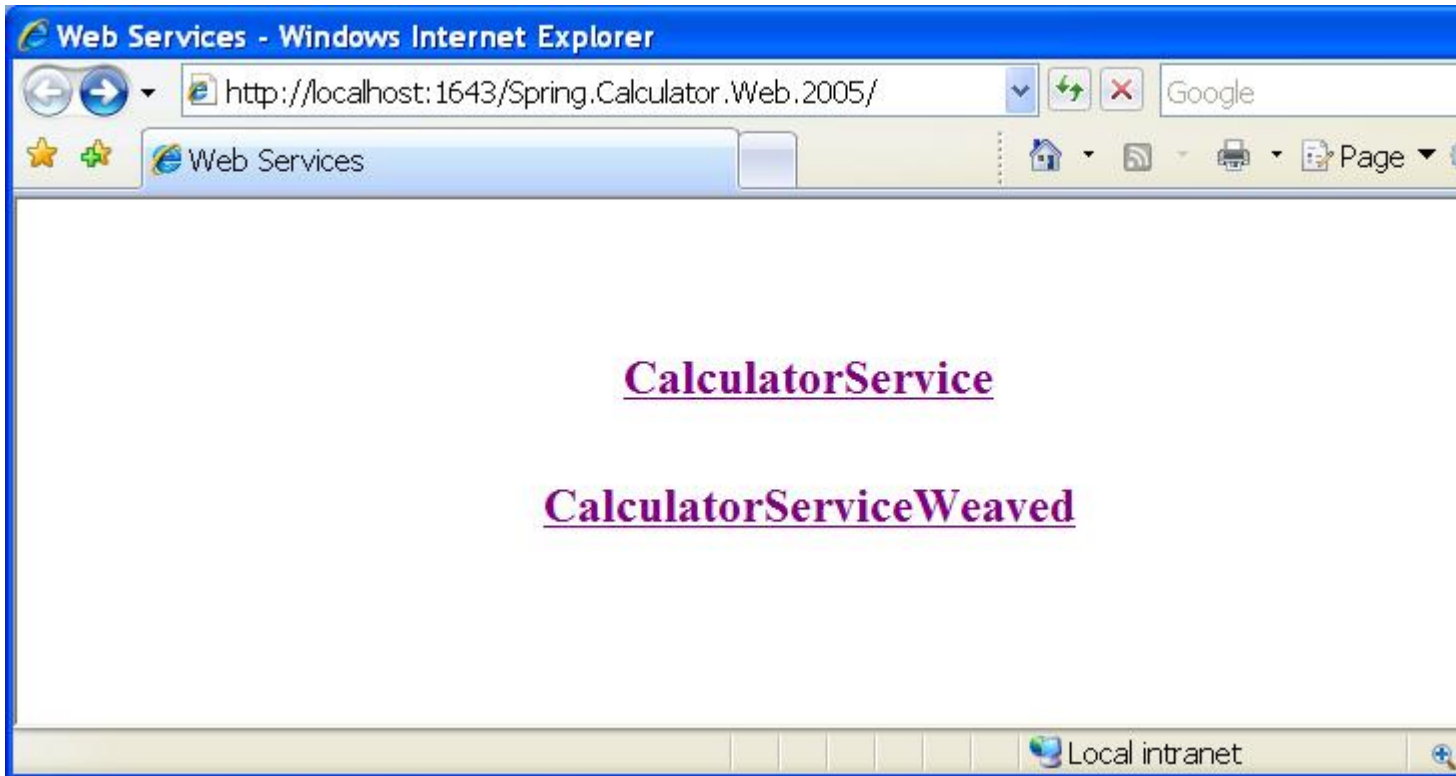
The configuration file webService.xml simply exports the named calculator object

```
<object id="calculatorService" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="calculator" />
  <property name="Namespace" value="http://SpringCalculator/WebServices" />
  <property name="Description" value="Spring Calculator Web Services" />
</object>
```

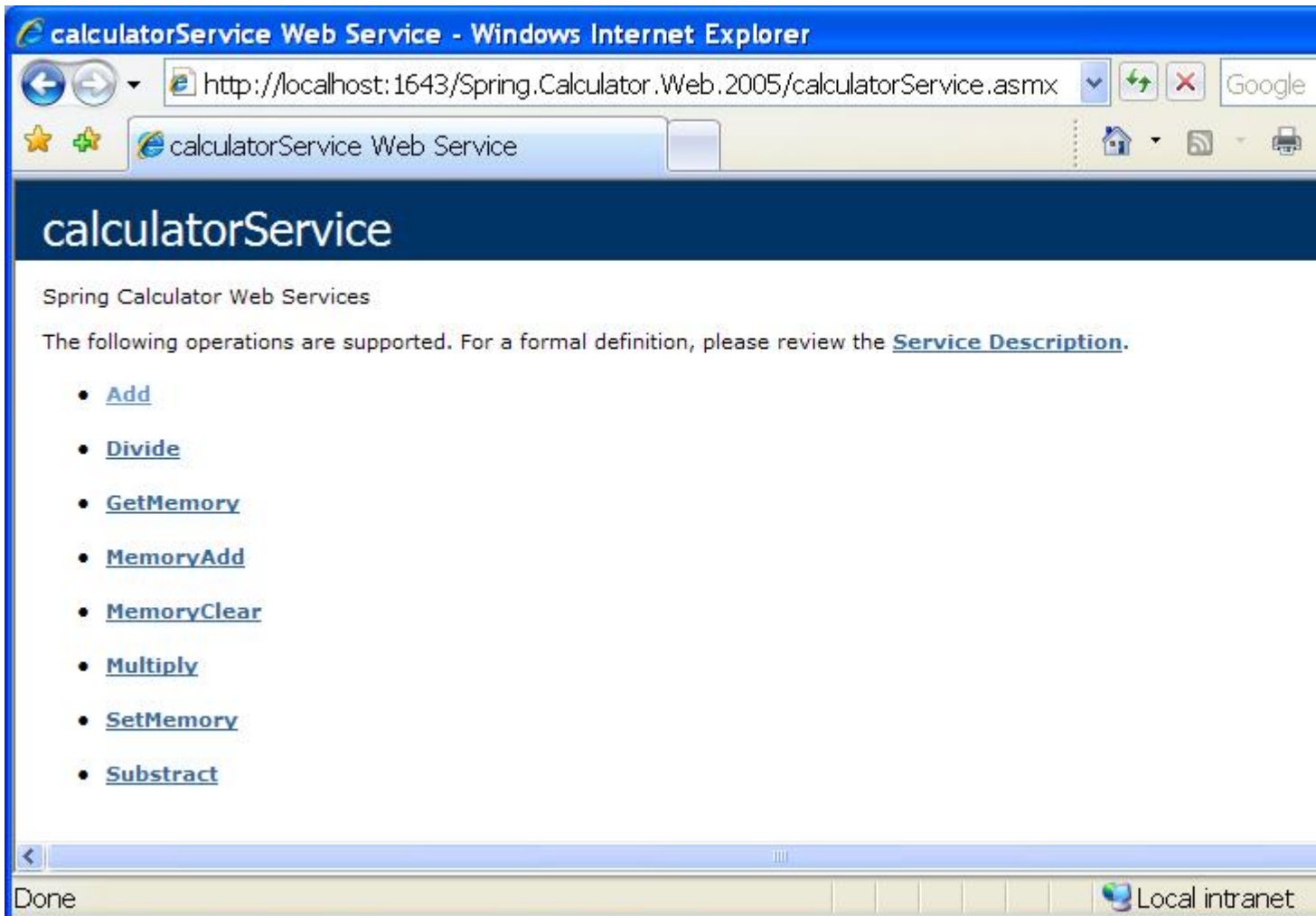
Whereas the webService-aop.xml exports the calculator instance that has AOP advice applied to it.

```
<object id="calculatorServiceWeaved" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="calculatorWeaved" />
  <property name="Namespace" value="http://SpringCalculator/WebServices" />
  <property name="Description" value="Spring Calculator Web Services" />
</object>
```

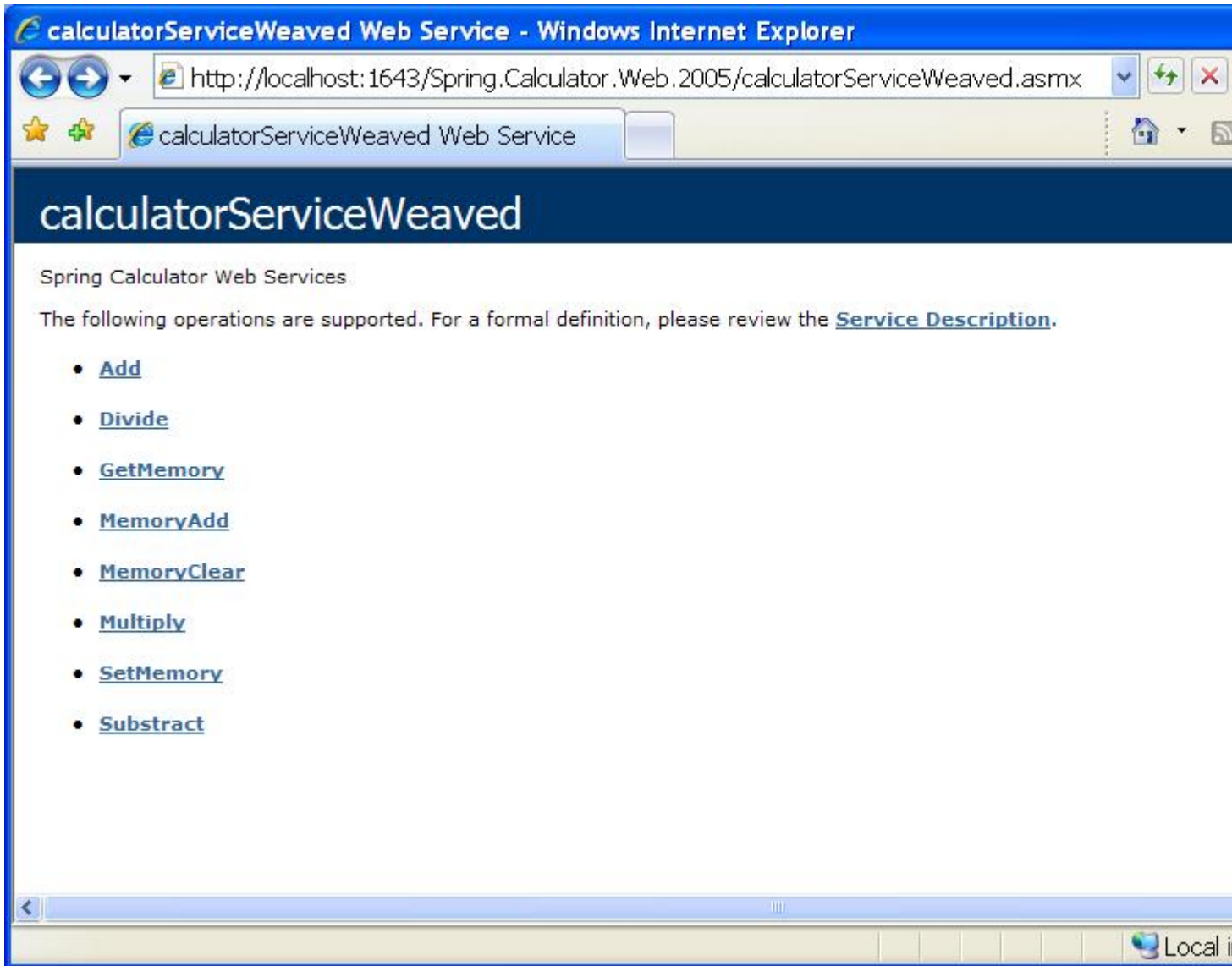
Setting the solution to run the web project as the startup, you will be presented with a screen as shown below



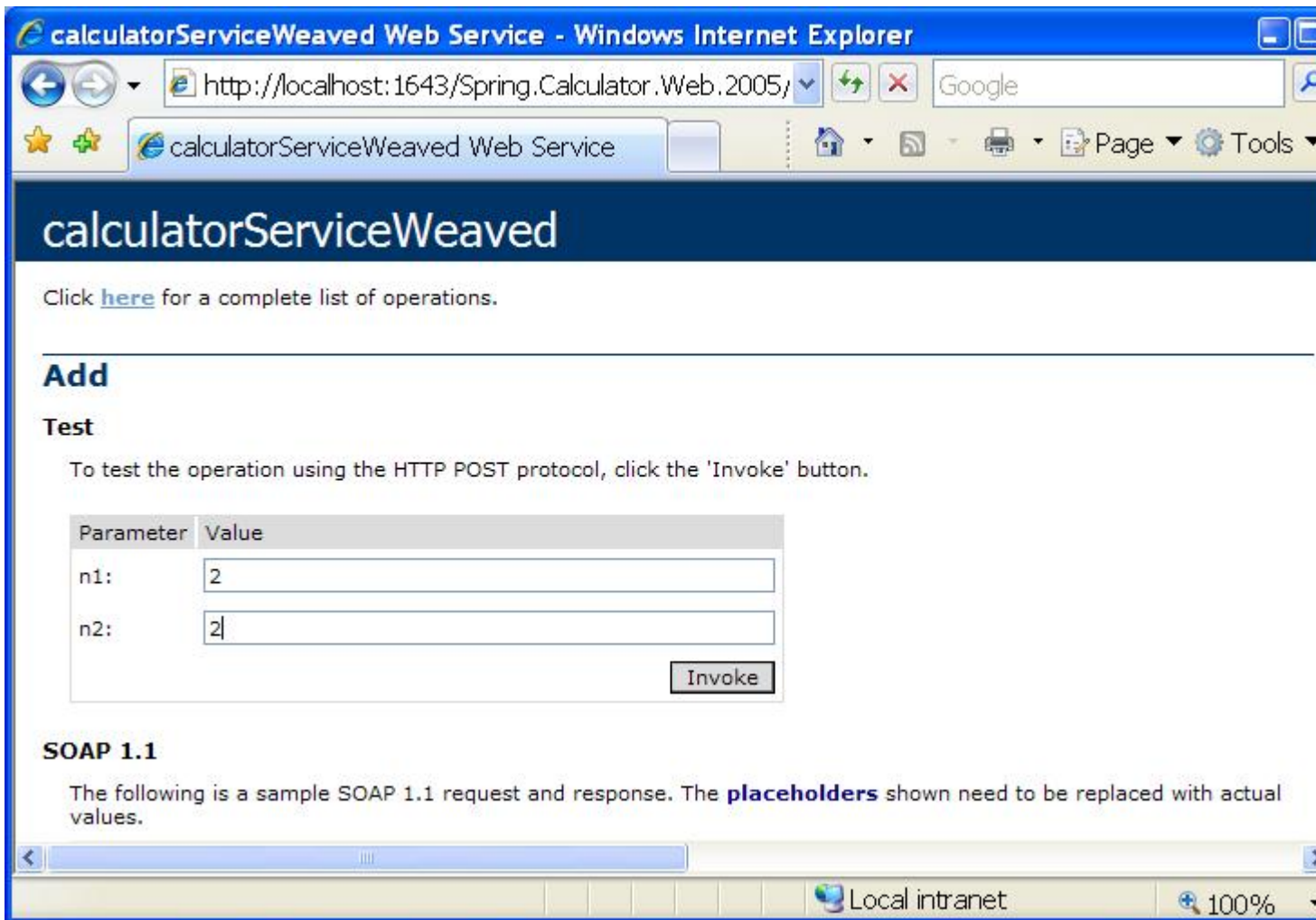
Selecting the `CalculatorService` and `CalculatorServiceWeaved` links will bring you to the standard user interface generated for browsing a web service, as shown below



And similarly for the calculator service with AOP applied



Invoking the Add method for calculatorServiceWeaved shows the screen



Invoking add will then show the result '4' in a new browser instance and the log file log.txt will contain the following entries

```
2007-10-15 17:59:47,375 [DEBUG] Spring.Aspects.Logging.CommonLoggingAroundAdvice - Intercepted call : about to invoke method
2007-10-15 17:59:47,421 [DEBUG] Spring.Aspects.Logging.CommonLoggingAroundAdvice - Intercepted call : returned '4'
```

37.8. Additional Resources

Some introductory articles on .NET remoting can be found online at MSDN. Ingo Rammer is also a very good authority on .NET remoting, and the .NET Remoting FAQ (link below) which is maintained by Ingo is chock full of useful information.

- [An Introduction to Microsoft .NET Remoting Framework](#)
- [Microsoft .NET Remoting: A Technical Overview](#)
- [Advanced .NET Remoting](#) (authored by Ingo Rammer)
- [.NET Remoting FAQ](#)

Chapter 38. Web Quickstarts

38.1. Introduction

The Web Quickstart solution provides basic 'Hello World' examples for using Spring.Web features. You can use this solution as a starting point and then move on to the SpringAir application that uses a wider range of Spring.Web features. The documentation inside the solution and web application itself can help guide you through the functionality.



Note

To follow this Quarts QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.WebQuickStart`

Chapter 39. SpringAir - Reference Application

39.1. Introduction

The SpringAir sample application demonstrates a selection of Spring.NET's powerful features making a .NET programmer's life easier. It demonstrates the following features of Spring.Web

- Spring.NET IoC container configuration
- Dependency Injection as applied to ASP.NET pages
- Master Page support
- Web Service support
- Bi-directional data binding
- Declarative validation of domain objects
- Internationalization
- Result mapping to better encapsulate page navigation flows

The application models a flight reservation system where you can browse flights, book a trip and even attach your own clients by leveraging the web services exposed by the SpringAir application.

All pages within the application are fully Spring managed. Dependencies get injected as configured within a Spring Application Context. For NET 1.1 it shows how to apply centrally managed layouts to all pages in an application by using master pages - a well-known feature from NET 2.0.

When selecting your flights, you are already experiencing a fully localized form. Select your preferred language from the bottom of the form and see, how the new language is immediately applied. As soon as you submit your desired flight, the submitted values are automatically unbound from the form onto the application's data model by leveraging Spring.Web's support for Data Binding. With Data Binding you can easily associate properties on your PONO model with elements on your ASP.NET form.

39.2. Getting Started

The application is located in the installation directory under 'examples/SpringAir'. The directory 'SpringAir.Web.2003' contains the .NET 1.1 version of the application and the directory 'SpringAir.Web.2005' contains the .NET 2.0 version. For .NET 1.1 you will need to create a virtual directory named 'SpringAir.2003' using IIS Administrator and point it to the following directory examples\Spring\SpringAir\src\SpringAir.Web.2003. The solution file for .NET 1.1 is examples\Spring\SpringAir\SpringAir.2003.sln. For .NET 2.0 simply open the solution examples\Spring\SpringAir\SpringAir.2005.sln. Set your startup project to be SpringAir.Web and the startup page to .\Web\Home.aspx

39.3. Container configuration

The web project's top level Web.config configures the IoC container that is used within the web application. You do not need to explicitly instantiate the IoC container. The important parts of that configuration are shown below

```

<spring>
  <parsers>
    <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
  </parsers>

  <context>
    <resource uri="~/Config/Aspects.xml"/>
    <resource uri="~/Config/Web.xml"/>
    <resource uri="~/Config/Services.xml"/>

    <!-- TEST CONFIGURATION -->

    <resource uri="~/Config/Test/Services.xml"/>
    <resource uri="~/Config/Test/Dao.xml"/>

    <!-- PRODUCTION CONFIGURATION -->

    <!--
    <resource uri="~/Config/Production/Services.xml"/>
    <resource uri="~/Config/Production/Dao.xml"/>
    -->
  </context>
</spring>

```

In this example there are separate configuration files for test and production configuration. The Services.xml file is in fact the same between the two, and the example will be refactored in future to remove that duplication. The Dao layer in the test configuration is an in-memory version, faking database access, whereas the production version uses an ADO.NET based solution.

The pages that comprise the application are located in the directory 'Web/BookTrip'. In that directory is another Web.config that is responsible for configuring that directory's .aspx pages. There are three main pages in the flow of the application.

- TripForm - form to enter in airports, times, round-trip or one-way
- Suggested Flights - form to select flights
- ReservationConfirmationPage - your confirmation ID from the booking process.

The XML configuration to configure the TripForm form is shown below

```

<object type="TripForm.aspx" parent="standardPage">
  <property name="BookingAgent" ref="bookingAgent" />
  <property name="AirportDao" ref="airportDao" />
  <property name="TripValidator" ref="tripValidator" />
  <property name="Results">
    <dictionary>
      <entry key="displaySuggestedFlights" value="redirect:SuggestedFlights.aspx" />
    </dictionary>
  </property>
</object>

```

As you can see the various services it needs are set using standard DI techniques. The Results property externalizes the page flow, redirecting to the next page in the flow, SuggestedFlights. The 'parent' attribute lets this page inherit properties from a template. The is located in the top level Web.config file, packaged under the Config directory. The standardPage sets up properties of Spring's base page class, from which all the pages in this application inherit from. (Note that to perform only dependency injection on pages you do not need to inherit from Spring's Page class).

39.4. Bi-directional data binding

The TripForm page demonstrates the bi-directional data binding features. A Trip object is used to back the information of the form. The family of methods that are overridden to support the bi-directional data binding are listed below.

```
protected override void InitializeModel()
{
    trip = new Trip();
    trip.Mode = TripMode.RoundTrip;
    trip.StartingFrom.Date = DateTime.Today;
    trip.ReturningFrom.Date = DateTime.Today.AddDays(1);
}

protected override void LoadModel(object savedModel)
{
    trip = (Trip)savedModel;
}

protected override object SaveModel()
{
    return trip;
}

protected override void InitializeDataBindings()
{
    BindingManager.AddBinding("tripMode.Value", "Trip.Mode");
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue", "Trip.StartingFrom.AirportCode");
    BindingManager.AddBinding("goingToAirportCode.SelectedValue", "Trip.ReturningFrom.AirportCode");
    BindingManager.AddBinding("leavingFromDate.SelectedDate", "Trip.StartingFrom.Date");
    BindingManager.AddBinding("returningOnDate.SelectedDate", "Trip.ReturningFrom.Date");
}
```

This is all you need to set up in order to have values from the Trip object 'marshaled' to and from the web controls. The InitializeDataBindings method set this up, using the Spring Expression Language to define the UI element property that is associate with the model (Trip) property.

39.5. Declarative Validation

The method called when the Search button is clicked will perform validation. If validation succeeds as well as additional business logic checks, the next page in the flow is loaded. This is shown in the code below. Notice how much cleaner and more business focused the code reads than if you were using standard ASP.NET APIs.

```
protected void SearchForFlights(object sender, EventArgs e)
{
    if (Validate(trip, tripValidator))
    {
        FlightSuggestions suggestions = this.bookingAgent.SuggestFlights(trip);
        if (suggestions.HasOutboundFlights)
        {
            Session[Constants.SuggestedFlightsKey] = suggestions;
            SetResult(DisplaySuggestedFlights);
        }
    }
}
```

The 'Validate' method of the page takes as arguments the object to validate and a IValidator instance. The TripForm property TripValidator is set via dependency injection (as shown above). The validation logic is defined declaratively in the XML configuration file and is shown below.

```
<v:group id="tripValidator">

    <v:required id="departureAirportValidator" test="StartingFrom.AirportCode">
        <v:message id="error.departureAirport.required" providers="departureAirportErrors, validationSummary"/>
    </v:required>
```

```

<v:group id="destinationAirportValidator">
  <v:required test="ReturningFrom.AirportCode">
    <v:message id="error.destinationAirport.required" providers="destinationAirportErrors, validationSummary"/>
  </v:required>
  <v:condition test="ReturningFrom.AirportCode != StartingFrom.AirportCode" when="ReturningFrom.AirportCode != ''">
    <v:message id="error.destinationAirport.sameAsDeparture" providers="destinationAirportErrors, validationSummary"/>
  </v:condition>
</v:group>

<v:group id="departureDateValidator">
  <v:required test="StartingFrom.Date">
    <v:message id="error.departureDate.required" providers="departureDateErrors, validationSummary"/>
  </v:required>
  <v:condition test="StartingFrom.Date >= DateTime.Today" when="StartingFrom.Date != DateTime.MinValue">
    <v:message id="error.departureDate.inThePast" providers="departureDateErrors, validationSummary"/>
  </v:condition>
</v:group>

<v:group id="returnDateValidator" when="Mode == 'RoundTrip'">
  <v:required test="ReturningFrom.Date">
    <v:message id="error.returnDate.required" providers="returnDateErrors, validationSummary"/>
  </v:required>
  <v:condition test="ReturningFrom.Date >= StartingFrom.Date" when="ReturningFrom.Date != DateTime.MinValue">
    <v:message id="error.returnDate.beforeDeparture" providers="returnDateErrors, validationSummary"/>
  </v:condition>
</v:group>

</v:group>

```

The validation logic has 'when' clauses so that return dates can be ignored if the Mode property of the Trip object is set to 'RoundTrip'.

39.6. Internationalization

Both image and text based internationalization are supported. You can see this in action by clicking on the English, Srpski, or ##### links on the bottom of the page.

39.7. Web Services

The class BookingAgent that was used by the TripForm class is a standard .NET class, i.e no WebMethod attributes are on any of its methods. Spring can expose this object as a web service by declaring the following XML defined in the top level Config/Services.xml file

```

<object id="bookingAgentWebService" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
  <property name="TargetName" value="bookingAgent"/>
  <property name="Name" value="BookingAgent"/>
  <property name="Namespace" value="http://SpringAir/WebServices"/>
  <property name="Description" value="SpringAir Booking Agent Web Service"/>
  <property name="MemberAttributes">
    <dictionary>
      <entry key="SuggestFlights">
        <object type="System.Web.Services.WebMethodAttribute, System.Web.Services">
          <property name="Description" value="Gets those flight suggestions that are applicable for the supplied trip."/>
        </object>
      </entry>
      <entry key="Book">
        <object type="System.Web.Services.WebMethodAttribute, System.Web.Services">
          <property name="Description" value="Goes ahead and actually books what up until this point has been a transient">
        </object>
      </entry>
      <entry key="GetAirportList">
        <object type="System.Web.Services.WebMethodAttribute, System.Web.Services">
          <property name="Description" value="Return a collection of all those airports that can be used for the purposes">
        </object>
      </entry>
    </dictionary>
  </property>
</object>

```

```
</property>  
</object>
```

Chapter 40. Data Access QuickStart

40.1. Introduction

The data access quick start demonstrates the API usage of AdoTemplate (both generic and non-generic versions) as well as the use of the object based data access classes contained in Spring.Data.Objects. It uses the Northwind database and is located under the directory examples/DataAccessQuickStart.

The quick start contains pseudo DAO objects and a collection of NUnit tests to exercise them rather than a full blown application. To run the tests from within VS.NET install [TestDriven.NET](#), [ReSharper](#), or an equivalent . The listing of DAO classes and the parts of Spring.Data that they demonstrate is shown below.

- CommandCallbackDao - Use of the ICommandCallback and CommandCallbackDelegate
- ResultSetExtractorDao - Use of IResultSetExtractor and ResultSetExtractorDelegate
- RowCallbackDao - Use of IRowCallback and RowCallbackDelegate
- RowMapperDao - Use of IRowMapper and RowMapperDelegate
- QueryForObject - Use of QueryForObject method.
- StoredProcDao - Use of Spring.Data.Objects.StoredProcedure

The are simple domain objects in the Spring.DataQuickStart.Domain namespace, collections of which are generally returned from the DAO methods.



Note

To follow this Data Access QuickStart load the solution file found in the directory <spring-install-dir>\examples\Spring\Spring.DataQuickStart

40.1.1. Database configuration

To get started running the 'unit test' you should configure the database connection string. The listing in DataQuickStart.GenericTemplate.ExampleTests.xml is shown below

```
<objects xmlns="http://www.springframework.net"
  xmlns:db="http://www.springframework.net/database">

  <db:provider id="dbProvider"
    provider="SqlServer-1.1"
    connectionString="Data Source=(local);Database=Northwind;User ID=springqa;Password=springqa;Trusted_Connection=

  <!-- other definitions not shown

</objects>
```

You should change the value of the provider element to correspond to you database and the connection string as appropriate. Please refer to the documentation on the DbProvider abstraction for details particular to your database configuration. You should also install the Northwind database, which is available for SqlServer 2005 from this [download location](#). The minimal schema to support other database providers may be supported in the future.

40.1.1.1. AdoTemplate Configuration

The various DAO objects refer to an instance of AdoTemplate which is responsible for performing data access operations. This is declared in ExampleTest.xml as shown below

```
<object id="adoTemplate" type="Spring.Data.Generic.AdoTemplate, Spring.Data">
  <property name="DbProvider" ref="dbProvider"/>
  <property name="DataReaderWrapperType" value="Spring.Data.Support.NullMappingDataReader, Spring.Data"/>
</object>
```

The property DbProvider refers to the database configuration you previously defined. Also the property DataReaderWrapper is set to the NullMappingDataReader that ships with Spring. This provides convenient default values for null values returned from the database. To read more about AdoTemplate, refer to the chapter, Data access using ADO.NET.

40.1.2. CommandCallback

The code that exercises the use of a CommandCallback is shown below

```
[Test]
public void CallbackDaoTest()
{
    CommandCallbackDao commandCallbackDao = ctx["commandCallbackDao"] as CommandCallbackDao;
    int count = commandCallbackDao.FindCountWithPostalCode("1010");
    Assert.AreEqual(3, count);
}
```

The configuration of the CommandCallbackDao is shown below

```
<object id="commandCallbackDao" type="Spring.DataQuickStart.Dao.GenericTemplate.CommandCallbackDao, Spring.DataQuickStart">
  <property name="AdoTemplate" ref="adoTemplate"/>
</object>
```

This is the minimal configuration required for a DAO object, typically DAO objects in your application will include other configuration information, for example properties to specify the maximum size of the result set returned etc. The implementation of the FindCountWithPostalCode is shown below

```
public virtual int FindCountWithPostalCodeWithDelegate(string postalCode)
{
    // Using anonymous delegates allows you to easily reference the
    // surrounding parameters for use with the DbCommand processing.

    return AdoTemplate.Execute<int>(delegate(DbCommand command)
    {
        // Do whatever you like with the DbCommand... downcast to get
        // provider specific functionality if necessary.

        command.CommandText = cmdText;

        DbParameter p = command.CreateParameter();
        p.ParameterName = "@PostalCode";
        p.Value = postalCode;
        command.Parameters.Add(p);

        return (int)command.ExecuteScalar();
    });
}
```

Anonymous delegates are used to specify the implementation of the callback function that passes in a DbCommand object. You can then use the DbCommand object as you see fit to access the database. If you are using Spring's declarative transaction management features then this DbCommand would have its transaction and connection properties based on the context of the surrounding transaction. All resource management for the

DbCommand are handled for you by the framework, as well as error reporting on error etc. If you execute the test, it will pass, assuming you haven't modified any data in the Northwind database from its raw installation.

Chapter 41. Transactions QuickStart

41.1. Introduction

The Transaction Quickstart demonstrates Spring's transaction management features. The database schema are two simple tables, credit and debit, which contain an Identifier and an Amount. The quick start shows the use of declarative transactions using attributes and also the ability to change the transaction manager (local or distributed) via changes to only the configuration files - no code changes are required. It also demonstrates some techniques for unit and integration testing an application as well as separating Spring's configuration files so that one is responsible for describing how the core business classes are configured and others that are responsible for the database environment and application of AOP.

This quickstart assumes you have installed a way to run NUnit tests within your IDE. Some excellent tools that let you do this are [TestDriven.NET](#) and [ReSharper](#).



Note

To follow this Quarts QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.TxQuickStart`

41.2. Application Overview

The design of the application is very simple and consists of two logical layers, a business service layer in the namespace `Spring.TxQuickStart.Services` and a DAO layer in the namespace `Spring.TxQuickStart.Dao`. As this is just a toy example the business service layer does nothing more than call two DAO objects. The business service is to transfer money in a bank account and is blatantly taken from the book [Pro ADO.NET](#) by Sahil Malik. The transfer service is defined by the interface `IAccountManager` with the implementation `AccountManager` located in the namespace `Spring.TxQuickStart.Services`. The money is recorded in a credit and debit table in the database. The SQL Server schema for the tables is located in the file `CreditsDebitsSchema.sql`. Transferring the money requires an ACID operation on these two tables. The credit operation is defined via a `IAccountCreditDao` interface and the debit operation via an `IAccountDebitDao` interface. Implementations of these interfaces using `AdoTemplate` are in the namespace `Spring.TxQuickStart.Dao.Ado`.

41.2.1. Interfaces

The Manager and DAO interfaces are shown below

```
public interface IAccountManager
{
    void DoTransfer(float creditAmount, float debitAmount);
}

public interface IAccountCreditDao
{
    void CreateCredit(float creditAmount);
}

public interface IAccountDebitDao
{
    void DebitAccount(float debitAmount);
}
```

41.3. Implementation

The implementation of the Account Credit DAO is shown below

```
public class AccountCreditDao : AdoDaoSupport, IAccountCreditDao
{
    public void CreateCredit(float creditAmount)
    {
        AdoTemplate.ExecuteNonQuery(CommandType.Text,
            "insert into Credits (CreditAmount) VALUES (@amount)", "amount", DbType.Decimal, 0,
            creditAmount);
    }
}
```

and for the Debit DAO

```
public class AccountDebitDao : AdoDaoSupport, IAccountDebitDao
{
    public void DebitAccount(float debitAmount)
    {
        AdoTemplate.ExecuteNonQuery(CommandType.Text,
            "insert into dbo.Debits (DebitAmount) VALUES (@amount)", "amount", DbType.Decimal, 0,
            debitAmount);
    }
}
```

Both of these DAO implementations inherit from Spring's `AdoDaoSupport` class that provides convenient access to an `AdoTemplate` for performing data access operations. With no other properties that can be configured in these implementations, the only configuration required is setting of `AdoDaoSupport`'s `DbProvider` property representing the connection to the database.

The implementation of the service layer interface, `IAccountManager`, is shown below.

```
public class AccountManager : IAccountManager
{
    private IAccountCreditDao accountCreditDao;
    private IAccountDebitDao accountDebitDao;

    private float maxTransferAmount = 1000000;

    public AccountManager(IAccountCreditDao accountCreditDao, IAccountDebitDao accountDebitDao)
    {
        this.accountCreditDao = accountCreditDao;
        this.accountDebitDao = accountDebitDao;
    }

    public float MaxTransferAmount
    {
        get { return maxTransferAmount; }
        set { maxTransferAmount = value; }
    }

    [Transaction]
    public void DoTransfer(float creditAmount, float debitAmount)
    {
        accountCreditDao.CreateCredit(creditAmount);

        if (creditAmount > maxTransferAmount || debitAmount > maxTransferAmount)
        {
            throw new ArithmeticException("see a teller big spender...");
        }

        accountDebitDao.DebitAccount(debitAmount);
    }
}
```

The if statement is a poor-mans representation of business logic, namely that there is a policy that does not allow the use of this service for amounts larger than \$1,000,000. If the credit or debit amount is larger than 1,000,000 then an exception will be thrown. We can write a unit test that will test for this business logic and provide stub implementations of the DAO objects so that our tests are not only independent of the database but will also execute very quickly.



Note

Notice the Transaction attribute on the `DoTransfer` method. This attribute can be read by Spring and used to create a transactional proxy to `AccountManager` in order to perform declarative transaction management.

The NUnit unit test for `AccountManager` is shown below

```
public class AccountManagerUnitTests
{
    private IAccountManager accountManager;

    [SetUp]
    public void Setup()
    {
        IAccountCreditDao stubCreditDao = new StubAccountCreditDao();
        IAccountDebitDao stubDebitDao = new StubAccountDebitDao();
        accountManager = new AccountManager(stubCreditDao, stubDebitDao);
    }

    [Test]
    public void TransferBelowMaxAmount()
    {
        accountManager.DoTransfer(217, 217);
    }

    [Test]
    [ExpectedException(typeof(ArithmeticException))]
    public void TransferAboveMaxAmount()
    {
        accountManager.DoTransfer(2000000, 200000);
    }
}
```

Running these tests we exercise both code pathways through the method `DoTransfer`. Nothing we have done so far is Spring specific (aside from the presence of the `[Transaction]` attribute). Now that we know the class works in isolation, we can now 'wire' up the application for use in production by specifying how the service and DAO layers are related. This configuration file is shown below and can loosely be referred to as your 'application blueprint'. This configuration file is named `application-config.xml` and is an embedded resource inside the 'main' project, `Spring.TxQuickStart`.

```
<objects xmlns='http://www.springframework.net'>

    <!-- DAO Implementations -->
    <object id="accountCreditDao" type="Spring.TxQuickStart.Dao.Ado.AccountCreditDao, Spring.TxQuickStart">
        <property name="DbProvider" ref="CreditDbProvider"/>
    </object>

    <object id="accountDebitDao" type="Spring.TxQuickStart.Dao.Ado.AccountDebitDao, Spring.TxQuickStart">
        <property name="DbProvider" ref="DebitDbProvider"/>
    </object>

    <!-- The service that performs multiple data access operations -->
    <object id="accountManager"
        type="Spring.TxQuickStart.Services.AccountManager, Spring.TxQuickStart">
        <constructor-arg name="accountCreditDao" ref="accountCreditDao"/>
        <constructor-arg name="accountDebitDao" ref="accountDebitDao"/>
    </object>
```

```
</objects>
```

This configuration is selecting the real ADO.NET implementations that will insert records into the database. We can now write a NUnit integration test that will test the service and DAO layers. To do this we add on configuration information specific to our test environment. This extra configuration information will determine what databases we speak to and what transaction manager (local or distribute) to use. The code for this integration style NUnit test is shown below

```
[TestFixture]
public class AccountManagerTests
{
    private AdoTemplate adoTemplateCredit;
    private AdoTemplate adoTemplateDebit;

    private IAccountManager accountManager;

    [SetUp]
    public void SetUp()
    {
        // Configure Spring programmatically
        NamespaceParserRegistry.RegisterParser(typeof(DatabaseNamespaceParser));
        NamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParser));
        NamespaceParserRegistry.RegisterParser(typeof(AopNamespaceParser));
        IApplicationContext context = new XmlApplicationContext(
            "assembly://Spring.TxQuickStart.Tests/Spring.TxQuickStart/system-test-local-config.xml"
        );
        accountManager = context["accountManager"] as IAccountManager;
        CleanDb(context);
    }

    [Test]
    public void TransferBelowMaxAmount()
    {
        accountManager.DoTransfer(217, 217);

        int numCreditRecords = (int)adoTemplateCredit.ExecuteScalar(CommandType.Text, "select count(*) from Credits");
        int numDebitRecords = (int)adoTemplateDebit.ExecuteScalar(CommandType.Text, "select count(*) from Debits");
        Assert.AreEqual(1, numCreditRecords);
        Assert.AreEqual(1, numDebitRecords);
    }

    [Test]
    [ExpectedException(typeof(ArithmeticException))]
    public void TransferAboveMaxAmount()
    {
        accountManager.DoTransfer(2000000, 200000);
    }

    private void CleanDb(IApplicationContext context)
    {
        IDbProvider dbProvider = (IDbProvider)context["DebitDbProvider"];
        adoTemplateDebit = new AdoTemplate(dbProvider);
        adoTemplateDebit.ExecuteNonQuery(CommandType.Text, "truncate table Debits");

        dbProvider = (IDbProvider)context["CreditDbProvider"];
        adoTemplateCredit = new AdoTemplate(dbProvider);
        adoTemplateCredit.ExecuteNonQuery(CommandType.Text, "truncate table Credits");
    }
}
```

The essential element is to create an instance of Spring's application context where the relevant layers of the application are 'wired' together. The `IAccountManager` implementation is retrieved from the IoC container and stored as a field of the test class. The basic logic of the test is the same as in the unit test but in addition there is the verification of actions performed in the database. The set up method puts the database tables into a known

state before running the tests. Other techniques for performing integration testing that can alleviate the need to do extensive database state management for integration tests is described in the testing section.

41.4. Configuration

The configuration file `system-test-local-config.xml` shown in the previous program listing includes `application-config.xml` and specifies the database to use and the local (not distributed) transaction manager `AdoPlatformTransactionManager`. This configuration file is shown below

```
<objects xmlns="http://www.springframework.net"
  xmlns:db="http://www.springframework.net/database"
  xmlns:tx="http://www.springframework.net/tx">

  <!-- Imports application configuration -->
  <import resource="assembly://Spring.TxQuickStart/Spring.TxQuickStart/application-config.xml"/>

  <!-- Imports additional aspects -->
  <!--
  <import resource="assembly://Spring.TxQuickStart.Tests/Spring.TxQuickStart/aspects-config.xml"/>
  -->

  <!-- Database Providers -->

  <db:provider id="DebitDbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=MARKT60\SQL2005;Initial Catalog=CreditsAndDebits;User ID=springqa; Password=spr

  <db:provider id="CreditDbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=MARKT60\SQL2005;Initial Catalog=CreditsAndDebits;User ID=springqa; Password=spr

  <alias name="DebitDbProvider" alias="CreditDbProvider"/>

  <!-- Transaction Manager if using a single database that contain both credit and debit tables -->
  <object id="transactionManager"
    type="Spring.Data.Core.AdoPlatformTransactionManager, Spring.Data">
    <property name="DbProvider" ref="DebitDbProvider"/>
  </object>

  <!-- Transaction aspect -->

  <tx:attribute-driven/>

</objects>
```

Moving from top to bottom in the configuration file, the 'application-blueprint' configuration file is included. Then the database type and connection parameters are specified for the two databases. The names of these providers must match those specific in `application-config.xml`. Since the two names point to the same database, an alias configuration element is used to have them point to the same `dbProvider` under different names. The type of transaction manager is then selected, in this case we are showing the use of local transactions with `AdoPlatformTransactionManager`. Running the tests will result in 217 being entered into the Credits and Debits table of each database. You can fire up SQL Server Management Studio or equivalent to verify this.

To switch to a distributed transaction you can refer to the configuration file `system-test-dtc-config.xml`, which is shown below

```
<objects xmlns='http://www.springframework.net'
  xmlns:db="http://www.springframework.net/database"
  xmlns:tx="http://www.springframework.net/tx">

  <!-- Imports application configuration -->
  <import resource="assembly://Spring.TxQuickStart/Spring.TxQuickStart/application-config.xml"/>
```

```

<!-- Imports additional aspects -->
<!--
<import resource="assembly://Spring.TxQuickStart.Tests/Spring.TxQuickStart/aspects-config.xml"/>
-->

<db:provider id="DebitDbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=MARKT60\SQL2005;Initial Catalog=Debits;User ID=springqa; Password=springqa"/>

<db:provider id="CreditDbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=MARKT60\SQL2005;Initial Catalog=Credits;User ID=springqa; Password=springqa"/>

<!-- Transaction Manager if using two databases, one containing the credit table and the other a debit table -->

<object id="transactionManager"
    type="Spring.Data.Core.TxScopeTransactionManager, Spring.Data">
</object>

<!-- Transaction aspect -->
<tx:attribute-driven/>

</objects>

```

TxScopeTransactionManager uses .NET 2.0 System.Transactions as the implementation, allowing for distributed transactions between the two different databases listed. In a larger application the different layers would typically be broken up into individual configuration files and imported into the main configuration file. This allows your configuration to mirror your architecture.

You can also use the configuration file system-test-dtc-es-config.xml that will use EnterpriseServices to perform transaction management.

41.4.1. Rollback Rules

Using Rollback rules allows you to specify which exceptions will not cause a rollback and instead only stop execution flow, committing the work done up to the exception. An alternative implementation of AccountManager's DoTransfer method (included in the sample code) is shown below.

```

[Transaction(NoRollbackFor = new Type[] { typeof(ArithmeticException) })]
public void DoTransfer(float creditAmount, float debitAmount)
{
    accountCreditDao.CreateCredit(creditAmount);

    if (creditAmount > maxTransferAmount || debitAmount > maxTransferAmount)
    {
        throw new ArithmeticException("see a teller big spender...");
    }

    accountDebitDao.DebitAccount(debitAmount);
}

```

All that has changed is the use of the NoRollbackFor property on the transaction attribute.

The expected behavior is that the credit table will be updated even though the exception is thrown. This is due to specifying that exceptions of the type ArithmeticException should not rollback the database transaction. Running the test code below verifies that the exception still propagates out of the method.

```

[Test]
public void DeclarativeWithAttributesNoRollbackFor()
{
    try
    {

```



```

        accountManager.DoTransfer(2000000, 2000000);
        Assert.Fail("Should have thrown Arithmetic Exception");
    } catch (ArithmeticException) {
        int numCreditRecords = (int)adoTemplateCredit.ExecuteScalar(CommandType.Text, "select count(*) from Credits");
        int numDebitRecords = (int)adoTemplateDebit.ExecuteScalar(CommandType.Text, "select count(*) from Debits");
        Assert.AreEqual(1, numCreditRecords);
        Assert.AreEqual(0, numDebitRecords);
    }
}

```

41.5. Adding additional Aspects

Transactional advice is just one type of advice that can be applied to the service layer. You can also configure other pieces of advice to be executed as part of the general advice chain that is associated with methods that have the Transaction attribute applied. In this example we will add logging of thrown exceptions using Spring's `ExceptionHandlerAdvice` as well as logging of the service layer method invocation. No code is required to be changed in order to have this additional functionality. Instead all you have to do is uncomment the line

```
<import resource="assembly://Spring.TxQuickStart.Tests/Spring.TxQuickStart/aspects-config.xml"/>
```

in either `system-test-dtc-config.xml` or `system-test-local-config.xml`. The aspect configuration file is shown below

```

<objects xmlns='http://www.springframework.net'
         xmlns:aop="http://www.springframework.net/aop">

    <object name="exceptionAdvice" type="Spring.Aspects.Exceptions.ExceptionHandlerAdvice, Spring.Aop">
        <property name="exceptionHandlers">
            <list>
                <value>on exception name ArithmeticException log 'Logging an exception thrown from method ' + #method.Name </value>
            </list>
        </property>
    </object>

    <object name="loggingAdvice" type="Spring.Aspects.Logging.SimpleLoggingAdvice, Spring.Aop">
        <property name="logUniqueIdentifier" value="true"/>
        <property name="logExecutionTime" value="true"/>
        <property name="logMethodArguments" value="true"/>
        <property name="Separator" value=";"/>

        <property name="HideProxyTypeNames" value="true"/>
        <property name="UseDynamicLogger" value="true"/>

        <property name="LogLevel" value="Info"/>
    </object>

    <object id="txAttributePointcut" type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop">
        <property name="Attribute" value="Spring.Transaction.Interceptor.TransactionAttribute, Spring.Data"/>
    </object>

    <aop:config>

        <aop:advisor id="exceptionProcessAdvisor" order="1"
                    advice-ref="exceptionAdvice"
                    pointcut-ref="txAttributePointcut"/>

        <aop:advisor id="loggingAdvisor" order="2"
                    advice-ref="loggingAdvice"
                    pointcut-ref="txAttributePointcut"/>

    </aop:config>

</objects>

```

The transaction aspect is now additionally configured with an order value of "10", which will place it after the execution of the exception aspect, which is configured to use an order value of 1.

The behavior for logging the exception is specified by creating and configuring an instance of `Spring.Aspects.Exceptions.ExceptionHandlerAdvice`. The location where that behavior is applied, the pointcut, is the `Transaction` attribute. The logging of method arguments and execution time is specified by configuring an instance of `Spring.Aspects.Logging.SimpleLoggingAdvice`.

The AOP configuration section on the bottom is what ties together the behavior and where it will take place in the program flow. Under the covers the transaction configuration, `<tx:attribute-driven/>` creates similar advice and pointcut definitions. Running the test `TransferBelowMaxAmount` will then log the following messages

```
INFO - Entering DoTransfer;45b6af04-b736-4efa-a489-45462726ddf2;creditAmount=217; debitAmount=217
INFO - Exiting DoTransfer;45b6af04-b736-4efa-a489-45462726ddf2;1328.125 ms;return=
```

When the test case of the test `TransferAboveMaxAmount` is run the following messages are logged

```
INFO - Entering DoTransfer;d94bc81b-a4ff-4cal-9aaa-f2834f262307;creditAmount=2000000; debitAmount=200000
INFO - Exception thrown in DoTransferDoTransfer;d94bc81b-a4ff-4cal-9aaa-f2834f262307;1140.625
System.ArithmeticException: see a teller big spender...
    at Spring.TxQuickStart.Services.AccountManager.DoTransfer(Single creditAmount, Single debitAmount) in L:\projects\Spring.
    at Spring.DynamicReflection.Method_DoTransfer_ec48557f22b149958fd2243413136600.Invoke(Object target, Object[] args)
    at Spring.Reflection.Dynamic.SafeMethod.Invoke(Object target, Object[] arguments) in l:\projects\Spring.Net\src\Spring\Sp
    at Spring.Aop.Framework.DynamicMethodInvocation.InvokeJoinpoint() in l:\projects\Spring.Net\src\Spring\Spring.Aop\Aop\Fra
    at Spring.Aop.Framework.AbstractMethodInvocation.Proceed() in l:\projects\Spring.Net\src\Spring\Spring.Aop\Aop\Framework\
    at Spring.Transaction.Interceptor.TransactionInterceptor.Invoke(IMethodInvocation invocation) in l:\projects\Spring.Net\s
    at Spring.Aop.Framework.AbstractMethodInvocation.Proceed() in l:\projects\Spring.Net\src\Spring\Spring.Aop\Aop\Framework\
    at Spring.Aspects.Logging.SimpleLoggingAdvice.InvokeUnderLog(IMethodInvocation invocation, ILog log) in l:\projects\Spring
TRACE - Logging an exception thrown from method DoTransfer
```

Chapter 42. NHibernate QuickStart

42.1. Introduction

This QuickStart application uses the all too familiar Northwind database and uses NHibernate browse and edit customers. It is a very simple application that directly uses the DAO layer in many use-cases, as it is doing nothing more than table maintenance, but there is also a simple service layer that handles a fulfillment process. The application uses Spring's declarative transaction management features, standard NHibernate API, and Open Session In View module. See Chapter 21, *Object Relational Mapping (ORM) data access* for information on those features.



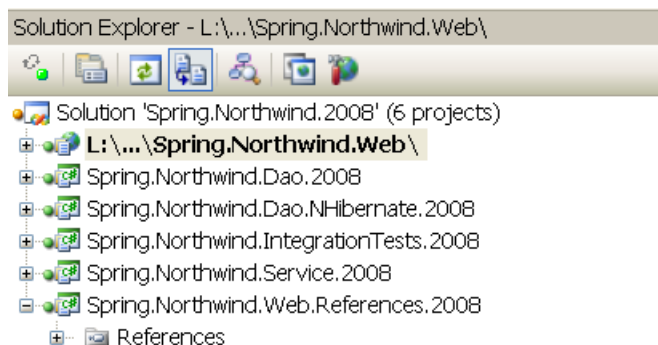
Note

Even though data access is performed through NHibernate API all Spring.NET provided functionality is still present when using the standard NHibernate API, as Spring transaction management is integrated into NHibernate extension points and exception translation is provided by AOP advice.

42.2. Getting Started

The QuickStart application is located in the directory `<spring-install-dir>\examples\Spring\Spring.Data.NHibernate.Northwind`. Load the application using the VS.NET 2008 solution file `Spring.Northwind.2008.sln`. The application uses the SQLite database so no additional configuration is needed. To run the application set the Web application as the project that starts and `Default.aspx` as the start page.

The application has several layers with each layer represented as one or more VS.NET projects.



Solution Explorer for NHibernate QuickStart Application

The data access layer consists of two projects, `Spring.Northwind.Dao` and `Spring.Northwind.Dao.NHibernate`. The former contains only the DAO (data access object) interfaces and the latter the NHibernate implementation of those interfaces. The project `Spring.Northwind.Service` contains a simple service that calls into multiple DAO objects in order to satisfy a fulfillment process. The Web project is a ASP.NET web application and the `Spring.Northwind.IntegrationTests` project contains integration tests for the DAO and Service layers.

When you run the application you will see



Goes Northwind with NHibernate

Welcome to Spring.NET Northwind sample application!

This sample demonstrates Spring.NET NHibernate integration and concepts. All data access is done using NHibernate and all transactions are automatically handled by Spring.NET.

[Proceed to customer listing »](#)

Following the link to the customer listing pages bring up the following screen



Goes Northwind with NHibernate

Customers in database

Below you can see all customers in the database paged to grid. By clicking customer name you enter the details and edit information. By clicking orders you can see all orders that customer has. Orders can be in shipped state (they have shipped date) or they can be waiting for shipment. You can ship orders that aren't shipped yet on the orders screen.

Id	Name	Company	
ALFKI	Maria Anders	Alfreds Futterkiste	Orders
ANATR	Ana Trujillo	Ana Trujillo Emparedados y helados	Orders
ANTON	Antonio Moreno	Antonio Moreno Taquería	Orders
AROUT	Thomas Hardy	Around the Horn	Orders
BERGS	Christina Berglund	Berglunds snabbköp	Orders
BLAUS	Hanna Moos	Blauer See Delikatessen	Orders
BLONP	Frédérique Citeaux	Blondesddsl père et fils	Orders
BOLID	Martin Sommer	Bólide Comidas preparadas	Orders
BONAP	Laurence Lebihan	Bon app'	Orders
BOTTM	Elizabeth Lincoln	Bottom-Dollar Markets	Orders
< ≥			

You can click on the Name of the customer or the Orders link to view that customers orders. Selecting "BOTTM"'s orders brings us to the next page

Orders for customer **Bottom-Dollar Markets**

On this screen you can see all orders that customer has. By clicking "Process Orders" you can send service request to ship all unshipped orders for that customer. Behind the scenes Spring.NET will begin service level transaction that propagates to DAO layer which then automatically participates int the existing transaction.

OrderID	OrderDate	ShippedDate
10389	20.12.1996	24.12.1996
10410	10.1.1997	15.1.1997
10411	10.1.1997	21.1.1997
10431	30.1.1997	7.2.1997
10492	1.4.1997	11.4.1997
10742	14.11.1997	18.11.1997
10918	2.3.1998	11.3.1998
10944	12.3.1998	13.3.1998
10949	13.3.1998	17.3.1998
10975	25.3.1998	27.3.1998
10982	27.3.1998	8.4.1998
11027	16.4.1998	20.4.1998
11045	23.4.1998	
11048	24.4.1998	30.4.1998

Process Orders

[« Back to customer list](#)

Notice that the order 11045 has yet to be shipped. If you select 'Process Orders' this will call the Fulliment Service and the order will be processed and shipped.p

Order processing results

Below you can see the results from order fulfillment.

```
11:33:56 WARN FulfillmentService: Order with 10389 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10410 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10411 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10431 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10492 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10742 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10918 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10944 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10949 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10975 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 10982 has already been shipped, skipping.
11:33:56 WARN FulfillmentService: Order with 11027 has already been shipped, skipping.
11:33:56 INFO FulfillmentService: Order 11045 validated, proceeding with shipping..
11:33:56 INFO FedExShippingService: Shipping order id = 11045
11:33:56 WARN FulfillmentService: Order with 11048 has already been shipped, skipping.
```

[« Back to customer order list](#)

You can then go back to the customer list. If you select the name Elizabeth Lincoln, then you can edit the customer details.

Update customer information

Customer details

ID:

Contact:

[« Back to customer list](#) | [Cancel edit »](#)

42.3. Implementation

This section discussed the Spring implementation details for each layer.

42.3.1. The Data Access Layer

The interface IDao is a generic DAO layer that provides basic retrieval methods. They are located in the Spring.Northwind.Dao project.

```
public interface IDao<TEntity, TId>
```

```
{  
    TEntity Get(TId id);  
  
    IList<TEntity> GetAll();  
  
}
```

The `ISupportsSave` and `ISupportsDeleteDao` interfaces provide the rest of the CRUD functionality.

```
public interface ISupportsSave<TEntity, TId>  
{  
    TId Save(TEntity entity);  
  
    void Update(TEntity entity);  
}  
  
public interface ISupportsDeleteDao<TEntity>  
{  
    void Delete(TEntity entity);  
}
```

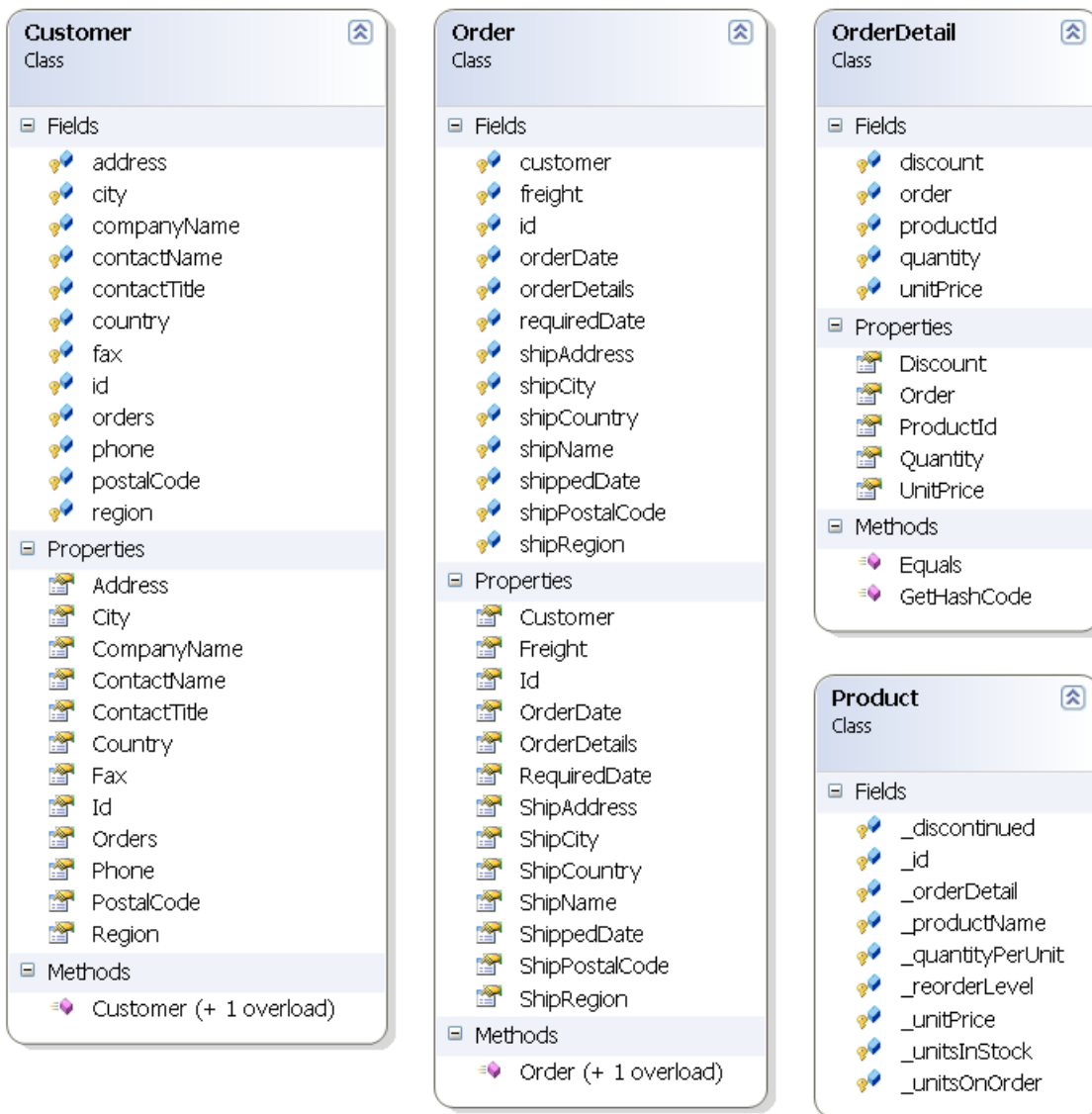
The `ICustomerDao` interface combines these to manage the persistence of customer objects.

```
public interface ICustomerDao : IDao<Customer, string>, ISupportsDeleteDao<Customer>, ISupportsSave<Customer, string>  
{  
}
```

Similar interfaces are defined to manage [Order](#) and [Products](#) in [IOrderDao](#) and [IProductDao](#) respectfully.

42.3.2. The domain objects

The POCO domain objects, `Customer`, `Order`, `OrderDetail` and `Product` are defined in the `Spring.Northwind.Domain` namespace within the `Spring.Northwind.Dao` project.



42.3.3. NHibernate based DAO implementation

The NHibernate based DAO implementation uses the standard NHibernate APIs, retrieving the current session from the SessionFactory and using the session to retrieve or store objects to the database. An abstract base class HibernateDao is used to capture the common ISessionFactory property, provide a convenience property to access the current session, and define a GetAll Method.p

```
public abstract class HibernateDao
{
    private ISessionFactory sessionFactory;

    /// <summary>
    /// Session factory for sub-classes.
    /// </summary>
    public ISessionFactory SessionFactory
    {
        protected get { return sessionFactory; }
        set { sessionFactory = value; }
    }

    /// <summary>
    /// Get's the current active session. Will retrieve session as managed by the
    /// Open Session In View module if enabled.
    /// </summary>
    protected ISession CurrentSession
    {

```



```

        get { return sessionFactory.GetCurrentSession(); }
    }

    protected IList<T> GetAll<T>() where T : class
    {
        ICriteria criteria = CurrentSession.CreateCriteria<T>();
        return criteria.List<T>();
    }
}

```

The implementation of `ICustomerDao` is shown below

```

@Repository
public class HibernateCustomerDao : HibernateDao, ICustomerDao
{
    // Note that the transaction demaraction is here only for the case when
    // the DAO object is being used directly, i.e. not as part of a service layer
    // call. This would be commonly only when creating an application that contains
    // no business logic and is essentially a table maintenance application.
    // These applications are affectionally known as 'CRUD' applications, the acronym
    // referring to Create, Retrieve, Update, And Delete and the only operations
    // performed by the application.

    // If called from a transactional service layer, typically with the transaction
    // propagation setting set to REQUIRED, then any DAO operations will use the
    // same settings as started from the transactional layer.

    [Transaction(ReadOnly = true)]
    public Customer Get(string customerId)
    {
        return CurrentSession.Get<Customer>(customerId);
    }

    [Transaction(ReadOnly = true)]
    public IList<Customer> GetAll()
    {
        return GetAll<Customer>();
    }

    [Transaction(ReadOnly = false)]
    public string Save(Customer customer)
    {
        return (string) CurrentSession.Save(customer);
    }

    [Transaction(ReadOnly = false)]
    public void Update(Customer customer)
    {
        CurrentSession.SaveOrUpdate(customer);
    }

    [Transaction(ReadOnly = false)]
    public void Delete(Customer customer)
    {
        CurrentSession.Delete(customer);
    }
}

```



Note

As mentioned in the code comments above, as this application has a distinctly CRUD based component, Spring's Transaction attribute is used to ensure that that method exeuctes as a unit of work. Often in more sophisticated applications even the basic of CRUD are handled through a service layer so as to enforce security, auditing, altering or enforce business rules.

The `Repository` attribute is used to indicate that this class plays the role of a Repository or a Data Access Object. The term repository comes from modeling terminology popularized by Eric Evan's book Domain Driven Design

(DDD). Those familiar with DDD will note that this implementation is very simply and does not expose higher level persistence functionality to the application, for example `FindCustomersWithOpenOrders`. How well the role of Repository applies to this implementation is not relevant, and we will often refer to Repository and DAO interchangeably when describing the data access layer. What *is* relevant is that the `Repository` attribute serves as a marker, a place in the code that can be used to identify methods whose invocation should be intercepted so that additional behavior can be added. In Aspect-Oriented Programming terminology, the Repository attribute represents a pointcut. The behavior that we would like to add to this DAO implementation is exception translation. Exception translation from the data access layer to a service layer is important as it shields the service layer from the implementation details of the data access layer. A NHibernate based DAO will throw different exceptions and an ADO.NET based implementation and so on. Spring provides a rich technology neutral data-access exception hierarchy. See Chapter 18, *DAO support*.

Instead of adding exception translation code in each data access method, AOP offers a simple solution. Using Spring's `IOExceptionPostProcessor` extension point, each DAO object that is managed by Spring will be automatically wrapped up in a proxy that adds the exception translation behavior. This is done by adding the following object definition to the Spring application context.

```
<objects>

  <!-- configure session factory -->

  <!-- Exception translation object post processor -->
  <object type="Spring.Dao.Attributes.PersistenceExceptionTranslationPostProcessor, Spring.Data"/>

  <!-- Configure transaction management strategy -->
  <!-- DAO objects go here -->

</objects>
```

The Spring managed DAO object definitions are shown below, referring to a SessionFactory that is created via Spring's `LocalSessionFactoryObject`. See the file `Dao.xml` for more details.

```
<objects xmlns="http://www.springframework.net"
  xmlns:db="http://www.springframework.net/database">

  <!-- Referenced by main application context configuration file -->
  <description>
    The Northwind object definitions for the Data Access Objects.
  </description>

  <!-- Database Configuration -->
  <db:provider id="DbProvider"
    provider="SQLite-1.0.65"
    connectionString="Data Source=|DataDirectory|Northwind.db;Version=3;FailIfMissing=True;"/>

  <!-- NHibernate SessionFactory configuration -->
  <object id="NHibernateSessionFactory" type="Spring.Data.NHibernate.LocalSessionFactoryObject, Spring.Data.NHibernate21">
    <property name="DbProvider" ref="DbProvider"/>
    <property name="MappingAssemblies">
      <list>
        <value>Spring.Northwind.Dao.NHibernate</value>
      </list>
    </property>
    <property name="HibernateProperties">
      <dictionary>
        <entry key="hibernate.connection.provider" value="NHibernate.Connection.DriverConnectionProvider"/>
        <entry key="dialect" value="NHibernate.Dialect.SQLiteDialect"/>
        <entry key="connection.driver_class" value="NHibernate.Driver.SQLite20Driver"/>
      </dictionary>
    </property>

    <!-- provides integration with Spring's declarative transaction management features -->
    <property name="ExposeTransactionAwareSessionFactory" value="true" />
  </object>
</objects>
```

```

</object>

<!-- Transaction Management Strategy - local database transactions -->
<object id="transactionManager"
    type="Spring.Data.NHibernate.HibernateTransactionManager, Spring.Data.NHibernate21">
    <property name="DbProvider" ref="DbProvider"/>
    <property name="SessionFactory" ref="NHibernateSessionFactory"/>
</object>

<!-- Exception translation object post processor -->
<object type="Spring.Dao.Attributes.PersistenceExceptionTranslationPostProcessor, Spring.Data"/>

<!-- Data Access Objects -->
<object id="CustomerDao" type="Spring.Northwind.Dao.NHibernate.HibernateCustomerDao, Spring.Northwind.Dao.NHibernate">
    <property name="SessionFactory" ref="NHibernateSessionFactory"/>
</object>

<object id="OrderDao" type="Spring.Northwind.Dao.NHibernate.HibernateOrderDao, Spring.Northwind.Dao.NHibernate">
    <property name="SessionFactory" ref="NHibernateSessionFactory"/>
</object>

</objects>

```



Note

It is not required that you use Spring's [\[Repository\]](#) attribute. You can specify an attribute type to the [PersistenceExceptionTranslationPostProcessor](#) via the property [RepositoryAttributeType](#) to avoid coupling your DAO implementation to Spring.

42.3.4. The Service layer

The service layer is located in the `Spring.Northwind.Services` project. It defines a single service for the fulfillment process

```

public interface IFulfillmentService
{
    void ProcessCustomer(string customerId);
}

```

The implementation class is shown below

```

public class FulfillmentService : IFulfillmentService
{
    private IProductDao productDao;

    private ICustomerDao customerDao;

    private IOrderDao orderDao;

    private IShippingService shippingService;

    // Properties for the preceding fields omitted for brevity

    [Transaction]
    public void ProcessCustomer(string customerId)
    {
        //Find all orders for customer
        Customer customer = CustomerDao.Get(customerId);

        foreach (Order order in customer.Orders)
        {
            if (order.ShippedDate.HasValue)
            {
                log.Warn("Order with " + order.Id + " has already been shipped, skipping.");
                continue;
            }
        }
    }
}

```

```

    }

    //Validate Order
    Validate(order);
    log.Info("Order " + order.Id + " validated, proceeding with shipping..");

    //Ship with external shipping service
    ShippingService.ShipOrder(order);

    //Update shipping date
    order.ShippedDate = DateTime.Now;

    //Update shipment date
    OrderDao.Update(order);

    //Other operations...Decrease product quantity... etc
}

}

private void Validate(Order order)
{
    //no-op - throw exception on error.
}
}
}

```

What is important to note about this method is that it uses two DAO objects, CustomerDao and OrderDao as well as an additional collaborating service, IShippingService. The fact that all of the collaborating objects are interfaced based means that we can write a unit test for the business functionality of the ProcessCustomer method. Also, the use of the [Transaction] attribute will enable this business processing to proceed as a single unit-of-work. Spring's declarative transaction management features make it very easy to mix and match different DAO objects with a service method without having to worry about propagating the transaction/connection or hibernate session to each DAO object.

The Fullfillment service layer is configured to refer to its collaborating objects as shown below in the configuration file Services.xml

```

<!-- Property placeholder configurer for database settings -->
<object id="FulfillmentService" type="Spring.Northwind.Service.FulfillmentService, Spring.Northwind.Service">
    <property name="CustomerDao" ref="CustomerDao"/>
    <property name="OrderDao" ref="OrderDao"/>
    <property name="ShippingService" ref="ShippingService"/>
</object>

<object id="ShippingService" type="Spring.Northwind.Service.FedExShippingService, Spring.Northwind.Service"/>

<tx:attribute-driven/>

```

42.3.5. Integration testing

Integraton testing in addition to unit testing can be done before integrating the service and data access layer into the Web application - where automated testing is much more difficult. While coding to interfaces and using an IoC container help enable unit testing, unit tests should not have any Spring dependency. Integration tests however greatly benefit from being able to access the objects that Spring is managing in production. This way the gap between what you test in QA and what runs is minimized, ideally with only environment specific settings being different. In addition to easily obtaining, say a transactionally aware service object from the Spring IoC container, Spring intergration testing support classes allow you to implicitly start a transaction at the start of test method and rollback at the end. The isolation guaranteed by the database means that multiple developers can run integration tests for their data access layers simultaneously and the rollback ensures that the changes made are not persisted. While in the test method, you have a consistent view of the data and can therefore exercise all the methods of your DAO object.

The project `Spring.Northwind.IntegrationTests` shows how this works. As a convenience, an abstract base class is created that in turn inherits from Spring's integration testing class [AbstractTransactionalDbProviderSpringContextTests](#)

```
[TestFixture]
public abstract class AbstractDaoIntegrationTests : AbstractTransactionalDbProviderSpringContextTests
{
    protected override string[] ConfigLocations
    {
        get
        {
            return new string[]
            {
                "assembly://Spring.Northwind.Dao.NHibernate/Spring.Northwind.Dao/Dao.xml",
                "assembly://Spring.Northwind.Service/Spring.Northwind.Service/Services.xml"
            };
        }
    }
}
```



Note

This unit test is NUnit based but there is similar support available for Microsoft MSTest framework.

The exact same object definition files that will be used in the production application are loaded for the integration test. To test the data access layer, you inherit from `AbstractDaoIntegrationTests` and expose public properties for each DAO implementation you want to test. Within each test method exercise the API of the DAO. This also tests the NHibernate mappings.

```
[TestFixture]
public class NorthwindIntegrationTests : AbstractDaoIntegrationTests
{
    private ICustomerDao customerDao;
    private IOrderDao orderDao;

    private ISessionFactory sessionFactory;

    // These properties will be injected based on type
    public ICustomerDao CustomerDao
    {
        set { customerDao = value; }
    }

    public IOrderDao OrderDao
    {
        set { orderDao = value; }
    }

    public ISessionFactory SessionFactory
    {
        set { sessionFactory = value; }
    }

    [Test]
    public void CustomerDaoTests()
    {
        Assert.AreEqual(91, customerDao.GetAll().Count);

        Customer c = new Customer();
        c.Id = "MPOLL";
        c.CompanyName = "Interface21";
        customerDao.Save(c);
        c = customerDao.Get("MPOLL");
        Assert.AreEqual(c.Id, "MPOLL");
        Assert.AreEqual(c.CompanyName, "Interface21");

        //Without flushing, nothing changes in the database:
        int customerCount = (int)AdoTemplate.ExecuteScalar(CommandType.Text, "select count(*) from Customers");
    }
}
```

```

Assert.AreEqual(91, customerCount);

//Flush the session to execute sql in the db.
SessionFactoryUtils.GetSession(sessionFactory, true).Flush();

//Now changes are visible outside the session but within the same database transaction
customerCount = (int)AdoTemplate.ExecuteScalar(CommandType.Text, "select count(*) from Customers");
Assert.AreEqual(92, customerCount);

Assert.AreEqual(92, customerDao.GetAll().Count);

c.CompanyName = "SpringSource";

customerDao.Update(c);

c = customerDao.Get("MPOLL");
Assert.AreEqual(c.Id, "MPOLL");
Assert.AreEqual(c.CompanyName, "SpringSource");

customerDao.Delete(c);

SessionFactoryUtils.GetSession(sessionFactory, true).Flush();
customerCount = (int)AdoTemplate.ExecuteScalar(CommandType.Text, "select count(*) from Customers");
Assert.AreEqual(92, customerCount);

try
{
    c = customerDao.Get("MPOLL");
    Assert.Fail("Should have thrown HibernateObjectRetrievalFailureException when finding customer with Id = MPOLL");
}
catch (HibernateObjectRetrievalFailureException e)
{
    Assert.AreEqual("Customer", e.PersistentClassName);
}
}

[Test]
public void ProductDaoTests()
{
    // ommited for brevity
}
}

```

This test uses `AdoTemplate` to access the database using the standard ADO.NET APIs. It is done to demonstrate that the common configuration of NHibernate is for it not to flush to the database until a commit occurs. If we did not explicitly flush, then no SQL would be sent down to the database and some potential errors would go undetected. Since the test method will rollback the transaction, we don't have to worry about 'dirtying' the database and changing its state.

42.3.6. Web Application

The Web application uses Dependency Injection on the .aspx pages so that they can access the services of the middle tier, for example the `FullfillmentService`, or in the case of simple table maintenance, the DAO objects directly.

For example the `FullfillmentResult.aspx` code behind is shown below

```

public partial class FullfillmentResult : Page
{
    private IFulfillmentService fulfillmentService;
    private ICustomerEditController customerEditController;

    public IFulfillmentService FulfillmentService
    {
        set { fulfillmentService = value; }
    }
}

```

```

public ICustomerEditController CustomerEditController
{
    set { customerEditController = value; }
}

protected void Page_Load(object sender, EventArgs e)
{
    /// code omitted for brevity

    fulfillmentService.ProcessCustomer(customerEditController.CurrentCustomer.Id);
}

protected void customerOrders_Click(object sender, EventArgs e)
{
    SetResult("Back");
}

```

The page is configured in Spring as shown below

```

<object type="FulfillmentResult.aspx">
  <property name="FulfillmentService" ref="FulfillmentService" />
  <property name="CustomerEditController" ref="CustomerEditController" />
  <property name="Results">
    <dictionary>
      <entry key="Back" value="redirect:CustomerOrders.aspx" />
    </dictionary>
  </property>
</object>

```

The page is injected with a reference to the FulfillmentService and also another UI component. While Spring's ASP.NET framework supports DI for standard ASP.NET pages and user controls, you can also inherit from Spring's base page class to get added functionality. In this example the use of externalized page flow, or Result Mapping is shown. The Results property indicates the 'how', 'where' and 'what data' to bring along when moving between different web pages and associates it with a logical name "Back". This avoid hardcoding server side transfers or redirects in your code as well as other ASP.NET page references. See the chapter on Spring's ASP.NET Web Framework for more details.

Chapter 43. Quartz QuickStart

43.1. Introduction

In many applications the need arises to perform a certain action at a given time without any user interaction, usually to perform some administrative tasks. These tasks need to be scheduled, say to perform a job in the early hours of the morning before the start of business. This functionality is provided by using job scheduling software. Quartz.NET is an excellent open source job scheduler that can be used for these purposes. It provides a wealth of features, such as persistent jobs and clustering. To find out more about Quartz.NET visit their [web site](#). Spring integration allows you to use Spring to configure Quartz jobs, triggers, and schedulers and also provides integration with Spring's transaction management features.

The full details of Quartz are outside the scope of this quickstart but here is 'quick tour for the impatient' of the main classes and interfaces used in Quartz so you can get your sea legs. A Quartz `IJob` interface represents the task you would like to execute. You either directly implement Quartz's `IJob` interface or a convenience base class. The Quartz `Trigger` controls when a job is executed, for example in the wee hours of the morning every weekday . This would be done using Quartz's `CronTrigger` implementation. Instances of your job are created every time the trigger fires. As such, in order to pass information between different job instances you stash data away in a hashtable that gets passed to the each Job instance upon its creation. Quartz's `JobDetail` class combines the `IJob` and this hashtable of data. Instead of the standard `System.Collections.Hashtable` the class `JobDataMap` is used. Triggers are registered with a Quartz `IScheduler` implementation that manages the overall execution of the triggers and jobs. The `StdSchedulerFactory` implementation is generally used.



Note

To follow this Quartz QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.Scheduling.Quartz.Example`

43.2. Application Overview

The sample application has two types of Jobs. One that inherits from Spring's convenience base class `QuartzJobObject` and another which does not inherit from any base class. The latter class is adapted by Spring to be a Job. Two triggers, one for each of the jobs, are created. These triggers are in turn registered with a scheduler. In each case the job implementation will write information to the console when it is executed.

43.3. Standard job scheduling

The Spring base class `QuartzJobObject` implements `IJob` and allows for your object's properties to be set via values that are stored inside Quartz's `JobDataMap` that is passed along each time your job is instantiated due a trigger firing. This class is shown below

```
public class ExampleJob : QuartzJobObject
{
    private string userName;

    public string UserName
    {
        set { userName = value; }
    }

    protected override void ExecuteInternal(JobExecutionContext context)
    {

```



```

        Console.WriteLine("{0}: ExecuteInternal called, user name: {1}, next fire time {2}",
            DateTime.Now, userName, context.NextFireTimeUtc.Value.ToLocalTime());
    }
}

```

The method `ExecuteInternal` is called when the trigger fires and is where you would put your business logic. The `JobExecutionContext` passed in lets you access various pieces of information about the current job execution, such as the `JobDataMap` or information on when the next time the trigger will fire. The `ExampleJob` is configured by creating a `JobDetail` object as shown below in the following XML snippet taken from `spring-objects.xml`

```

<object name="exampleJob" type="Spring.Scheduling.Quartz.JobDetailObject, Spring.Scheduling.Quartz">
  <property name="JobType" value="Spring.Scheduling.Quartz.Example.ExampleJob, Spring.Scheduling.Quartz.Example" />
  <!-- We can inject values through JobDataMap -->
  <property name="JobDataAsMap">
    <dictionary>
      <entry key="UserName" value="Alexandre" />
    </dictionary>
  </property>
</object>

```

The dictionary property of the `JobDetailObject`, `JobDataAsMap`, is used to set the values of the `ExampleJob`'s properties. This will result in the `ExampleJob` being instantiated with it's `UserName` property value set to 'Alexandre' the first time the trigger fires.

We then will schedule this job to be executed on 20 second increments of every minute as shown below using Spring's `CronTriggerObject` which creates a Quartz `CronTrigger`.

```

<object id="cronTrigger" type="Spring.Scheduling.Quartz.CronTriggerObject, Spring.Scheduling.Quartz">
  <property name="jobDetail" ref="exampleJob" />
  <!-- run every 20 second of minute -->
  <property name="cronExpressionString" value="0/20 * * * * ?" />
</object>

```

Lastly, we schedule this trigger with the scheduler as shown below

```

<object type="Spring.Scheduling.Quartz.SchedulerFactoryObject, Spring.Scheduling.Quartz">
  <property name="triggers">
    <list>
      <ref object="cronTrigger" />
    </list>
  </property>
</object>

```

Running this configuration will produce the following output

```

8/8/2008 1:29:40 PM: ExecuteInternal called, user name: Alexandre, next fire time 8/8/2008 1:30:00 PM
8/8/2008 1:30:00 PM: ExecuteInternal called, user name: Alexandre, next fire time 8/8/2008 1:30:20 PM
8/8/2008 1:30:20 PM: ExecuteInternal called, user name: Alexandre, next fire time 8/8/2008 1:30:40 PM

```

43.4. Scheduling arbitrary methods as jobs

It is very convenient to schedule the execution of method as a job. The `AdminService` class in the example demonstrates this functionality and is listed below.

```

public class AdminService
{
    private string userName;

    public string UserName
    {
        set { userName = value; }
    }
}

```

```

    public void DoAdminWork()
    {
        Console.WriteLine("{0}: DoAdminWork called, user name: {1}", DateTime.Now, userName);
    }
}

```

Note that it does not inherit from any base class. To instruct Spring to create a `JobDetail` object for this method we use Spring's factory object class `MethodInvokingJobDetailFactoryObject` as shown below

```

<object id="adminService" type="Spring.Scheduling.Quartz.Example.AdminService, Spring.Scheduling.Quartz.Example">
  <!-- we inject straight to target object -->
  <property name="UserName" value="admin-service" />
</object>

<object id="jobDetail" type="Spring.Scheduling.Quartz.MethodInvokingJobDetailFactoryObject, Spring.Scheduling.Quartz">
  <!-- We don't actually need to implement IJob as we can use delegation -->
  <property name="TargetObject" ref="adminService" />
  <property name="TargetMethod" value="DoAdminWork" />
</object>

```

Note that `AdminService` object is configured using Spring as you would do normally, without consideration for Quartz. The trigger associated with the jobDetail object is listed below. Also note that when using `MethodInvokingJobDetailFactoryObject` you can't use database persistence for Jobs. See the class documentation for additional details.

```

<object id="simpleTrigger" type="Spring.Scheduling.Quartz.SimpleTriggerObject, Spring.Scheduling.Quartz">
  <!-- see the example of method invoking job above -->
  <property name="jobDetail" ref="jobDetail" />
  <!-- 5 seconds -->
  <property name="startDelay" value="5s" />
  <!-- repeat every 5 seconds -->
  <property name="repeatInterval" value="5s" />
</object>

```

This creates an instances of Quartz's `SimpleTrigger` class (as compared to its `CronTrigger` class used in the previous section). `StartDelay` and `RepeatInterval` properties are `TimeSpan` objects than can be set using the convenient strings such as 10s, 1h, etc, as supported by Spring's custom `TypeConverter` for `TimeSpans`.

This trigger can then be added to the scheduler's list of registered triggers as shown below.

```

<object type="Spring.Scheduling.Quartz.SchedulerFactoryObject, Spring.Scheduling.Quartz">
  <property name="triggers">
    <list>
      <ref object="cronTrigger" />
      <ref object="simpleTrigger" />
    </list>
  </property>
</object>

```

The interleaved output of both these jobs being triggered is shown below.

```

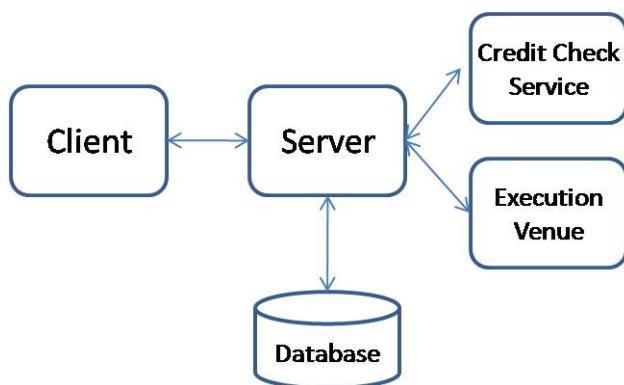
8/8/2008 1:40:18 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:20 PM: ExecuteInternal called, user name: Alexandre, next fire time 8/8/2008 1:40:40 PM
8/8/2008 1:40:23 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:28 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:33 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:38 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:40 PM: ExecuteInternal called, user name: Alexandre, next fire time 8/8/2008 1:41:00 PM
8/8/2008 1:40:43 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:48 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:53 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:40:58 PM: DoAdminWork called, user name: Gabriel
8/8/2008 1:41:00 PM: ExecuteInternal called, user name: Alexandre, next fire time 8/8/2008 1:41:20 PM
8/8/2008 1:41:03 PM: DoAdminWork called, user name: Gabriel

```

Chapter 44. NMS QuickStart

44.1. Introduction

The NMS quick start application demonstrates how to use asynchronous messaging to implement a system for purchasing a stock. To purchase a stock, a client application will send a stock request message containing the information about the stock, i.e. ticker symbol, quantity, etc. The client request message will be received by the server where it will perform business processing on the request, for example to determine if the user has sufficient credit to purchase the stock or if the user is even allowed to make the purchase due to existing account restrictions. These are typically external processes as well. Usually the server application will persist state about the request and forward it on to an execute venue where the actual execution of the stock request is performed. In addition, market data for the stock will be sent from the server process to the client. The high level exchange of information is shown below.



Note

To follow this NMS QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.NmsQuickStart`

44.2. Message Destinations

To implement this flow using messaging the following queues and topics will be used. All requests from the client to the server will be sent on the queue named `APP.STOCK.REQUEST`. Responses to the requests will be sent from the server to the client on a queue unique to each client. In this example the queue name is of the form `APP.STOCK.<UserName>`, and more specifically is configured to be `APP.STOCK.JOE`. Market data does not need to be delivered to an individual client as many client applications are interested in this shared information. As such, the server will send market data information on a topic named `APP.STOCK.MARKETDATA`. The messaging communication between the server and the execution venue is not included as part of the application. An local implementation of the service interface that represents the execution venue is used instead of one based on messaging or another middleware technology. The messaging flow showing the queues and topics used is shown below.



Queues are shown in red and topics in green.

44.3. Gateways

Gateways represent the service operation to send a message. The client will send a stock request to the server based on the contract defined by the `IStockService` interface .

```
public interface IStockService
{
    void Send(TradeRequest tradeRequest);
}
```

The server will send market data to the clients based on the contract defined by the `IMarketDataService` interface.

```
public interface IMarketDataService
{
    void SendMarketData();
}
```

The market data gateway has no method parameters as it is assumed that implementations will manage the data to send internally. The `TradeRequest` object is one of the data objects that will be exchanged in the application and is discussed in the next section.

The use of interfaces allows for multiple implementations to be created. Implementations that use messaging to communicate will be based on the Spring's `NmsGateway` class and will be discussed later. stub or mock implementations can be used for testing purposes.

44.4. Message Data

The `TradeRequest` object shown above contains all the information required to process a stock order. To promote the interoperability of this data across different platforms the `TradeRequest` class is generated from an XML Schema using Microsoft's Schema Definition Tool (xsd.exe). The schema for trade request is shown below

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    targetNamespace="http://www.springframework.net/nms/common/2008-08-05">

    <xs:element name="TradeRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Ticker" type="xs:string"/>
                <xs:element name="Quantity" type="xs:long"/>
                <xs:element name="Price" type="xs:decimal"/>
                <xs:element name="OrderType" type="xs:string"/>
                <xs:element name="AccountName" type="xs:string"/>
                <xs:element name="BuyRequest" type="xs:boolean"/>
                <xs:element name="UserName" type="xs:string"/>
                <xs:element name="RequestID" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

Running `xsd.exe` on this schema will result in a class that contains properties for each of the element names. A partial code listing of the `TradeRequest` class is shown below

```
// This code was generated by a tool.
public partial class TradeRequest {

    public string Ticker {
        get {
            return this.tickerField;
        }
        set {
```

```

        this.tickerField = value;
    }
}

public long Quantity {
    get {
        return this.quantityField;
    }
    set {
        this.quantityField = value;
    }
}

// Additional properties not shown for brevity.
}

```

The schema and the `TradeRequest` class are located in the project `Spring.NmsQuickStart.Common`. This common project will be shared between the server and client for convenience.

When sending a response back to the client the type `TradeResponse` will be used. The schema for the `TradeResponse` is shown below

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    targetNamespace="http://www.springframework.net/nms/common/2008-08-05">

    <xs:element name="TradeResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Ticker" type="xs:string"/>
                <xs:element name="Quantity" type="xs:integer"/>
                <xs:element name="Price" type="xs:decimal"/>
                <xs:element name="OrderType" type="xs:string"/>
                <xs:element name="Error" type="xs:boolean"/>
                <xs:element name="ErrorMessage" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>

```

The `TradeResponse` type also generated from a schema using `xsd.exe`. A partial code listing is shown below

```

// This code was generated by a tool.

public partial class TradeResponse {

    public string Ticker {
        get {
            return this.tickerField;
        }
        set {
            this.tickerField = value;
        }
    }

    public long Quantity {
        get {
            return this.quantityField;
        }
        set {
            this.quantityField = value;
        }
    }

    // Additional properties not shown for brevity.
}

```

The market data information will be sent using a `Hashtable` data structure.

44.5. Message Handlers

When the `TradeRequest` message is received by the server, it will be handled by the class `Spring.NmsQuickStart.Server.Handlers.StockAppHandler` shown below

```
public class StockAppHandler
{
    private IExecutionVenueService executionVenueService;

    private ICreditCheckService creditCheckService;

    private ITradingService tradingService;

    public TradeResponse Handle(TradeRequest tradeRequest)
    {
        TradeResponse tradeResponse;
        IList errors = new ArrayList();
        if (creditCheckService.CanExecute(tradeRequest, errors))
        {
            tradeResponse = executionVenueService.ExecuteTradeRequest(tradeRequest);
            tradingService.ProcessTrade(tradeRequest, tradeResponse);
        }
        else
        {
            tradeResponse = new TradeResponse();
            tradeResponse.Error = true;
            tradeResponse.ErrorMessage = errors[0].ToString();
        }
        return tradeResponse;
    }
}
```

The stub implementations of the services, located in the namespace `Spring.NmsQuickStart.Server.Services.Stubs`, will result in always sending back a error-free trade response message. A realistic implementation would likely have the execution venue and trading service be remote services and the trading service could be implemented as a local transactional service layer that uses spring's declarative transaction management features.

The client will receive a `TradeResponse` message as well as a `Hashtable` of data representing the market data. The message handle for the client is the class `Spring.NmsQuickStart.Client.Handlers.StockAppHandler` and is shown below.

```
public class StockAppHandler
{
    // definition of stockController omitted for brevity.

    public void Handle(Hashtable data)
    {
        // forward to controller to update view
        stockController.UpdateMarketData(data);
    }

    public void Handle(TradeResponse tradeResponse)
    {
        // forward to controller to update view
        stockController.UpdateTrade(tradeResponse);
    }
}
```

What is important to note about these handlers is that they contain no messaging API artifacts. As such you can write unit and integration tests against these classes independent of the middleware. The missing link between the messaging world and the objects processed by the message handlers are message converters. Spring's messaging helper classes, i.e. `SimpleMessageListenerContainer` and `NmsTemplate` use message converters to pass data to the handlers and to send data via messaging for gateway implementations

44.6. Message Converters

The implementation of `IMessageConverter` used is `Spring.NmsQuickStart.Common.Converters.XmlMessageConverter`. This converter adds the ability to marshal and unmarshal objects to and from XML strings. It also uses Spring's `SimpleMessageConverter` to convert Hashtables, strings, and byte arrays. In order to pass information about the serialized type, type information is put in the message properties. The type information can be either the class name or an integer value identifying the type. In systems where the client and server are deployed together and are tightly coupled, sharing the class name is a convenient shortcut. The alternative is to register a type for a given integer value. The XML configuration used to configure these objects is shown below

```
<object name="XmlMessageConverter" type="Spring.NmsQuickStart.Common.Converters.XmlMessageConverter, Spring.NmsQuickStart.Common"
  <property name="TypeMapper" ref="TypeMapper"/>
</object>

<object name="TypeMapper" type="Spring.NmsQuickStart.Common.Converters.TypeMapper, Spring.NmsQuickStart.Common">
  <!-- use simple configuration style -->
  <property name="DefaultNamespace" value="Spring.NmsQuickStart.Common.Data"/>
  <property name="DefaultAssemblyName" value="Spring.NmsQuickStart.Common"/>
</object>
```

This configuration is common between the server and the client.

44.7. Messaging Infrastructure

The implementations of the gateway interfaces inherit from Spring's helper class `NmsGatewaySupport` in order to get easy access to a `NmsTemplate` for sending. The implementation of the `IStockService` interface is shown below

```
public class NmsStockServiceGateway : NmsGatewaySupport, IStockService
{
    private IDestination defaultReplyToQueue;

    public IDestination DefaultReplyToQueue
    {
        set { defaultReplyToQueue = value; }
    }

    public void Send(TradeRequest tradeRequest)
    {
        NmsTemplate.ConvertAndSendWithDelegate(tradeRequest, // post process message
        delegate(IMessage message)
        {
            message.NMSReplyTo = defaultReplyToQueue;
            message.NMSCorrelationID = new Guid().ToString();
            return message;
        });
    }
}
```

The `Send` method is using `NmsTemplate`'s `ConvertAndSendWithDelegate(object obj, MessagePostProcessorDelegate messagePostProcessorDelegate)` method. The anonymous delegate allows you to modify the message properties, such as `NMSReplyTo` and `NMSCorrelationID` after the message has been converted from an object but before it has been sent. The use of an anonymous delegate allows makes it very easy to apply any post processing logic to the converted message.

The object definition for the `NmsStockServiceGateway` is shown below along with its dependent object definitions of `NmsTemplate` and the `ConnectionFactory`.

```
<object name="StockServiceGateway" type="Spring.NmsQuickStart.Client.Gateways.NmsStockServiceGateway, Spring.NmsQuickStart.Common"
  <property name="NmsTemplate" ref="NmsTemplate"/>
</object>
```

```

<property name="DefaultReplyToQueue">
  <object type="Apache.NMS.ActiveMQ.Commands.ActiveMQQueue, Apache.NMS.ActiveMQ">
    <constructor-arg value="APP.STOCK.JOE"/>
  </object>
</property>
</object>

<object name="NmsTemplate" type="Spring.Messaging.Nms.Core.NmsTemplate, Spring.Messaging.Nms">
  <property name="ConnectionFactory" ref="ConnectionFactory"/>
  <property name="DefaultDestinationName" value="APP.STOCK.REQUEST"/>
  <property name="MessageConverter" ref="XmlMessageConverter"/>
</object>

<object id="ConnectionFactory" type="Apache.NMS.ActiveMQ.ConnectionFactory, Apache.NMS.ActiveMQ">
  <constructor-arg index="0" value="tcp://localhost:61616"/>
</object>

```

In this example the 'raw' `Apache.NMS.ActiveMQ.ConnectionFactory` connection factory was used. It would be more efficient resource wise to use Spring's `CachingConnectionFactory` wrapper class so that connections will not be open and closed for each message send as well as allowing for the caching of other intermediate NMS API objects such as sessions and message producers.

A similar configuration is used on the server to configure the class `Spring.NmsQuickStart.Server.Gateways.MarketDataServiceGateway` that implements the `IMarketDataService` interface.

Since the client is also a consumer of messages, on the topic `APP.STOCK.MARKETDATA` and the queue `APP.STOCK.JOE` (for Trader Joe!), two message listener containers are defined as shown below.

```

<nms:listener-container connection-factory="ConnectionFactory">
  <nms:listener ref="MessageListenerAdapter" destination="APP.STOCK.JOE" />
  <nms:listener ref="MessageListenerAdapter" destination="APP.STOCK.MARKETDATA" pubsub-domain="true" />
</nms:listener-container>

```

Refer to the messages reference docs for all the available attributes to configure the container and also the section on registering the NMS schema with Spring..

On the server we define a message listener container for the queue `APP.STOCK.REQUEST` but set the concurrency property to 10 so that 10 threads will be consuming messages from the queue.

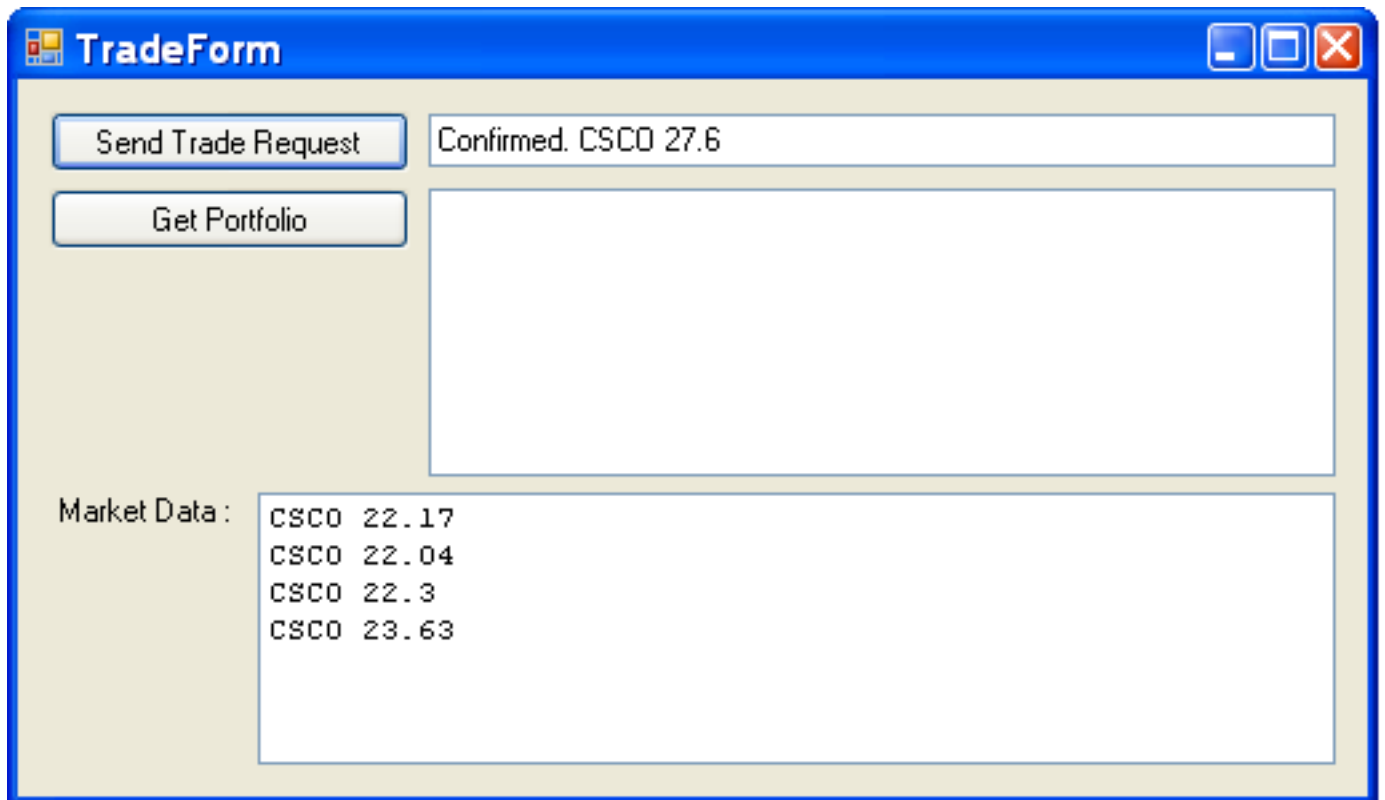
```

<nms:listener-container connection-factory="ConnectionFactory" concurrency="10">
  <nms:listener ref="MessageListenerAdapter" destination="APP.STOCK.REQUEST" />
</nms:listener-container>

```

44.8. Running the application

To run both the client and server make sure that you select 'Multiple Startup Projects' within VS.NET. The GUI has a button to make a hardcoded trade request and show confirmation in a text box. A text area is used to display the market data. There is a 'Get Portfolio' button that is not implemented at the moment. A picture of the GUI after it has been running for a while and trade has been sent and responded to is shown below



The image shows a Java Swing window titled "TradeForm". It has a blue title bar with standard window controls (minimize, maximize, close). The main content area has a light beige background. On the left side, there are two buttons: "Send Trade Request" and "Get Portfolio". To the right of the "Send Trade Request" button is a text field containing the text "Confirmed. CSC0 27.6". Below the "Get Portfolio" button is a large, empty rectangular text area. At the bottom left, the text "Market Data :" is followed by a list of four lines of data: "CSC0 22.17", "CSC0 22.04", "CSC0 22.3", and "CSC0 23.63".

TradeForm

Send Trade Request

Confirmed. CSC0 27.6

Get Portfolio

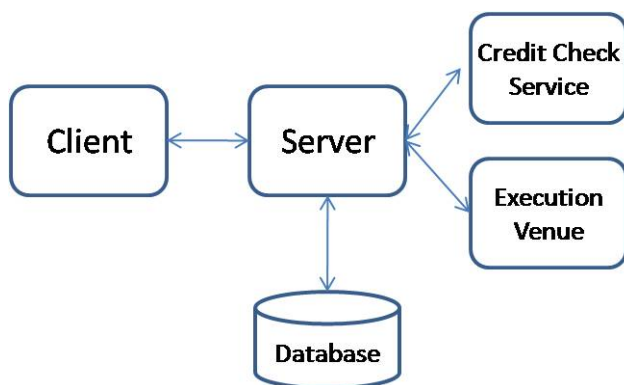
Market Data :

- CSC0 22.17
- CSC0 22.04
- CSC0 22.3
- CSC0 23.63

Chapter 45. TIBCO EMS QuickStart

45.1. Introduction

The TIBCO EMS quick start application demonstrates how to use asynchronous messaging to implement a system for purchasing a stock. To purchase a stock, a client application will send a stock request message containing the information about the stock, i.e. ticker symbol, quantity, etc. The client request message will be received by the server where it will perform business processing on the request, for example to determine if the user has sufficient credit to purchase the stock or if the user is even allowed to make the purchase due to existing account restrictions. These are typically external processes as well. Usually the server application will persist state about the request and forward it on to an execute venue where the actual execution of the stock request is performed. In addition, market data for the stock will be sent from the server process to the client. The high level exchange of information is shown below.



Note

To follow this EMS QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.EmsQuickStart`

45.2. Message Destinations

To implement this flow using messaging the following queues and topics will be used. All requests from the client to the server will be sent on the queue named APP.STOCK.REQUEST. Responses to the requests will be sent from the server to the client on a queue unique to each client. In this example the queue name is of the form APP.STOCK.<UserName>, and more specifically is configured to be APP.STOCK.JOE. Market data does not need to be delivered to an individual client as many client applications are interested in this shared information. As such, the server will send market data information on a topic named APP.STOCK.MARKETDATA. The messaging communication between the server and the execution venue is not included as part of the application. An local implementation of the service interface that represents the execution venue is used instead of one based on messaging or another middleware technology. The messaging flow showing the queues and topics used is shown below.



Queues are shown in red and topics in green.

45.3. Messaging Infrastructure

Much of this application mirrors the quickstart that is available with ActiveMQ and you should refer to the NMS QuickStart for the description on how the application is structured in terms of Gateway, Message Data formats, Message Handler, and Message Converters. What this section describes are the specific configuration related to using TIBCO EMS.

The implementations of the gateway interfaces inherit from Spring's helper class `EmsGatewaySupport` in order to get easy access to a [EmsTemplate](#) for sending. The implementation of the `IStockService` interface is shown below

```
public class EmsStockServiceGateway : EmsGatewaySupport, IStockService
{
    private Destination defaultReplyToQueue;

    public Destination DefaultReplyToQueue
    {
        set { defaultReplyToQueue = value; }
    }

    public void Send(TradeRequest tradeRequest)
    {
        EmsTemplate.ConvertAndSendWithDelegate(tradeRequest, delegate(Message message)
        {
            message.ReplyTo = defaultReplyToQueue;
            message.CorrelationID = new Guid().ToString();
            return message;
        });
    }
}
```

The `Send` method is using `EmsTemplate's ConvertAndSendWithDelegate(object obj, MessagePostProcessorDelegate messagePostProcessorDelegate)` method. The anonymous delegate allows you to modify the message properties, such as `ReplyTo` and `CorrelationID` after the message has been converted from an object but before it has been sent. The use of an anonymous delegate allows makes it very easy to apply any post processing logic to the converted message.

The object definition for the `EmsStockServiceGateway` is shown below along with its dependent object definitions of [EmsTemplate](#) and the [ConnectionFactory](#).

```
<object id="ConnectionFactory" type="Spring.Messaging.Ems.Common.EmsConnectionFactory, Spring.Messaging.Ems">
  <constructor-arg index="0" value="tcp://localhost:7222"/>
</object>

<!-- EMS based implementation of technology neutral IStockServiceGateway -->
<object name="StockServiceGateway" type="Spring.EmsQuickStart.Client.Gateways.EmsStockServiceGateway, Spring.EmsQuickStart">
  <property name="EmsTemplate" ref="EmsTemplate"/>
  <property name="DefaultReplyToQueue">
    <object type="TIBCO.EMS.Queue, TIBCO.EMS">
      <constructor-arg value="APP.STOCK.JOE"/>
    </object>
  </property>
</object>

<object name="EmsTemplate" type="Spring.Messaging.Ems.Core.EmsTemplate, Spring.Messaging.Ems">
  <property name="ConnectionFactory" ref="ConnectionFactory"/>
  <property name="DefaultDestinationName" value="APP.STOCK.REQUEST"/>
  <property name="MessageConverter" ref="XmlMessageConverter"/>
</object>
```

In this example the `Spring.Messaging.Ems.Common.EmsConnectionFactory` connection factory was used. It would be more efficient resource wise to use Spring's `CachingConnectionFactory` wrapper class so that

connections will not be open and closed for each message send as well as allowing for the caching of other intermediate EMS API objects such as sessions and message producers.

A similar configuration is used on the server to configure the class `Spring.EmsQuickStart.Server.Gateways.MarketDataServiceGateway` that implements the `IMarketDataService` interface.

Since the client is also a consumer of messages, on the topic `APP.STOCK.MARKETDATA` and the queue `APP.STOCK.JOE` (for Trader Joe!), two message listener containers are defined as shown below.

```
<ems:listener-container connection-factory="ConnectionFactory">
  <ems:listener ref="MessageListenerAdapter" destination="APP.STOCK.JOE" />
  <ems:listener ref="MessageListenerAdapter" destination="APP.STOCK.MARKETDATA" pubsub-domain="true"/>
</ems:listener-container>
```

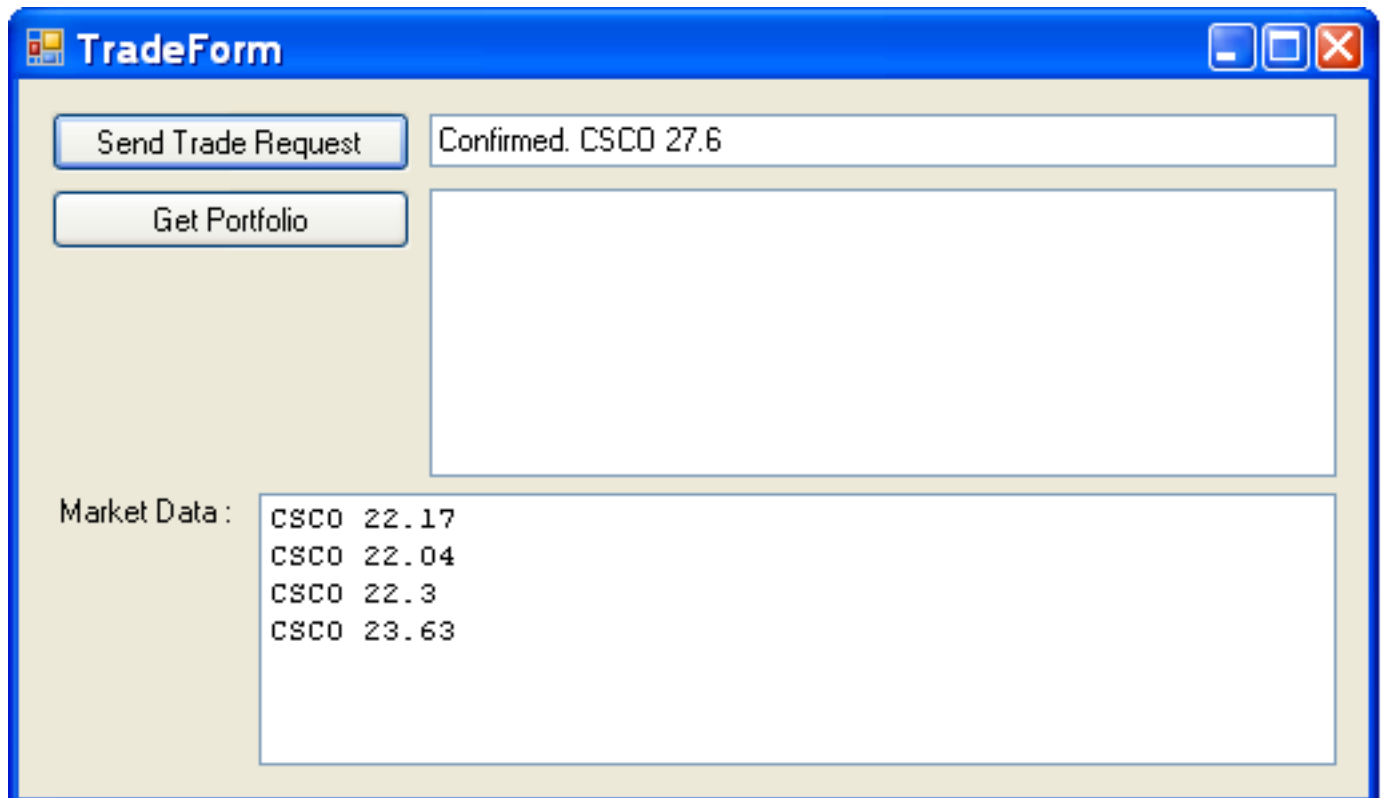
Refer to the messages reference docs for all the available attributes to configure the container and also the section on registering the EMS schema with Spring..

On the server we define a message listener container for the queue `APP.STOCK.REQUEST` but set the concurrency property to 10 so that 10 threads will be consuming messages from the queue.

```
<ems:listener-container connection-factory="ConnectionFactory" concurrency="10">
  <ems:listener ref="MessageListenerAdapter" destination="APP.STOCK.REQUEST" />
</ems:listener-container>
```

45.4. Running the application

To run both the client and server make sure that you select 'Multiple Startup Projects' within VS.NET. The GUI has a button to make a hardcoded trade request and show confirmation in a text box. A text area is used to display the market data. There is a 'Get Portfolio' button that is not implemented at the moment. A picture of the GUI after it has been running for a while and trade has been sent and responded to is shown below



Chapter 46. MSMQ QuickStart

46.1. Introduction

The MSMQ quick start application demonstrates how to use asynchronous messaging to implement a system for purchasing a stock. It follows the same basic approach as in the NMS QuickStart but is adapted as needed for use with MSMQ. Please read the introduction in that chapter to get an overview of the system.

When there is direct overlap in functionality between the MSMQ and NMS quickstart a reference to the appropriate section in the NMS QuickStart documentation is given.



Note

To follow this MSMQ QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.MsmqQuickStart`

46.2. Message Destinations

To communicate between the client and server a pair of queues will be used. Messages sent from the client to the server will use the transactional queue named `.\Private$\request.txqueue`. Messages sent from the server to the client will use the transactional queue `.\Private$\response.joe.txqueue`. The queue for messages that cannot be processed, so called 'poison messages' will be sent to the queue `.\Private$\dead.txqueue`. You can create these queues using the computer management administration console. Private queues are used to simplify the application setup requirements.



Note

You must create the queues mentioned previously using standard Windows Computer Management console to manage MSMQ. This article [http://www.worldofasp.net/tut/MSMQ/Basic_Introduction_about_MSMQ_in_NET_Framework_98.aspx] covers the basics of creating the queues in the management console.

Since MSMQ does not natively support the publish-subscribe messaging style as in other messaging systems, Apache MQ, IBM Websphere MQ, TIBCO EMS, the market data information is sent on the same queue as the responses from the server to the client for trade requests..

46.3. Gateways

The gateway interfaces are the same as those described in the NMS QuickStart here.

46.4. Message Data

TradeRequest and TradeResponse messages are defined using XML Schema and classes are generated from that schema. This is the same approach as described in more details in the NMS QuickStart here.

An important difference in the types of message data formats supported 'out-of-the-box' with Apache, IBM, TIBCO as compared to Microsoft MSMQ is the latter support sending a hashtable data structure. As a result, the hashtable that was used to send market data information from the server to the client was changed to be of type `System.String` in the MSMQ example.

46.5. Message Handlers

The message handlers are the same as used in the NMS QuickStart here, aside from the change of the hashtable data structure to a string. This is an important benefit of enforcing a separation between the messaging specific classes and the business processing layer.

46.6. MessageConverters

The message converter used is `Spring.Messaging.Support.Converters.XmlMessageConverter`. It is configured by specifying the data types that will be send and received. Here is a configuration example for types generated from the XML Schema and a plain string.

```
<object id="xmlMessageConverter" type="Spring.Messaging.Support.Converters.XmlMessageConverter, Spring.Messaging">
  <property name="TargetTypes">
    <list>
      <value>Spring.MsmqQuickStart.Common.Data.TradeRequest, Spring.MsmqQuickStart.Common</value>
      <value>Spring.MsmqQuickStart.Common.Data.TradeResponse, Spring.MsmqQuickStart.Common</value>
      <value>System.String, mscorlib</value>
    </list>
  </property>
</object>
```

46.7. Messaging Infrastructure

The implementations of the gateway interfaces inherit from Spring's helper class `MessageQueueGatewaySupport` in order to get easy access to a `MessageQueueTemplate` for sending. The implementation of the `IStockService` interface is shown below

```
public class MsmqStockServiceGateway : MessageQueueGatewaySupport, IStockService
{
    private Random random = new Random();

    private string defaultResponseQueueObjectName;

    public string DefaultResponseQueueObjectName
    {
        set { defaultResponseQueueObjectName = value; }
    }

    public void Send(TradeRequest tradeRequest)
    {
        MessageQueueTemplate.ConvertAndSend(tradeRequest, delegate(Message message)
        {
            message.ResponseQueue = GetResponseQueue();
            message.AppSpecific = random.Next();
            return message;
        });
    }

    private MessageQueue GetResponseQueue()
    {
        return MessageQueueFactory.CreateMessageQueue(defaultResponseQueueObjectName);
    }
}
```

The `Send` method is using `MessageQueueTemplate's ConvertAndSend(object obj, MessagePostProcessorDelegate messagePostProcessorDelegate)` method. The anonymous delegate allows you to modify the message properties, such as `ResponseQueue` and `AppSpecific` after the message has been converted from an object but before it has been sent. The use of an anonymous delegate allows makes it very easy to apply any post processing logic to the converted message.

The configuration for `MsmqStockServiceGateway` and all its dependencies is shown below, highlighting important dependency links.

```
<object name="stockServiceGateway" type="Spring.MsmqQuickStart.Client.Gateways.MsmqStockServiceGateway, Spring.MsmqQuickStart"
  <property name="MessageQueueTemplate" ref="messageQueueTemplate"/>
  <property name="DefaultResponseQueueObjectName" value="responseTxQueue"/>
</object>

<object id="messageQueueTemplate" type="Spring.Messaging.Core.MessageQueueTemplate, Spring.Messaging">
  <property name="DefaultMessageQueueObjectName" value="requestTxQueue"/>
  <property name="MessageConverterObjectName" value="xmlMessageConverter"/>
</object>

<object id="xmlMessageConverter" type="Spring.Messaging.Support.Converters.XmlMessageConverter, Spring.Messaging">
  <property name="TargetTypes">
    <list>
      <value>Spring.MsmqQuickStart.Common.Data.TradeRequest, Spring.MsmqQuickStart.Common</value>
      <value>Spring.MsmqQuickStart.Common.Data.TradeResponse, Spring.MsmqQuickStart.Common</value>
      <value>System.String, mscorlib</value>
    </list>
  </property>
</object>

<object id="requestTxQueue" type="Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging">
  <property name="Path" value=".\Private$\request.txqueue"/>
  <property name="MessageReadPropertyFilterSetAll" value="true"/>
</object>

<object id="responseTxQueue" type="Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging">
  <property name="Path" value=".\Private$\response.joe.txqueue"/>
  <property name="MessageReadPropertyFilterSetAll" value="true"/>
</object>
```

Since the client also needs to listen to incoming messages on the `responseTxQueue`, a `TransactionalMessageListenerContainer` is configured. The configuration for the message listener container and all its dependencies is shown below, highlighting important dependency links.

```
<!-- MSMQ Transaction Manager -->
<object id="messageQueueTransactionManager" type="Spring.Messaging.Core.MessageQueueTransactionManager, Spring.Messaging"/>

<!-- Message Listener Container that uses MSMQ transactional for receives -->
<object id="transactionalMessageListenerContainer" type="Spring.Messaging.Listener.TransactionalMessageListenerContainer, Sp
  <property name="MessageQueueObjectName" value="responseTxQueue"/>
  <property name="PlatformTransactionManager" ref="messageQueueTransactionManager"/>
  <property name="MessageListener" ref="messageListenerAdapter"/>
  <property name="MessageTransactionExceptionHandler" ref="sendToQueueExceptionHandler"/>

</object>

<!-- Delegate to plain .NET object for message handling -->
<object id="messageListenerAdapter" type="Spring.Messaging.Listener.MessageListenerAdapter, Spring.Messaging">
  <property name="HandlerObject" ref="stockAppHandler"/>
  <property name="DefaultHandlerMethod" value="Handle"/>
  <property name="MessageConverterObjectName" value="xmlMessageConverter"/>
</object>

<object id="sendToQueueExceptionHandler" type="Spring.Messaging.Listener.SendToQueueExceptionHandler, Spring.Messaging">
  <property name="MessageQueueObjectName" value="deadTxQueue"/>
</object>

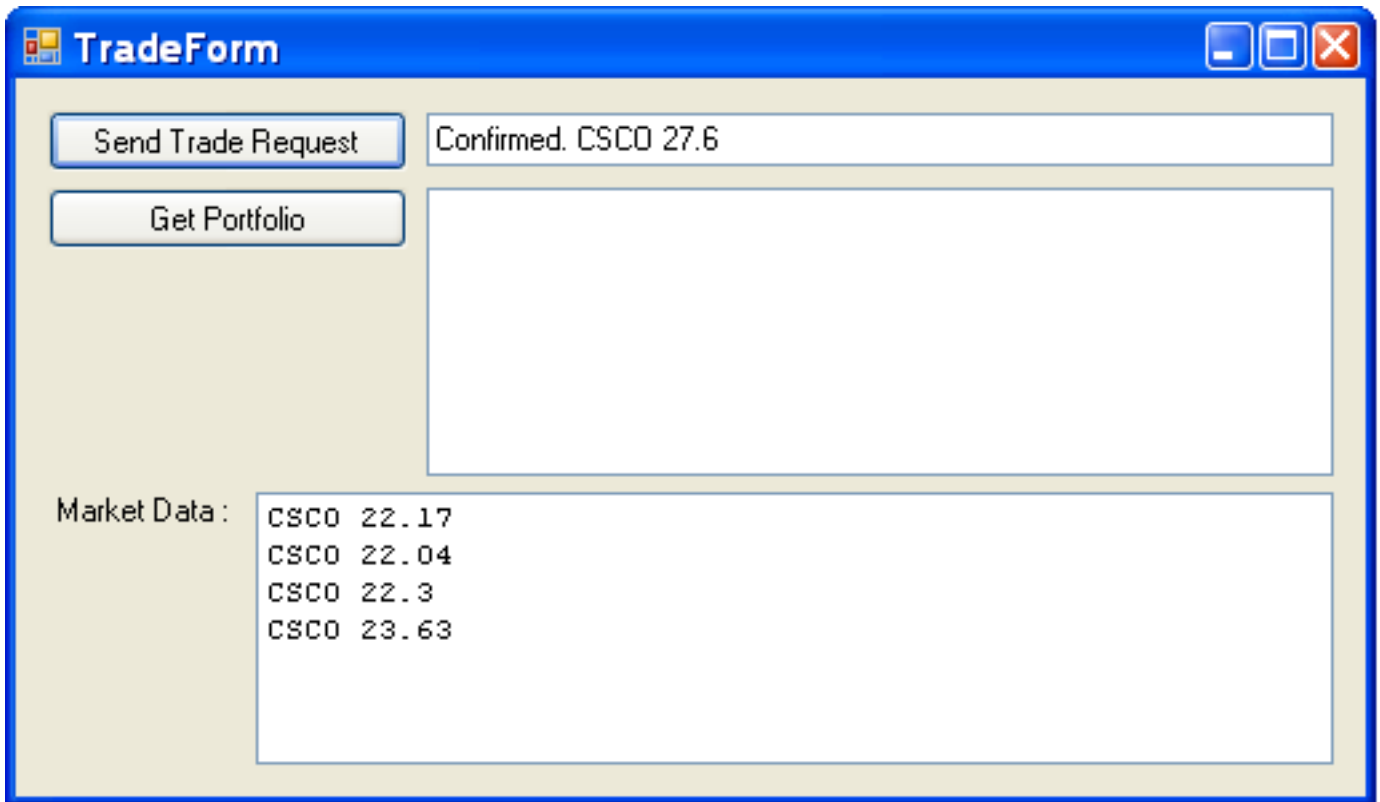
<object id="deadTxQueue" type="Spring.Messaging.Support.MessageQueueFactoryObject, Spring.Messaging">
  <property name="Path" value=".\Private$\dead.queue"/>
  <property name="MessageReadPropertyFilterSetAll" value="true"/>
</object>
```

A similar configuration is used on the server to configure the class `Spring.MsmqQuickStart.Server.Gateways.MarketDataServiceGateway` that implements the `IMarketDataService` interface and a `TransactionalMessageListenerContainer` to process messages on the `requestTxQueue`. You can increase the number of processing thread in the

`TransactionalMessageListenerContainer` by setting the property `MaxConcurrentListeners`, the default value is 1.

46.8. Running the application

To run both the client and server make sure that you select 'Multiple Startup Projects' within VS.NET. The GUI has a button to make a hard coded trade request and show confirmation in a text box. A text area is used to display the market data. There is a 'Get Portfolio' button that is not implemented at the moment. A picture of the GUI after it has been running for a while and trade has been sent and responded to is shown below.



Chapter 47. WCF QuickStart

47.1. Introduction

The WCF quickstart application shows how to configure your WCF services using dependency injection and how to apply AOP advice to your services. It is based on the same interfaces used in the portable service abstractions quickstart example that demonstrates similar features for .NET Remoting, Enterprise Services, and ASMX web services. The quickstart example is only available as a VS.NET 2008 solution.

There are two server applications in the solution, one is a web application where the WCF service will be hosted, and the other is a self-hosting console application, `Spring.WcfQuickStart.Server.2008`. The client application is located in `Spring.WcfQuickStart.ClientApp.2008`. To run the solution make sure that all three projects are set to startup.



Note

To follow this Quarts QuickStart load the solution file found in the directory `<spring-install-dir>\examples\Spring\Spring.WcfQuickStart`

47.2. The server side

The service contract is shown below

```
[ServiceContract(Namespace = "http://Spring.WcfQuickStart")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
    [OperationContract]
    double Subtract(double n1, double n2);
    [OperationContract]
    double Multiply(double n1, double n2);
    [OperationContract]
    double Divide(double n1, double n2);
    [OperationContract]
    string GetName();
}
```

and the implementation is straightforward, only adding a property that controls how long each method should sleep. An abbreviated listing of the implementation is shown below

```
public class CalculatorService : ICalculator
{
    private int sleepInSeconds;

    public int SleepInSeconds
    {
        get { return sleepInSeconds; }
        set { sleepInSeconds = value; }
    }

    public double Add(double n1, double n2)
    {
        Thread.Sleep(sleepInSeconds*1000);
        return n1 + n2;
    }

    // other methods omitted for brevity.
}
```

}

47.2.1. WCF Dependency Injection and AOP in self-hosted application

The approach using dynamic proxies is used in the console application inside the `Spring.WcfQuickStart.Server.2008` project. For more information on this approach refer to this section in the reference docs. The configuration of your service is done as you would typically do with Spring, including applying of any AOP advice. The class is hosted inside the console application through the use of Spring's `ServiceHostFactoryObject` exporter. The configuration for the server console application is shown below.

```
<objects xmlns="http://www.springframework.net"
  xmlns:aop="http://www.springframework.net/aop">

  <!-- Service definition -->
  <object id="calculator" singleton="false"
    type="Spring.WcfQuickStart.CalculatorService, Spring.WcfQuickStart.ServerApp">
    <property name="SleepInSeconds" value="1"/>
  </object>

  <object id="serviceOperation" type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut, Spring.Aop">
    <property name="pattern" value="Spring.WcfQuickStart.*"/>
  </object>

  <object id="perfAdvice" type="Spring.WcfQuickStart.SimplePerformanceInterceptor, Spring.WcfQuickStart.ServerApp">
    <property name="Prefix" value="Service Layer Performance"/>
  </object>

  <aop:config>
    <aop:advisor pointcut-ref="serviceOperation" advice-ref="perfAdvice"/>
  </aop:config>

  <!-- host the service object -->
  <object id="calculatorServiceHost" type="Spring.ServiceModel.Activation.ServiceHostFactoryObject, Spring.Services">
    <property name="TargetName" value="calculator" />
  </object>

</objects>
```

Look at the standard WCF configuration section in `App.config` for additional configuration details. In that section you will see that the name of the WCF service corresponds to the name of the service object inside the spring container.

47.2.2. WCF Dependency Injection and AOP in IIS web application

Much of the configuration of the objects is the same as before, the `.svc` file though refers to the name of the service inside the Spring container as well as using Spring's `Spring.ServiceModel.Activation.ServiceHostFactory`. The `.svc` file is shown below.

```
<%@ ServiceHost Language="C#" Debug="true" Service="calculator"
  Factory="Spring.ServiceModel.Activation.ServiceHostFactory" %>
```

47.3. Client access

The project `Spring.WcfQuickStart.ClientApp.2008` is a console application that calls the two WCF services. It creates a client side proxy based on using `ChannelFactory<T>.CreateChannel`. Running the client application produces the following output.

```
--- Press <return> to continue ---
Web Calculator
Add(1, 1) : 2
Divide(11, 2) : 5.5
Multiply(2, 5) : 10
```

```
Subtract(7, 4) : 3

ServerApp Calculator
Add(1, 1) : 2
Divide(11, 2) : 5.5
Multiply(2, 5) : 10
Subtract(7, 4) : 3

--- Press <return> to continue ---
```

Part VIII. Spring.NET for Java developers

This part of the reference documentation is for Java developers who would like a quick orientation to what is different between the Java and .NET versions of the framework.

- Chapter 48, *Spring.NET for Java Developers*

Chapter 48. Spring.NET for Java Developers

48.1. Introduction

This chapter is to help Java developers get their sea legs using Spring.NET. It is not intended to be a comprehensive comparison between .NET and Java. Rather, it highlights the day-to-day differences you will experience when you start to use Spring.NET.

48.2. Beans to Objects

There are some simple name changes, basically everywhere you saw the word 'bean' you will now see the word 'object'. A comparison of a simple Spring configuration file highlights these small name changes. Here is the application.xml file for the sample MovieFinder application in Spring.Java

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="MyMovieLister" class="MovieFinder.MovieLister">
    <property name="finder" ref="MyMovieFinder"/>
  </bean>
  <bean id="MyMovieFinder" class="MovieFinder.SimpleMovieFinder"/>
</beans>
```

Here is the corresponding file in Spring.NET

```
<objects xmlns="http://www.springframework.net"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects-1.1.xsd">
  <object name="MyMovieLister"
    type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
    <property name="movieFinder" ref="MyMovieFinder"/>
  </object>
  <object name="MyMovieFinder"
    type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
</objects>
```

As you can easily see the <beans> and <bean> elements are replaced by <objects> and <object> elements. The class definition in Spring.Java contains the fully qualified class name. The Spring.NET version also contains the fully qualified classname but in addition specifies the name of the assembly where that type is located. This is necessary since .NET does not have a 'classpath' concept. Assembly names in .NET can have up to four parts to describe the exact version.

The other XML Schema elements in Spring.NET are the same as in Spring.Java's DTD except for specifying string based key value pairs. In Java this is represented by the java.util.Properties class and the xml element is name <props> as shown below

```
<property name="people">
  <props>
    <prop key="PennAndTeller">The magic property</prop>
    <prop key="GeorgeCarlin">The funny property</prop>
  </props>
</property>
```

In .NET the analogous class is System.Collections.Specialized.NameValueCollection and is represented by the xml element <name-values>. The listing of the elements also follows the .NET convention of application configuration files using the <add> element with 'key' and 'value' attributes. This is show below

```
<property name="people">
  <name-values>
    <add key="PennAndTeller" value="The magic property"/>
    <add key="GeorgeCarlin" value="The funny property"/>
  </name-values>
</property>
```

48.3. PropertyEditors to TypeConverters

PropertyEditors from the `java.beans` package provide the ability to convert from a string to an instance of a Java class and vice-versa. For example, to set a string array property, a comma delimited string can be used. The Java class that provides this functionality is the appropriately named `StringArrayPropertyEditor`. In .NET, `TypeConverters` from the `System.ComponentModel` namespace provide the same functionality. The type conversion functionality in .NET also allows for `TypeConverters` to be explicitly registered with a data type. This allows for transparent setting of complex object properties. However, some classes in the .NET framework do not support the style of conversion we are used to from Spring.Java, such as setting of a `string[]` with a comma delimited string. The type converter, `StringArrayConverter` in the `Spring.Objects.TypeConverters` namespace is therefore explicitly registered with Spring.NET in order to provide this functionality. As in the case of Spring.Java, Spring.NET allows user defined type converters to be registered. However, if you are creating a custom type in .NET, using the standard .NET mechanisms for type conversion is the preferred approach.

48.4. ResourceBundle-ResourceManager

48.5. Exceptions

Exceptions in Java can either be checked or unchecked. .NET supports only unchecked exceptions. Spring.Java prefers the use of unchecked exceptions, frequently making conversions from checked to unchecked exceptions. In this respect Spring.Java is similar to the default behavior of .NET

48.6. Application Configuration

In Spring.Java it is very common to create an `ObjectFactory` or `ApplicationContext` from an external XML configuration file. This functionality is also provided in Spring.NET. However, in .NET the `System.Configuration` namespace provides support for managing application configuration information. The functionality in this namespace depends on the availability of specially named files: `Web.config` for ASP.NET applications and `<MyExe>.exe.config` for WinForms and console applications. `<MyExe>` is the name of your executable. As part of the compilation process, if you have a file name `App.config` in the root of your project, the compiler will rename the file to `<MyExe>.exe.config` and place it into the runtime executable folder.

These application configuration files are XML based and contain configuration sections that can be referenced by name to retrieve custom configuration objects. In order to inform the .NET configuration system how to create a custom configuration object from one of these sections, an implementation of the interface, `IConfigurationSectionHandler`, needs to be registered. Spring.NET provides two implementations, one to create an `IApplicationContext` from a `<context>` section and another to configure the context with object definitions contained in an `<objects>` section. The `<context>` section is very powerful and expressive. It provides full support for locating all `IResource` via `Uri` syntax and hierarchical contexts without coding or using more verbose XML as would be required in the current version of Spring.Java

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <configSections>
```

```

<sectionGroup name="spring">
  <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
  <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
</sectionGroup>
</configSections>

<spring>

  <context>
    <resource uri="config://spring/objects"/>
  </context>

  <objects>
    <description>An example that demonstrates simple IoC features.</description>
    <object name="MyMovieLister" type="Spring.Examples.MovieFinder.MovieLister, MovieFinder">
      <property name="movieFinder" ref="AnotherMovieFinder"/>
    </object>
    <object name="MyMovieFinder" type="Spring.Examples.MovieFinder.SimpleMovieFinder, MovieFinder"/>
    <!--
    An IMovieFinder implementation that uses a text file as it's movie source...
    -->
    <object name="AnotherMovieFinder" type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, MovieFinder">
      <constructor-arg index="0" value="movies.txt"/>
    </object>
  </objects>

</spring>

</configuration>

```

The `<configSections>` and `<section>` elements are a standard part of the .NET application configuration file. These elements are used to register an instance of `IConfigurationSectionHandler` and associate it with another xml element in the file, in this case the `<context>` and `<objects>` elements.

The following code segment is used to retrieve the `IApplicationContext` from the .NET application configuration file.

```

IApplicationContext ctx
    = ConfigurationUtils.GetSection("spring/context") as IApplicationContext;

```

In order to enforce the usage of the named configuration section `spring/context` the preferred instantiation mechanism is via the use of the registry class `ContextRegistry` as shown below

```

IApplicationContext ctx = ContextRegistry.GetContext();

```

48.7. AOP Framework

48.7.1. Cannot specify target name at the end of interceptorNames for ProxyFactoryObject

When configuring the list of interceptor names on a `ProxyFactoryObject` instance (or object definition), one *cannot* specify the name of the target (i.e. the object being proxied) at the end of the list of interceptor names. This shortcut is valid in Spring Java, where the `ProxyFactoryBean` will automatically detect this, and use the last name in the interceptor names list as the target of the `ProxyFactoryBean`. The following configuration, which would be valid in Spring Java (barring the obvious element name changes), is **not** valid in Spring.NET (so don't do it).

```

<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net">
  <object id="target" type="Spring.Objects.TestObject">
    <property name="name" value="Bingo"/>
  </object>

  <object id="nopInterceptor" type="Spring.Aop.Interceptor.NopInterceptor"/>

```

```
<object id="prototypeTarget" type="Spring.Aop.Framework.ProxyFactoryObject">
  <property name="interceptorNames" value="nopInterceptor,target"/> <!-- not valid! -->
</object>
</objects>
```

In Spring.NET, the `InterceptorNames` property of the `ProxyFactoryObject` can *only* be used to specify the names of interceptors. Use the `TargetName` property to specify the name of the target object that is to be proxied.

The main reason for not supporting exactly the same style of configuration as Spring Java is because this 'feature' is regarded as a legacy holdover from Rod Johnson's initial Spring AOP implementation, and is currently only kept as-is (in Spring Java) for reasons of backward compatibility.

Part IX. Appendices

Appendix A. Classic Spring Usage

This appendix discusses some classic Spring usage patterns as a reference for developers maintaining legacy Spring applications. These usage patterns no longer reflect the recommended way of using these features and the current recommended usage is covered in the respective sections of the reference manual.

A.1. Classic Hibernate Usage

For the currently recommended usage patterns for NHibernate see Section 21.2, “NHibernate”

A.1.1. The `HibernateTemplate`

The basic programming model for templating looks as follows for methods that can be part of any custom data access object or business service. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a `HibernateSessionFactory`. It can get the latter from anywhere, but preferably as an object reference from a Spring IoC container - via a simple `SessionFactory` property setter. The following snippets show a DAO definition in a Spring container, referencing the above defined `SessionFactory`, and an example for a DAO method implementation.

```
<objects>

<object id="CustomerDao" type="Spring.Northwind.Dao.NHibernate.HibernateCustomerDao, Spring.Northwind.Dao.NHibernate">
  <property name="SessionFactory" ref="MySessionFactory"/>
</object>

</objects>
```

```
public class HibernateCustomerDao : ICustomerDao {

    private HibernateTemplate hibernateTemplate;

    public ISessionFactory SessionFactory
    {
        set { hibernateTemplate = new HibernateTemplate(value); }
    }

    public Customer SaveOrUpdate(Customer customer)
    {
        hibernateTemplate.SaveOrUpdate(customer);
        return customer;
    }
}
```

The `HibernateTemplate` class provides many methods that mirror the methods exposed on the `HibernateSession` interface, in addition to a number of convenience methods such as the one shown above. If you need access to the `Session` to invoke methods that are not exposed on the `HibernateTemplate`, you can always drop down to a callback-based approach like so.

```
public class HibernateCustomerDao : ICustomerDao {

    private HibernateTemplate hibernateTemplate;

    public ISessionFactory SessionFactory
    {
        set { hibernateTemplate = new HibernateTemplate(value); }
    }

    public Customer SaveOrUpdate(Customer customer)
    {
        return hibernateTemplate.Execute(
```

```

        delegate(ISession session)
        {
            // do whatever you want with the session....
            session.SaveOrUpdate(customer);
            return customer;
        }) as Customer;
    }
}

```

Using the anonymous delegate is particularly convenient when you would otherwise be passing various method parameter calls to the interface based version of this callback. Furthermore, when using generics, you can avoid the typecast and write code like the following

```

IList<Supplier> suppliers = HibernateTemplate.ExecuteFind<Supplier>(
    delegate(ISession session)
    {
        return session.CreateQuery("from Supplier s where s.Code = ?")
            .SetParameter(0, code)
            .List<Supplier>();
    });

```

where code is a variable in the surrounding block, accessible inside the anonymous delegate implementation.

A callback implementation effectively can be used for any Hibernate data access. `HibernateTemplate` will ensure that `Session` instances are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single step actions like a single Find, Load, SaveOrUpdate, or Delete call, `HibernateTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `HibernateDaoSupport` base class that provides a `SessionFactory` property for receiving a `SessionFactory` and for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```

public class HibernateCustomerDao : HibernateDaoSupport, ICustomerDao
{
    public Customer SaveOrUpdate(Customer customer)
    {
        HibernateTemplate.SaveOrUpdate(customer);
        return customer;
    }
}

```

A.1.2. Implementing Spring-based DAOs without callbacks

As an alternative to using Spring's `HibernateTemplate` to implement DAOs, data access code can also be written in a more traditional fashion, without wrapping the Hibernate access code in a callback, while still respecting and participating in Spring's generic `DataAccessException` hierarchy. The `HibernateDaoSupport` base class offers methods to access the current transactional `Session` and to convert exceptions in such a scenario; similar methods are also available as static helpers on the `SessionFactoryUtils` class. Note that such code will usually pass 'false' as the value of the `DoGetSession(...)` method's 'allowCreate' argument, to enforce running within a transaction (which avoids the need to close the returned `Session`, as its lifecycle is managed by the transaction). Asking for the

```

public class HibernateProductDao : HibernateDaoSupport, IProductDao {

    public Customer SaveOrUpdate(Customer customer)
    {
        ISession session = DoGetSession(false);
        session.SaveOrUpdate(customer);
        return customer;
    }
}

```

This code will *not* translate the Hibernate exception to a generic `DataAccessException`.

A.2. Classic Declarative Transaction Configurations

A.2.1. Declarative Transaction Configuration using `DefaultAdvisorAutoProxyCreator`

Using the `DefaultAdvisorAutoProxyCreator` to configure declarative transactions enables you to refer to the transaction attribute as the pointcut to use for the transactional advice for any object definition defined in the IoC container. The configuration to create a transactional proxy for the manager class shown in the chapter on transaction management is shown below.

```
<!-- The rest of the config file is common no matter how many objects you add -->
<!-- that you would like to have declarative tx management applied to -->

<object id="autoProxyCreator"
        type="Spring.Aop.Framework.AutoProxy.DefaultAdvisorAutoProxyCreator, Spring.Aop">
</object>

<object id="transactionAdvisor"
        type="Spring.Transaction.Interceptor.TransactionAttributeSourceAdvisor, Spring.Data">
    <property name="TransactionInterceptor" ref="transactionInterceptor"/>
</object>

<!-- Transaction Interceptor -->
<object id="transactionInterceptor"
        type="Spring.Transaction.Interceptor.TransactionInterceptor, Spring.Data">
    <property name="TransactionManager" ref="transactionManager"/>
    <property name="TransactionAttributeSource" ref="attributeTransactionAttributeSource"/>
</object>

<object id="attributeTransactionAttributeSource"
        type="Spring.Transaction.Interceptor.AttributesTransactionAttributeSource, Spring.Data">
</object>
```

Granted this is a bit verbose and hard to grok at first sight - however you only need to grok this once as it is 'boiler plate' XML you can reuse across multiple projects. What these object definitions are doing is to instruct Spring's to look for all objects within the IoC configuration that have the `[Transaction]` attribute and then apply the AOP transaction interceptor to them based on the transaction options contained in the attribute. The attribute serves both as a pointcut and as the declaration of transactional option information.

Since this XML fragment is not tied to any specific object references it can be included in its own file and then imported via the `<import>` element. In examples and test code this XML configuration fragment is named `autoDeclarativeServices.xml` See Section 5.2.2.3, "Composing XML-based configuration metadata" for more information.

The classes and their roles in this configuration fragment are listed below

- `TransactionInterceptor` is the AOP advice responsible for performing transaction management functionality.
- `TransactionAttributeSourceAdvisor` is an AOP Advisor that holds the `TransactionInterceptor`, which is the advice, and a pointcut (where to apply the advice), in the form of a `TransactionAttributeSource`.
- `AttributesTransactionAttributeSource` is an implementation of the `ITransactionAttributeSource` interface that defines where to get the transaction metadata defining the transaction semantics (isolation level, propagation behavior, etc) that should be applied to specific methods of specific classes. The transaction metadata is specified via implementations of the `ITransactionAttributeSource` interface. This example shows the use of the

implementation `Spring.Transaction.Interceptor.AttributesTransactionAttributeSource` to obtain that information from standard .NET attributes. By the very nature of using standard .NET attributes, the attribute serves double duty in identifying the methods where the transaction semantics apply. Alternative implementations of `ITransactionAttributeSource` available are `MatchAlwaysTransactionAttributeSource`, `NameMatchTransactionAttributeSource`, or `MethodMapTransactionAttributeSource`.

- `MatchAlwaysTransactionAttributeSource` is configured with a `ITransactionAttribute` instance that is applied to all methods. The shorthand string representation, i.e. `PROPAGATION_REQUIRED` can be used
- `AttributesTransactionAttributeSource` : Use a standard .NET attributes to specify the transactional information. See `TransactionAttribute` class for more information.
- `NameMatchTransactionAttributeSource` allows `ITransactionAttributes` to be matched by method name. The `NameMap` `IDictionary` property is used to specify the mapping. For example

```
<object name="nameMatchTxAttributeSource" type="Spring.Transaction.Interceptor.NameMatchTransactionAttributeSource, Spr
    <property name="NameMap">
        <dictionary>
            <entry key="Execute" value="PROPAGATION_REQUIRES_NEW, -ApplicationException"/>
            <entry key="HandleData" value="PROPAGATION_REQUIRED, -DataHandlerException"/>
            <entry key="Find*" value="ISOLATION_READUNCOMMITTED, -DataHandlerException"/>
        </dictionary>
    </property>
</object>
```

Key values can be prefixed and/or suffixed with wildcards as well as include the full namespace of the containing class.

- `MethodMapTransactionAttributeSource` : Similar to `NameMatchTransactionAttributeSource` but specifies that only fully qualified method names (i.e. `type.method`, `assembly`) and wildcards can be used at the start or end of the method name for matching multiple methods.
- `DefaultAdvisorAutoProxyCreator`: looks for Advisors in the context, and automatically creates proxy objects which are the transactional wrappers

Refer to the following section for a more convenient way to achieve the same goal of declarative transaction management using attributes.

A.2.2. Declarative Transactions using TransactionProxyFactoryObject

The `TransactionProxyFactoryObject` is easier to use than a `ProxyFactoryObject` for most cases since the transaction interceptor and transaction attributes are properties of this object. This removes the need to declare them as separate objects. Also, unlike the case with the `ProxyFactoryObject`, you do not have to give fully qualified method names, just the normal 'short' method name. Wild card matching on the method name is also allowed, which in practice helps to enforce a common naming convention for the methods of your DAOs. The example from chapter 5 is shown here using a `TransactionProxyFactoryObject`.

```
<object id="testObjectManager"
    type="Spring.Transaction.Interceptor.TransactionProxyFactoryObject, Spring.Data">

    <property name="PlatformTransactionManager" ref="adoTransactionManager"/>
    <property name="Target">
        <object type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
            <property name="TestObjectDao" ref="testObjectDao"/>
        </object>
    </property>
</object>
```

```

<property name="TransactionAttributes">
  <name-values>
    <add key="Save*" value="PROPAGATION_REQUIRED"/>
    <add key="Delete*" value="PROPAGATION_REQUIRED"/>
  </name-values>
</property>
</object>

```

Note the use of an inner object definition for the target which will make it impossible to obtain an unproxied reference to the `TestObjectManager`.

As can be seen in the above definition, the `TransactionAttributes` property holds a collection of name/value pairs. The key of each pair is a method or methods (a `*` wildcard ending is optional) to apply transactional semantics to. Note that the method name is not qualified with a package name, but rather is considered relative to the class of the target object being wrapped. The value portion of the name/value pair is the `TransactionAttribute` itself that needs to be applied. When specifying it as a string value as in this example, it's in String format as defined by `TransactionAttributeConverter`. This format is:

```
PROPAGATION_NAME, ISOLATION_NAME, readOnly, timeout_NNNN, +Exception1, -Exception2
```

Note that the only mandatory portion of the string is the propagation setting. The default transactions semantics which apply are as follows:

- Exception Handling: All exceptions thrown trigger a rollback.
- Transactions are read/write
- Isolation Level: `TransactionDefinition.ISOLATION_DEFAULT`
- Timeout: `TransactionDefinition.TIMEOUT_DEFAULT`

Multiple rollback rules can be specified here, comma-separated. A `-` prefix forces rollback; a `+` prefix specifies commit. Under the covers the `IDictionary` of name value pairs will be converted to an instance of `NameMatchTransactionAttributeSource`

The string used for `PROPAGATION_NAME` are those defined on the `Spring.Transaction.TransactionPropagation` enumeration, namely `Required`, `Supports`, `Mandatory`, `RequiresNew`, `NotSupported`, `Never`, `Nested`. The string used for `ISOLATION_NAME` are those defined on the `System.Data.IsolationLevel` enumeration, namely `ReadCommitted`, `ReadUncommitted`, `RepeatableRead`, `Serializable`.

The `TransactionProxyFactoryObject` allows you to set optional "pre" and "post" advice, for additional interception behavior, using the "PreInterceptors" and "PostInterceptors" properties. Any number of pre and post advices can be set, and their type may be `Advisor` (in which case they can contain a pointcut), `MethodInterceptor` or any advice type supported by the current Spring configuration (such as `ThrowsAdvice`, `AfterReturningAdvice` or `BeforeAdvice`, which are supported by default.) These advices must support a shared-instance model. If you need transactional proxying with advanced AOP features such as stateful mixins, it's normally best to use the generic `ProxyFactoryObject`, rather than the `TransactionProxyFactoryObject` convenience proxy creator.

A.2.3. Concise proxy definitions

Using abstract object definitions in conjunction with a `TransactionProxyFactoryObject` provides you a more concise means to reuse common configuration information instead of duplicating it over and over again with a definition of a `TransactionProxyFactoryObject` per object. Objects that are to be proxied typically have the same

pattern of method names, Save*, Find*, etc. This commonality can be placed in an abstract object definition, which other object definitions refer to and change only the configuration information that is different. An abstract object definition is shown below

```
<object id="txProxyTemplate" abstract="true"
      type="Spring.Transaction.Interceptor.TransactionProxyFactoryObject, Spring.Data">

  <property name="PlatformTransactionManager" ref="adoTransactionManager"/>

  <property name="TransactionAttributes">
    <name-values>
      <add key="Save*" value="PROPAGATION_REQUIRED"/>
      <add key="Delete*" value="PROPAGATION_REQUIRED"/>
    </name-values>
  </property>
</object>
```

Subsequent definitions can refer to this 'base' configuration as shown below

```
<object id="testObjectManager" parent="txProxyTemplate">
  <property name="Target">
    <object type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
      <property name="TestObjectDao" ref="testObjectDao"/>
    </object>
  </property>
</object>
```

A.2.4. Declarative Transactions using ProxyFactoryObject

Using the general ProxyFactoryObject to declare transactions gives you a great deal of control over the proxy created since you can specify additional advice, such as for logging or performance. Based on the example shown previously a sample configuration using ProxyFactoryObject is shown below

```
<object id="testObjectManagerTarget" type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
  <property name="TestObjectDao" ref="testObjectDao"/>
</object>

<object id="testObjectManager" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

  <property name="Target" ref="testObjectManagerTarget"/>
  <property name="ProxyInterfaces">
    <value>Spring.Data.ITestObjectManager</value>
  </property>
  <property name="InterceptorNames">
    <value>transactionInterceptor</value>
  </property>

</object>
```

The ProxyFactoryObject will create a proxy for the Target, i.e. a TestObjectManager instance. An inner object definition could also have been used such that it would make it impossible to obtain an unproxied object from the container. The interceptor name refers to the following definition.

```
<object id="transactionInterceptor" type="Spring.Transaction.Interceptor.TransactionInterceptor, Spring.Data">

  <property name="TransactionManager" ref="adoTransactionManager"/>

  <!-- note do not have converter from string to this property type registered -->
  <property name="TransactionAttributeSource" ref="methodMapTransactionAttributeSource"/>
</object>

<object name="methodMapTransactionAttributeSource"
      type="Spring.Transaction.Interceptor.MethodMapTransactionAttributeSource, Spring.Data">
  <property name="MethodMap">
    <dictionary>
      <entry key="Spring.Data.TestObjectManager.SaveTwoTestObjects, Spring.Data.Integration.Tests"
            value="PROPAGATION_REQUIRED"/>
    </dictionary>
  </property>
</object>
```

```
<entry key="Spring.Data.TestObjectManager.DeleteTwoTestObjects, Spring.Data.Integration.Tests"
      value="PROPAGATION_REQUIRED" />
</dictionary>
</property>
</object>
```

The transaction options for each method are specified using a dictionary containing the class name + method name, assembly as the key and the value is of the form

- <Propagation Behavior>, <Isolation Level>, <ReadOnly>, -Exception, +Exception

All but the propagation behavior are optional. The + and - are used in front of the name of an exception. Minus indicates to rollback if the exception is thrown, the Plus indicates to commit if the exception is thrown.

Appendix B. XML Schema-based configuration

B.1. Introduction

This appendix details the use of XML Schema-based configuration in Spring.

The 'classic' `<object/>`-based schema is good, but its generic-nature comes with a price in terms of configuration overhead. Creating a custom XML Schema-based configuration makes Spring XML configuration files substantially clearer to read. In addition, it allows you to express the intent of an object definition.

The key thing to remember is that creating custom schema tags work best for infrastructure or integration objects: for example, AOP, collections, transactions, integration with 3rd-party frameworks, etc., while the existing object tags are best suited to application-specific objects, such as DAOs, service layer objects, etc.

Please note the fact that the XML configuration mechanism is totally customisable and extensible. This means you can write your own domain-specific configuration tags that would better represent your application's domain; the process involved in doing so is covered in the appendix entitled Appendix C, *Extensible XML authoring*.

B.2. XML Schema-based configuration

B.2.1. Referencing the schemas

As a reminder, you reference the standard objects schema as shown below

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/schema/objects/spring-objects-1.1"
--
<!-- <object/> definitions here -->
</objects>
```

Note



The 'xsi:schemaLocation' fragment is not actually required, but can be included to reference a local copy of a schema (which can be useful during development) and assumes the XML editor will look to that location and load the schema.

The above Spring XML configuration fragment is boilerplate that you can copy and paste (!) and then plug `<object/>` definitions into like you have always done. However, the entire point of using custom schema tags is to make configuration easier.

The rest of this chapter gives an overview of custom XML Schema based configuration that are included with the release.

Note



As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

B.2.2. The `tx` (transaction) schema

The `tx` tags deal with configuring objects in Spring's comprehensive support for transactions. These tags are covered in the chapter entitled Chapter 17, *Transaction management*.



Tip

You are strongly encouraged to look at the '`spring-tx-1.1.xsd`' file that ships with the Spring distribution. This file is (of course), the XML Schema for Spring's transaction configuration, and covers all of the various tags in the `tx` namespace, including attribute defaults and suchlike. This file is documented inline, and thus the information is not repeated here in the interests of adhering to the DRY (Don't Repeat Yourself) principle.

In the interest of completeness, to use the tags in the `tx` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the `tx` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
<object xmlns="http://www.springframework.net"
  xmlns:aop="http://www.springframework.net/aop"
  xmlns:tx="http://www.springframework.net/tx">

  <!-- <object/> definitions here -->

  <!-- <tx/> transaction definitions here -->

  <!-- <aop/> AOP definitions here -->

</object>
```



Note

Often when using the tags in the `tx` namespace you will also be using the tags from the `aop` namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the `aop` schema so that the tags in the `aop` namespace are available to you.

You will also need to configure the AOP and Transaction namespace parsers in the main .NET application configuration file as shown below



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context, typeAliases, etc not shown for brevity -->
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
      <parser type="Spring.Transaction.Config.TxNamespaceParser, Spring.Data" />
    </parsers>
  </spring>
</configuration>
```

```

    </parsers>
  </spring>

</configuration>

```

B.2.3. The `aop` schema

The `aop` tags deal with configuring all things AOP in Spring. These tags are comprehensively covered in the chapter entitled Chapter 13, *Aspect Oriented Programming with Spring.NET*.

In the interest of completeness, to use the tags in the `aop` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the `aop` namespace are available to you.

```

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
         xmlns:aop="http://www.springframework.net/aop">

  <!-- <object/> definitions here -->

  <!-- <aop/> AOP definitions here -->

</objects>

```

You will also need to configure the AOP namespace parser in the main .NET application configuration file as shown below



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```

<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context, typeAliases, etc not shown for brevity -->
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
    </parsers>
  </spring>

</configuration>

```

B.2.4. The `db` schema

The `db` tags deal with creating `IDbProvider` instances for a given database client library. The following snippet references the correct schema so that the tags in the `db` namespace are available to you. The tags are comprehensively covered in the chapter entitled Chapter 19, *DbProvider*.

```

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
         xmlns:db="http://www.springframework.net/db">

  <!-- <object/> definitions here -->

  <!-- <db/> database definitions here -->


```

```
</objects>
```

You will also need to configure the Database namespace parser in the main .NET application configuration file as shown below



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context, typeAliases, etc not shown for brevity -->
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
    </parsers>
  </spring>

</configuration>
```

B.2.5. The remoting schema

The remoting tags are for use when you want to export an existing POCO object as a .NET remoted object or to create a client side .NET remoting proxy. The tags are comprehensively covered in the chapter Chapter 25, *.NET Remoting*

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
  xmlns:r="http://www.springframework.net/remoting">

  <!-- <object/> definitions here -->

  <!-- <r/> remoting definitions here -->

</objects>
```

You will also need to configure the remoting namespace parser in the main .NET application configuration file as shown below



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context, typeAliases, etc not shown for brevity -->
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>
```

```

<spring>
  <parsers>
    <parser type="Spring.Remoting.Config.RemotingNamespaceParser, Spring.Services" />
  </parsers>
</spring>

</configuration>

```

B.2.6. The `nms` messaging schema

The `nms` tags are for use when you want to configure Spring's messaging support. The tags are comprehensively covered in the chapter Chapter 29, *Message Oriented Middleware - Apache ActiveMQ and TIBCO EMS*

```

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
  xmlns:r="http://www.springframework.net/nms">

  <!-- <object/> definitions here -->

  <!-- <nms/> remoting definitions here -->

</objects>

```

You will also need to configure the remoting namespace parser in the main .NET application configuration file as shown below



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```

<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context, typeAliases, etc not shown for brevity -->
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Messaging.Nms.Config.NmsNamespaceParser, Spring.Messaging.Nms" />
    </parsers>
  </spring>

</configuration>

```

B.2.7. The `validation` schema

The `validation` tags are for use when you want to define `IValidator` object instances. The tags are comprehensively covered in the chapter Chapter 12, *Validation Framework*

```

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
  xmlns:v="http://www.springframework.net/validation">

  <!-- <object/> definitions here -->

  <!-- <v/> validation definitions here -->

</objects>

```

You will also need to configure the validation namespace parser in the main .NET application configuration file as shown below



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context, typeAliases, etc not shown for brevity -->
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Validation.Config.ValidationNamespaceParser, Spring.Core" />
    </parsers>
  </spring>

</configuration>
```

B.2.8. The objects schema

Last but not least we have the tags in the objects schema. Examples of the various tags in the objects schema are not shown here because they are quite comprehensively covered in the section entitled Section 5.3.2, “Dependencies and configuration in detail” (and indeed in that entire chapter).

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/schema/objects/spring-objects-1.1.xsd">

  <object id="foo" class="X.Y.Foo, X">
    <property name="name" value="Rick"/>
  </object>

</objects>
```

B.3. Setting up your IDE

To setup VS.NET to provide intellisense while editing XML file for your custom XML schemas you will need to copy your XSD files to an appropriate VS.NET directory. Refer to the following chapter for details, Chapter 34, *Visual Studio.NET Integration*

For SharpDevelop, follow the directions on the ["Editing XML"](#) product documentation.

Appendix C. Extensible XML authoring

C.1. Introduction

Spring supports adding custom schema-based extensions to the basic Spring XML format for defining and configuring objects. This section is devoted to detailing how you would go about writing your own custom XML object definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled Appendix B, *XML Schema-based configuration*.

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

1. Authoring an XML schema to describe your custom element(s).
2. Coding a custom `INamespaceParser` implementation (this is an easy step, don't worry).
3. Coding one or more `IOBJECTDefinitionParser` implementations (this is where the real work is done).
4. Registering the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type `Regex` (from the `System.Text.RegularExpressions` namespace) in an easy manner. When we are done, we will be able to define object definitions of type `Regex` like this:

```
<myns:regex id="regex"
  pattern="(^\\d{5}$)|(\\d{5}-\\d{4}$)"
  options="Compiled"/>
```

C.2. Authoring the schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we'll use to configure `Regex` objects.

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema id="myns"
  xmlns="http://www.mycompany.com/schema/myns"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:objects="http://www.springframework.net"
  xmlns:vs="http://schemas.microsoft.com/Visual-Studio-Intellisense"
  targetNamespace="http://www.mycompany.com/schema/myns"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  vs:friendlyname="Spring Regex Configuration" vs:ishtmlschema="false"
  vs:iscasesensitive="true" vs:requireattributequotes="true"
  vs:defaultnamespacequalifier="" vs:defaultnsprefix=""
  >

  <xsd:import namespace="http://www.springframework.net"/>

  <xsd:element name="regex">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="objects:identifiedType">
          <xsd:attribute name="pattern" type="xsd:string" use="required"/>
          <xsd:attribute name="options" type="xsd:string" use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

        </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:element>

</xsd:schema>

```

The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an `id` attribute that will be used as the object identifier in the container). We are able to use this attribute because we imported the Spring-provided 'objects' namespace. The `vs:` prefixed elements are for better integration with intellisense in VS.NET.

The above schema will be used to configure `Regex` objects, directly in an XML application context file using the `<myns:regex/>` element.

```

<myns:regex id="usZipCodeRegex"
    pattern="(^\\d{5}$)|(\\d{5}-\\d{4}$)"
    options="Compiled"/>

```

Note that after we've created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we're just creating an object in the container, identified by the name 'usZipCodeRegex' of type `Regex`, with a couple of constructor arguments set.

```

<object id="usZipCodeRegex" type="System.Text.RegularExpressions.Regex, System">
    <constructor-arg name="pattern" value="(^\\d{5}$)|(\\d{5}-\\d{4}$)"/>
    <constructor-arg name="options" value="Compiled"/>
</object>

```



Note

The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use intellisense to have a user choose between several configuration options defined in the enumeration. The schema for creating `IDbProvider` instances shows the use of XSD enumerations.

C.3. Coding a `INamespaceParser`

In addition to the schema, we need an `INamespaceParser` that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The `INamespaceParser` should in our case take care of the parsing of the `myns:regex` element.

The `INamespaceParser` interface is pretty simple in that it features just two methods:

- `Init()` - allows for initialization of the `INamespaceParser` and will be called by Spring before the handler is used
- `IObjectDefinition Parse(Element, ParserContext)` - called when Spring encounters a top-level element (not nested inside a object definition or a different namespace). This method can register object definitions itself and/or return a object definition.

Although it is perfectly possible to code your own `INamespaceParser` for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single object definition (as in our case, where a single `<myns:regex/>` element results in a single `Regex` object definition). Spring features a number of convenience classes that support this scenario. In this example, we'll make use the `NamespaceParserSupport` class:

```

using Spring.Objects.Factory.Xml;

```



```

namespace CustomNamespace
{
    [NamespaceParser(
        Namespace = "http://www.mycompany.com/schema/myns",
        SchemaLocationAssemblyHint = typeof(MyNamespaceParser),
        SchemaLocation = "/CustomNamespace/myns.xsd"
    )]
    public class MyNamespaceParser : NamespaceParserSupport
    {
        public override void Init()
        {
            RegisterObjectDefinitionParser("regex", new RegexObjectDefinitionParser());
        }
    }
}

```

Notice that there isn't actually a whole lot of parsing logic in this class. Indeed... the `NamespaceParserSupport` class has a built in notion of delegation. It supports the registration of any number of `IObjectDefinitionParser` instances, to which it will delegate to when it needs to parse an element in it's namespace. This clean separation of concerns allows an `INamespaceParser` to handle the orchestration of the parsing of *all* of the custom elements in it's namespace, while delegating to `IObjectDefinitionParsers` to do the grunt work of the XML parsing; this means that each `IObjectDefinitionParser` will contain just the logic for parsing a single custom element, as we can see in the next step.

To help in the registration of the parser for this namespace, the `NamespaceParser` attribute is used to map the XML namespace string, i.e. `http://www.mycompany.com/schema/myns`, to the location of the XML Schema file as an embedded assembly resource.

C.4. Coding an `IObjectDefinitionParser`

A `IObjectDefinitionParser` will be used if the `INamespaceParser` encounters an XML element of the type that has been mapped to the specific object definition parser (which is 'regex' in this case). In other words, the `IObjectDefinitionParser` is responsible for parsing *one* distinct top-level XML element defined in the schema. In the parser, we'll have access to the XML element (and thus it's subelements too) so that we can parse our custom XML content, as can be seen in the following example:

```

using System;
using System.Text.RegularExpressions;
using System.Xml;
using Spring.Objects.Factory.Support;
using Spring.Objects.Factory.Xml;
using Spring.Util;

namespace CustomNamespace
{
    public class RegexObjectDefinitionParser : AbstractSimpleObjectDefinitionParser { ❶

        protected override Type GetObjectType(XmlElement element)
        {
            return typeof (Regex); ❷
        }

        protected override void DoParse(XmlElement element, ObjectDefinitionBuilder builder)
        {
            // this will never be null since the schema explicitly requires that a value be supplied
            string pattern = element.GetAttribute("pattern");
            builder.AddConstructorArg(pattern);

            // this however is an optional property
            string options = element.GetAttribute("options");
            if (StringUtil.HasText(options))
            {
                RegexOptions regexOptions = (RegexOptions)Enum.Parse(typeof (RegexOptions), options);
            }
        }
    }
}

```

```

        builder.AddConstructorArg(regexOptions);
    }
}

protected override bool ShouldGenerateIdAsFallback
{
    get { return true; }
}
}

```

- ❶ We use the Spring-provided `AbstractSingleObjectDefinitionParser` to handle a lot of the basic grunt work of creating a *single* `IObjectDefinition`.
- ❷ We supply the `AbstractSingleObjectDefinitionParser` superclass with the type that our single `IObjectDefinition` will represent.

In this simple case, this is all that we need to do. The creation of our single `IObjectDefinition` is handled by the `AbstractSingleObjectDefinitionParser` superclass, as is the extraction and setting of the object definition's unique identifier. The property `ShouldGenerateIdAsFallback` will generate a throw-away object id incase one is not specified, this is useful when nesting object definitions.

C.5. Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom `INamespaceParser` using a special configuration section handler. The location of the XML Schema in this example has been directly associated with the parser though the use of the `Namespace` attribute.

C.5.1. NamespaceParsersSectionHandler

The custom configuration section handler is of the type `Spring.Context.Support.NamespaceParsersSectionHandler` and is registered with .NET in the normal manner. The custom configuration section will simply point to the `INamespaceParser` implementation that has the `Namespace` attribute. For our example, we need to write the following:

```

<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="CustomNamespace.MyNamespaceParser, CustomNamespace" />
    </parsers>
  </spring>

</configuration>

```

C.6. Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom' extensions that Spring provides straight out of the box. Find below an example of using the custom `<regex/>` element developed in the previous steps in a Spring XML configuration file.

```

<?xml version="1.0" encoding="utf-8" ?>

```

```
<objects xmlns="http://www.springframework.net"
  xmlns:myns="http://www.mycompany.com/schema/myns">

  <!-- as a top level object definition -->
  <myns:regex id="usZipCodeRegex"
    pattern="(^\d{5}$)|(^^\d{5}-\d{4}$)" />

  <object id="jobDetailTemplate" abstract="true">
    <property name="regex">
      <!-- as an inner object definition -->
      <myns:regex pattern="(^\d{5}$)|(^^\d{5}-\d{4}$)"
        options="Compiled" />
    </property>
  </object>

</objects>
```

C.7. Further Resources

Find below links to further resources concerning XML Schema and the extensible XML support described in this chapter.

- The [XML Schema Part 1: Structures Second Edition](#)
- The [XML Schema Part 2: Datatypes Second Edition](#)

Appendix D. Spring.NET's `spring-objects.xsd`

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns="http://www.springframework.net" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:vs="http://schemas.microsoft.com/2003/01/XMLSchema" >
  <xs:annotation>
    <xs:documentation>
      Spring Objects XML Schema Definition
      Based on Spring Beans DTD, authored by Rod Johnson & Juergen Hoeller

      Author: Griffin Caprio

      This defines a simple and consistent way of creating a namespace
      of managed objects configured by a Spring XmlObjectFactory.
      This document type is used by most Spring functionality, including
      web application contexts, which are based on object factories.

      Each object element in this document defines an object.
      Typically the object type (System.Type) is specified, along with plain vanilla
      object properties.

      Object instances can be "singletons" (shared instances) or "prototypes"
      (independent instances).

      References among objects are supported, i.e. setting an object property
      to refer to another object in the same factory or an ancestor factory.

      As alternative to object references, "inner object definitions" can be used.
      Singleton flags and names of such "inner object" are always ignored:
      Inner object are anonymous prototypes.

      There is also support for lists, dictionaries, and sets.
    </xs:documentation>
  </xs:annotation>
  <xs:annotation>
    <xs:documentation>Defines a base type for any required string. Defines a string with a minimum length of 0</xs:documentation>
  </xs:annotation>
  <xs:simpleType name="nonNullString">
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:annotation>
    <xs:documentation>
      Element containing informative text describing the purpose of the enclosing
      element. Always optional.
      Used primarily for user documentation of XML object definition documents.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType name="description">
    <xs:restriction base="nonNullString"/>
  </xs:simpleType>
  <xs:complexType name="valueObject">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="type" type="nonNullString" use="optional"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="expression">
    <xs:sequence>
      <xs:element name="property" type="property" minOccurs="0" maxOccurs="2"/>
    </xs:sequence>
    <xs:attribute name="value" type="nonNullString" use="required"/>
  </xs:complexType>
  <!--
    Defines a reference to another object in this factory or an external
    factory (parent or included factory).
  -->
  <xs:complexType name="objectReference">
```

```

<xs:attribute name="object" type="nonNullString" use="optional"/>
<xs:attribute name="local" type="xs:IDREF" use="optional"/>
<xs:attribute name="parent" type="nonNullString" use="optional"/>
<!--
    References must specify a name of the target object.
    The "object" attribute can reference any name from any object in the context,
    to be checked at runtime.
    Local references, using the "local" attribute, have to use object ids;
    they can be checked by this DTD, thus should be preferred for references
    within the same object factory XML file.
-->
</xs:complexType>
<!-- Defines a reference to another object or a type. -->
<xs:complexType name="objectOrClassReference">
    <xs:attribute name="object" type="nonNullString" use="optional"/>
    <xs:attribute name="local" type="xs:IDREF" use="optional"/>
    <xs:attribute name="type" type="nonNullString" use="optional"/>
</xs:complexType>
<xs:group name="objectList">
    <xs:sequence>
        <xs:element name="description" type="description" minOccurs="0"/>
        <xs:choice>
            <xs:element name="object" type="vanillaObject"/>
            <!--
                Defines a reference to another object in this factory or an external
                factory (parent or included factory).
            -->
            <xs:element name="ref" type="objectReference"/>
            <!--
                Defines a string property value, which must also be the id of another
                object in this factory or an external factory (parent or included factory).
                While a regular 'value' element could instead be used for the same effect,
                using idref in this case allows validation of local object ids by the xml
                parser, and name completion by helper tools.
            -->
            <xs:element name="idref" type="objectReference"/>
            <!--
                A objectList can contain multiple inner object, ref, collection, or value elements.
                Lists are untyped, pending generics support, although references will be
                strongly typed.
                A objectList can also map to an array type. The necessary conversion
                is automatically performed by AbstractObjectFactory.
            -->
            <xs:element name="list">
                <xs:complexType>
                    <xs:group ref="objectList" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:attribute name="element-type" type="nonNullString" use="optional"/>
                </xs:complexType>
            </xs:element>
            <!--
                A set can contain multiple inner object, ref, collection, or value elements.
                Sets are untyped, pending generics support, although references will be
                strongly typed.
            -->
            <xs:element name="set">
                <xs:complexType>
                    <xs:group ref="objectList" minOccurs="0" maxOccurs="unbounded"/>
                </xs:complexType>
            </xs:element>
            <!--
                A Spring map is a mapping from a string key to object (a .NET IDictionary).
                Maps may be empty.
            -->
            <xs:element name="dictionary" type="objectMap"/>
            <!--
                Name-values elements differ from map elements in that values must be strings.
                Name-values may be empty.
            -->
            <xs:element name="name-values" type="objectNameValues"/>
            <!--
                Contains a string representation of a property value.
                The property may be a string, or may be converted to the
                required type using the System.ComponentModel.TypeConverter
                machinery. This makes it possible for application developers
                to write custom TypeConverter implementations that can

```

```

        convert strings to objects.

        Note that this is recommended for simple objects only.
        Configure more complex objects by setting properties to references
        to other objects.

-->
<xs:element name="value" type="valueObject"/>
<!--
    Contains a string representation of an expression.
-->
<xs:element name="expression" type="expression"/>
<!--
    Denotes a .NET null value. Necessary because an empty "value" tag
    will resolve to an empty String, which will not be resolved to a
    null value unless a special TypeConverter does so.
-->
<xs:element name="null"/>
</xs:choice>
</xs:sequence>
</xs:group>
<xs:complexType name="objectNameValues">
    <xs:sequence>
        <!--
            The "value" attribute is the string value of the property. The "key"
            attribute is the name of the property.
        -->
        <xs:element name="add" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType mixed="true">
                <xs:attribute name="key" type="nonNullString" use="required"/>
                <xs:attribute name="value" use="required" type="xs:string"/>
                <xs:attribute name="delimiters" use="optional" type="xs:string"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="importElement">
    <xs:attribute name="resource" type="nonNullString" use="required"/>
</xs:complexType>
<xs:complexType name="aliasElement">
    <xs:attribute name="name" type="nonNullString" use="required"/>
    <xs:attribute name="alias" type="nonNullString" use="required"/>
</xs:complexType>
<xs:complexType name="objectMap">
    <xs:sequence>
        <xs:element type="mapEntryElement" name="entry" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="key-type" type="nonNullString" use="optional"/>
    <xs:attribute name="value-type" type="nonNullString" use="optional"/>
</xs:complexType>
<xs:complexType name="mapEntryElement">
    <xs:sequence>
        <xs:element type="mapKeyElement" name="key" minOccurs="0" maxOccurs="1"/>
        <xs:group ref="objectList" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="key" type="nonNullString" use="optional"/>
    <xs:attribute name="value" type="nonNullString" use="optional"/>
    <xs:attribute name="expression" type="nonNullString" use="optional"/>
    <xs:attribute name="key-ref" type="nonNullString" use="optional"/>
    <xs:attribute name="value-ref" type="nonNullString" use="optional"/>
</xs:complexType>
<xs:complexType name="mapKeyElement">
    <xs:group ref="objectList" minOccurs="1"/>
</xs:complexType>
<xs:annotation>
    <xs:documentation>Defines constructor argument.</xs:documentation>
</xs:annotation>
<xs:complexType name="lookupMethod">
    <xs:attribute name="name" type="nonNullString" use="required"/>
    <xs:attribute name="object" type="nonNullString" use="required"/>
</xs:complexType>
<xs:complexType name="constructorArgument">
    <xs:group ref="objectList" minOccurs="0"/>
    <!--
        The constructor-arg tag can have an optional named parameter attribute,
        to specify a named parameter in the constructor argument list.
    -->

```

```

-->
<xs:attribute name="name" type="nonNullString" use="optional"/>
<!--
    The constructor-arg tag can have an optional index attribute,
    to specify the exact index in the constructor argument list. Only needed
    to avoid ambiguities, e.g. in case of 2 arguments of the same type.
-->
<xs:attribute name="index" type="nonNullString" use="optional"/>
<!--
    The constructor-arg tag can have an optional type attribute,
    to specify the exact type of the constructor argument. Only needed
    to avoid ambiguities, e.g. in case of 2 single argument constructors
    that can both be converted from a String.
-->
<xs:attribute name="type" type="nonNullString" use="optional"/>
<xs:attribute name="value" type="nonNullString" use="optional"/>
<xs:attribute name="expression" type="nonNullString" use="optional"/>
<xs:attribute name="ref" type="nonNullString" use="optional"/>
</xs:complexType>
<xs:annotation>
    <xs:documentation>Defines property.</xs:documentation>
</xs:annotation>
<xs:complexType name="property">
    <xs:group ref="objectList" minOccurs="0"/>
    <!-- The property name attribute is the name of the objects property. -->
    <xs:attribute name="name" type="nonNullString" use="required"/>
    <xs:attribute name="value" type="nonNullString" use="optional"/>
    <xs:attribute name="expression" type="nonNullString" use="optional"/>
    <xs:attribute name="ref" type="nonNullString" use="optional"/>
</xs:complexType>
<xs:annotation>
    <xs:documentation>Defines a single named object.</xs:documentation>
</xs:annotation>
<xs:complexType name="vanillaObject">
    <xs:sequence>
        <xs:element name="description" type="description" minOccurs="0" maxOccurs="1"/>
        <!--
            Object definitions can specify zero or more constructor arguments.
            They correspond to either a specific index of the constructor argument list
            or are supposed to be matched generically by type.
            This is an alternative to "autowire constructor".
        -->
        <xs:element name="constructor-arg" type="constructorArgument" minOccurs="0" maxOccurs="unbounded"/>
        <!--
            Object definitions can have zero or more properties.
            Spring supports primitives, references to other objects in the same or
            related factories, lists, dictionaries and properties.
        -->
        <xs:element name="property" type="property" minOccurs="0" maxOccurs="unbounded"/>
        <!--
            Object definitions can specify zero or more lookup-methods.
        -->
        <xs:element name="lookup-method" type="lookupMethod" minOccurs="0" maxOccurs="unbounded"/>
        <!-- Object definitions can have zero or more replaced-methods. -->
        <xs:element name="replaced-method" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="arg-type" minOccurs="0" maxOccurs="unbounded">
                        <xs:complexType>
                            <xs:attribute name="match" type="nonNullString" use="required"/>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="name" type="nonNullString" use="required"/>
                <xs:attribute name="replacer" type="nonNullString" use="required"/>
            </xs:complexType>
        </xs:element>
        <!-- Object definitions can have zero or more subscriptions. -->
        <xs:element name="listener" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="ref" type="objectOrClassReference" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
                <!-- The event(s) the object is interested in. -->
                <xs:attribute name="event" type="nonNullString" use="optional"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>

```

```

        <!-- The name or name pattern of the method that will handle the event(s). -->
        <xs:attribute name="method" type="nonNullString" use="required"/>
    </xs:complexType>
</xs:element>
</xs:sequence>
<!--
    Objects can be identified by an id, to enable reference checking.
    There are constraints on a valid XML id: if you want to reference your object
    in .NET code using a name that's illegal as an XML id, use the optional
    "name" attribute. If neither given, the object type name is used as id.
-->
<xs:attribute name="id" type="xs:ID" use="optional"/>
<!--
    Optional. Can be used to create one or more aliases illegal in an id.
    Multiple aliases can be separated by any number of spaces or commas.
-->
<xs:attribute name="name" type="nonNullString" use="optional"/>
<!--
    Each object definition must specify the full, assembly qualified of the type,
    or the name of the parent object from which the type can be worked out.

    Note that a child object definition that references a parent will just
    add respectively override property values and be able to change the
    singleton status. It will inherit all of the parent's other parameters
    like lazy initialization or autowire settings.
-->
<xs:attribute name="type" type="nonNullString" use="optional"/>
<xs:attribute name="parent" type="nonNullString" use="optional"/>
<!--
    Is this object "abstract", i.e. not meant to be instantiated itself but
    rather just serving as parent for concrete child object definitions?
    Default is false. Specify true to tell the object factory to not try to
    instantiate that particular object in any case.
-->
<xs:attribute name="abstract" type="xs:boolean" use="optional" default="false"/>
<!--
    Is this object a "singleton" (one shared instance, which will
    be returned by all calls to GetObject() with the id),
    or a "prototype" (independent instance resulting from each call to
    getObject()). Default is singleton.

    Singletons are most commonly used, and are ideal for multi-threaded
    service objects.
-->
<xs:attribute name="singleton" type="xs:boolean" use="optional" default="true"/>
<!--
    Optional attribute controlling the scope of singleton instances. It is
    only applicable to ASP.Net web applications and it has no effect on prototype
    objects. Applications other than ASP.Net web applications simply ignore this attribute.
    It has 3 possible values:
    1. "application"
    Default object scope. Objects defined with application scope will behave like
    traditional singleton objects. Same instance will be returned from every call
    to IApplicationContext.GetObject()

    2. "session"
    Objects with this scope will be stored within user's HTTP session. Session scope
    is typically used for objects such as shopping cart, user profile, etc.

    3. "request"
    Object with this scope will be initialized for each HTTP request, but unlike with prototype
    objects, same instance will be returned from all calls to IApplicationContext.GetObject()
    within the same HTTP request. For example, if one ASP page forwards request to another using
    Server.Transfer method, they can easily share the state by configuring dependency to the same
    request-scoped object.
-->
<xs:attribute name="scope" use="optional" default="application">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="application"/>
            <xs:enumeration value="session"/>
            <xs:enumeration value="request"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>

```



```

<!--
    Is this object to be lazily initialized?
    If false, it will get instantiated on startup by object factories
    that perform eager initialization of singletons.
-->
<xs:attribute name="lazy-init" use="optional" default="default">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="true"/>
            <xs:enumeration value="false"/>
            <xs:enumeration value="default"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<!--
    Optional attribute controlling whether to "autowire" object properties.
    This is an automagical process in which object references don't need to be coded
    explicitly in the XML object definition file, but Spring works out dependencies.

    There are 5 modes:

    1. "no"
    The traditional Spring default. No automagical wiring. Object references
    must be defined in the XML file via the <ref> element. We recommend this
    in most cases as it makes documentation more explicit.

    2. "byName"
    Autowiring by property name. If a object of class Cat exposes a dog property,
    Spring will try to set this to the value of the object "dog" in the current factory.

    3. "byType"
    Autowiring if there is exactly one object of the property type in the object factory.
    If there is more than one, a fatal error is raised, and you can't use byType
    autowiring for that object. If there is none, nothing special happens - use
    dependency-check="objects" to raise an error in that case.

    4. "constructor"
    Analogous to "byType" for constructor arguments. If there isn't exactly one object
    of the constructor argument type in the object factory, a fatal error is raised.

    5. "autodetect"
    Chooses "constructor" or "byType" through introspection of the object class.
    If a default constructor is found, "byType" gets applied.

    The latter two are similar to PicoContainer and make object factories simple to
    configure for small namespaces, but doesn't work as well as standard Spring
    behaviour for bigger applications.

    Note that explicit dependencies, i.e. "property" and "constructor-arg" elements,
    always override autowiring. Autowire behaviour can be combined with dependency
    checking, which will be performed after all autowiring has been completed.
-->
<xs:attribute name="autowire" use="optional" default="default">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="no"/>
            <xs:enumeration value="byName"/>
            <xs:enumeration value="byType"/>
            <xs:enumeration value="constructor"/>
            <xs:enumeration value="autodetect"/>
            <xs:enumeration value="default"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<!--
    Optional attribute controlling whether to check whether all this
    objects dependencies, expressed in its properties, are satisfied.
    Default is no dependency checking.

    "simple" type dependency checking includes primitives and String
    "object" includes collaborators (other objects in the factory)
    "all" includes both types of dependency checking
-->
<xs:attribute name="dependency-check" use="optional" default="default">
    <xs:simpleType>

```

```

        <xs:restriction base="xs:string">
            <xs:enumeration value="none"/>
            <xs:enumeration value="objects"/>
            <xs:enumeration value="simple"/>
            <xs:enumeration value="all"/>
            <xs:enumeration value="default"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<!--
    The names of the objects that this object depends on being initialized.
    The object factory will guarantee that these objects get initialized before.

    Note that dependencies are normally expressed through object properties or
    constructor arguments. This property should just be necessary for other kinds
    of dependencies like statics (*ugh*) or database preparation on startup.
-->
<xs:attribute name="depends-on" type="nonNullString" use="optional"/>
<!--
    Optional attribute for the name of the custom initialization method
    to invoke after setting object properties. The method must have no arguments,
    but may throw any exception.
-->
<xs:attribute name="init-method" type="nonNullString" use="optional"/>
<!--
    Optional attribute for the name of the custom destroy method to invoke
    on object factory shutdown. The method must have no arguments,
    but may throw any exception. Note: Only invoked on singleton objects!
-->
<xs:attribute name="destroy-method" type="nonNullString" use="optional"/>
<xs:attribute name="factory-method" type="nonNullString" use="optional"/>
<xs:attribute name="factory-object" type="nonNullString" use="optional"/>
</xs:complexType>

<xs:annotation>
    <xs:documentation>The document root. At least one object definition is required.</xs:documentation>
</xs:annotation>
<xs:element name="objects">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="description" type="description" minOccurs="0" maxOccurs="1"/>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element name="import" type="importElement"/>
                <xs:element name="alias" type="aliasElement"/>
                <xs:element name="object" type="vanillaObject"/>
                <xs:any namespace="##other" processContents="strict"/>
            </xs:choice>
        </xs:sequence>
    <!--
        Default values for all object definitions. Can be overridden at
        the "object" level. See those attribute definitions for details.
    -->
    <xs:attribute name="default-lazy-init" type="xs:boolean" use="optional" default="false"/>
    <xs:attribute name="default-dependency-check" use="optional" default="none">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="none"/>
                <xs:enumeration value="objects"/>
                <xs:enumeration value="simple"/>
                <xs:enumeration value="all"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="default-autowire" use="optional" default="no">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="no"/>
                <xs:enumeration value="byName"/>
                <xs:enumeration value="byType"/>
                <xs:enumeration value="constructor"/>
                <xs:enumeration value="autodetect"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>

```

```
</xs:element>  
  
</xs:schema>
```