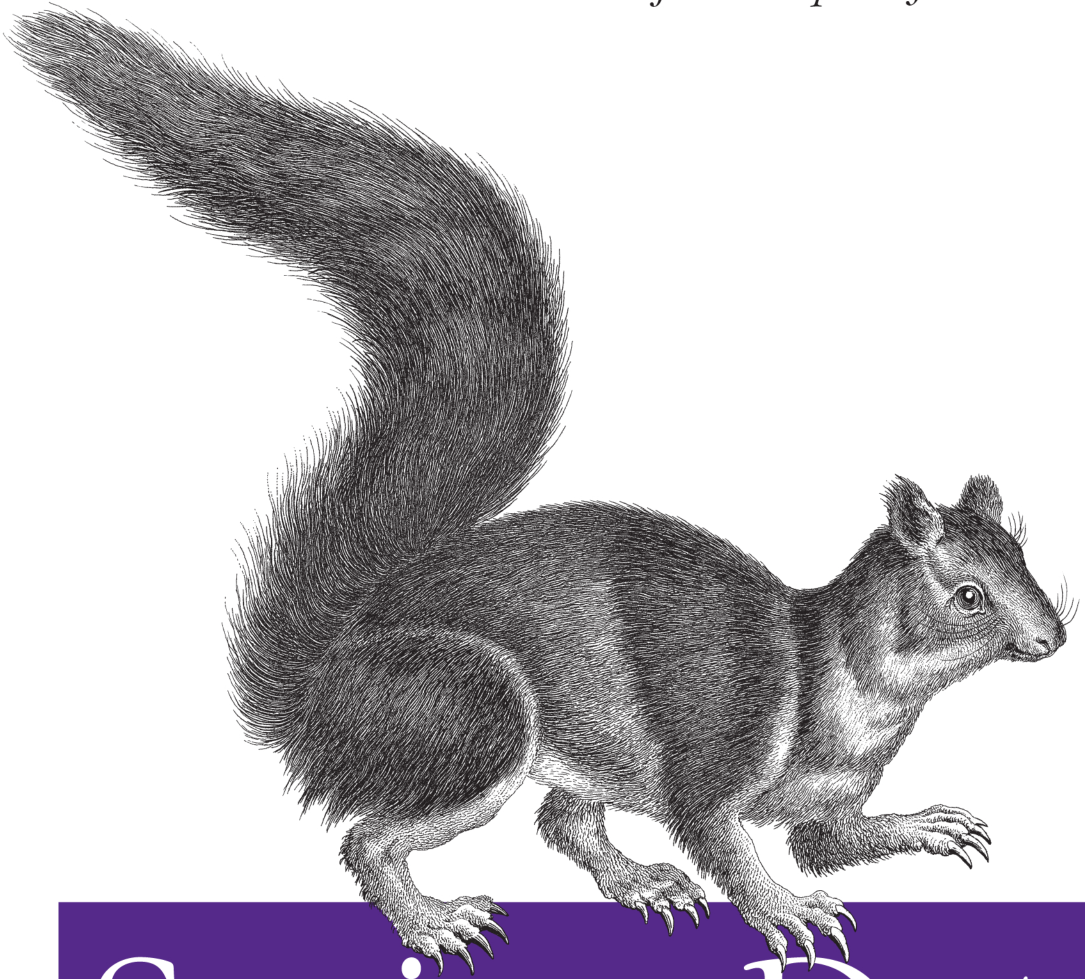
Modern Data Access for Enterprise Java



Spring Data

Mark Pollack, Oliver Gierke,
Thomas Risberg, Jonathan L. Brisbin,
& Michael Hunger

O'REILLY®

Spring Data

Modern Data Access for Enterprise Java

Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Spring Data
by Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Kristen Borg
Proofreader: Rachel Monaghan
Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2012: First Edition.

Revision History for the First Edition:
2012-10-11 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449323950* for release details.

ISBN: 978-1-449-32395-0

[LSI]

1349968177

Thanks to my wife, Daniela, and sons, Gabriel and Alexandre, whose patience with me stealing time away for "the book" made it possible.

—Mark Pollack

I'd like to thank my family, friends, fellow musicians, and everyone I've had the pleasure to work with so far; the entire Spring Data and SpringSource team for this awesome journey; and last, but actually first of all, Sabine, for her inexhaustible love and support.

—Oliver Gierke

To my wife, Carol, and my son, Alex, thank you for enriching my life and for all your support and encouragement.

—Thomas Risberg

*To my wife, Tisha; my sons, Jack, Ben, and Dan-
iel; and my daughters, Morgan and Hannah.
Thank you for your love, support, and patience.
All this wouldn't be worth it without you.*

—Jon Brisbin

*My special thanks go to Rod and Emil for starting
the Spring Data project and to Oliver for making
it great. My family is always very supportive of
my crazy work; I'm very grateful to have such
understanding women around me.*

—Michael Hunger

*I'd like to thank my wife, Nanette, and my kids for
their support, patience, and understanding.
Thanks also to Rod and my colleagues on the
Spring Data team for making all of this possible.*

—David Turanski

Table of Contents

Foreword xiii

Preface xv

Part I. Background

1. The Spring Data Project 3
- NoSQL Data Access for Spring Developers 3
- General Themes 5
- The Domain 6
- The Sample Code 6
  - Importing the Source Code into Your IDE 7

2. Repositories: Convenient Data Access Layers 13
- Quick Start 13
- Defining Query Methods 16
  - Query Lookup Strategies 16
  - Query Derivation 17
  - Pagination and Sorting 18
- Defining Repositories 19
  - Fine-Tuning Repository Interfaces 20
  - Manually Implementing Repository Methods 21
- IDE Integration 22
  - IntelliJ IDEA 24

3. Type-Safe Querying Using Querydsl 27
- Introduction to Querydsl 27
- Generating the Query Metamodel 30
  - Build System Integration 30
  - Supported Annotation Processors 31

Querying Stores Using Querydsl 32
Integration with Spring Data Repositories 32
Executing Predicates 33
Manually Implementing Repositories 34

Part II. Relational Databases

4. JPA Repositories 37
The Sample Project 37
The Traditional Approach 42
Bootstrapping the Sample Code 44
Using Spring Data Repositories 47
Transactionality 50
Repository Querydsl Integration 51

5. Type-Safe JDBC Programming with Querydsl SQL 53
The Sample Project and Setup 53
The HyperSQL Database 54
The SQL Module of Querydsl 54
Build System Integration 58
The Database Schema 59
The Domain Implementation of the Sample Project 60
The QueryDslJdbcTemplate 63
Executing Queries 64
The Beginning of the Repository Implementation 64
Querying for a Single Object 65
The OneToManyResultSetExtractor Abstract Class 67
The CustomerListExtractor Implementation 68
The Implementations for the RowMappers 69
Querying for a List of Objects 71
Insert, Update, and Delete Operations 71
Inserting with the SQLInsertClause 71
Updating with the SQLUpdateClause 72
Deleting Rows with the SQLDeleteClause 73

Part III. NoSQL

6. MongoDB: A Document Store 77
MongoDB in a Nutshell 77
Setting Up MongoDB 78
Using the MongoDB Shell 79

The MongoDB Java Driver 80
Setting Up the Infrastructure Using the Spring Namespace 81
The Mapping Subsystem 83
The Domain Model 83
Setting Up the Mapping Infrastructure 89
Indexing 91
Customizing Conversion 91
MongoTemplate 94
Mongo Repositories 96
Infrastructure Setup 96
Repositories in Detail 97
Mongo Querydsl Integration 99

7. Neo4j: A Graph Database 101
Graph Databases 101
Neo4j 102
Spring Data Neo4j Overview 105
Modeling the Domain as a Graph 106
Persisting Domain Objects with Spring Data Neo4j 111
Neo4jTemplate 112
Combining Graph and Repository Power 113
Basic Graph Repository Operations 115
Derived and Annotated Finder Methods 116
Advanced Graph Use Cases in the Example Domain 119
Multiple Roles for a Single Node 119
Product Categories and Tags as Examples for In-Graph Indexes 120
Leverage Similar Interests (Collaborative Filtering) 121
Recommendations 121
Transactions, Entity Life Cycle, and Fetch Strategies 122
Advanced Mapping Mode 123
Working with Neo4j Server 124
Continuing From Here 125

8. Redis: A Key/Value Store 127
Redis in a Nutshell 127
Setting Up Redis 127
Using the Redis Shell 128
Connecting to Redis 129
Object Conversion 130
Object Mapping 132
Atomic Counters 134
Pub/Sub Functionality 135
Listening and Responding to Messages 135

Using Spring's Cache Abstraction with Redis 136

Part IV. Rapid Application Development

9. Persistence Layers with Spring Roo 141
A Brief Introduction to Roo 141
Roo's Persistence Layers 143
Quick Start 143
Using Roo from the Command Line 143
Using Roo with Spring Tool Suite 145
A Spring Roo JPA Repository Example 147
Creating the Project 147
Setting Up JPA Persistence 148
Creating the Entities 148
Defining the Repositories 150
Creating the Web Layer 150
Running the Example 151
A Spring Roo MongoDB Repository Example 152
Creating the Project 153
Setting Up MongoDB Persistence 153
Creating the Entities 153
Defining the Repositories 154
Creating the Web Layer 154
Running the Example 154

10. REST Repository Exporter 157
The Sample Project 158
Interacting with the REST Exporter 160
Accessing Products 162
Accessing Customers 165
Accessing Orders 169

Part V. Big Data

11. Spring for Apache Hadoop 175
Challenges Developing with Hadoop 176
Hello World 177
Hello World Revealed 179
Hello World Using Spring for Apache Hadoop 183
Scripting HDFS on the JVM 187
Combining HDFS Scripting and Job Submission 190

Job Scheduling 191
Scheduling MapReduce Jobs with a TaskScheduler 191
Scheduling MapReduce Jobs with Quartz 192

12. Analyzing Data with Hadoop 195
Using Hive 195
Hello World 196
Running a Hive Server 197
Using the Hive Thrift Client 198
Using the Hive JDBC Client 201
Apache Logfile Analysis Using Hive 202
Using Pig 204
Hello World 205
Running a PigServer 207
Controlling Runtime Script Execution 209
Calling Pig Scripts Inside Spring Integration Data Pipelines 211
Apache Logfile Analysis Using Pig 212
Using HBase 214
Hello World 214
Using the HBase Java Client 215

13. Creating Big Data Pipelines with Spring Batch and Spring Integration 219
Collecting and Loading Data into HDFS 219
An Introduction to Spring Integration 220
Copying Logfiles 222
Event Streams 226
Event Forwarding 229
Management 230
An Introduction to Spring Batch 232
Processing and Loading Data from a Database 234
Hadoop Workflows 238
Spring Batch Support for Hadoop 238
Wordcount as a Spring Batch Application 240
Hive and Pig Steps 242
Exporting Data from HDFS 243
From HDFS to JDBC 243
From HDFS to MongoDB 249
Collecting and Loading Data into Splunk 250

Part VI. Data Grids

14. **GemFire: A Distributed Data Grid** 255
    - GemFire in a Nutshell 255
    - Caches and Regions 257
    - How to Get GemFire 257
    - Configuring GemFire with the Spring XML Namespace 258
        - Cache Configuration 258
        - Region Configuration 263
        - Cache Client Configuration 265
        - Cache Server Configuration 267
        - WAN Configuration 267
        - Disk Store Configuration 268
    - Data Access with GemfireTemplate 269
    - Repository Usage 271
        - POJO Mapping 271
        - Creating a Repository 272
        - PDX Serialization 272
    - Continuous Query Support 273

Bibliography 275

Index 277

Foreword

We live in interesting times. New business processes are driving new requirements. Familiar assumptions are under threat—among them, that the relational database should be the default choice for persistence. While this is now widely accepted, it is far from clear how to proceed effectively into the new world.

A proliferation of data store choices creates fragmentation. Many newer stores require more developer effort than Java developers are used to regarding data access, pushing into the application things customarily done in a relational database.

This book helps you make sense of this new reality. It provides an excellent overview of today's storage world in the context of today's hardware, and explains why NoSQL stores are important in solving modern business problems.

Because of the language's identification with the often-conservative enterprise market (and perhaps also because of the sophistication of Java object-relational mapping [ORM] solutions), Java developers have traditionally been poorly served in the NoSQL space. Fortunately, this is changing, making this an important and timely book. Spring Data is an important project, with the potential to help developers overcome new challenges.

Many of the values that have made Spring the preferred platform for enterprise Java developers deliver particular benefit in a world of fragmented persistence solutions. Part of the value of Spring is how it brings consistency (without descending to a lowest common denominator) in its approach to different technologies with which it integrates. A distinct "Spring way" helps shorten the learning curve for developers and simplifies code maintenance. If you are already familiar with Spring, you will find that Spring Data eases your exploration and adoption of unfamiliar stores. If you aren't already familiar with Spring, this is a good opportunity to see how Spring can simplify your code and make it more consistent.

The authors are uniquely qualified to explain Spring Data, being the project leaders. They bring a mix of deep Spring knowledge and involvement and intimate experience with a range of modern data stores. They do a good job of explaining the motivation of Spring Data and how it continues the mission Spring has long pursued regarding data access. There is valuable coverage of how Spring Data works with other parts of

Spring, such as Spring Integration and Spring Batch. The book also provides much value that goes beyond Spring—for example, the discussions of the repository concept, the merits of type-safe querying, and why the Java Persistence API (JPA) is not appropriate as a general data access solution.

While this is a book about data access rather than working with NoSQL, many of you will find the NoSQL material most valuable, as it introduces topics and code with which you are likely to be less familiar. All content is up to the minute, and important topics include document databases, graph databases, key/value stores, Hadoop, and the Gemfire data fabric.

We programmers are practical creatures and learn best when we can be hands-on. The book has a welcome practical bent. Early on, the authors show how to get the sample code working in the two leading Java integrated development environments (IDEs), including handy screenshots. They explain requirements around database drivers and basic database setup. I applaud their choice of hosting the sample code on GitHub, making it universally accessible and browsable. Given the many topics the book covers, the well-designed examples help greatly to tie things together.

The emphasis on practical development is also evident in the chapter on Spring Roo, the rapid application development (RAD) solution from the Spring team. Most Roo users are familiar with how Roo can be used with a traditional JPA architecture; the authors show how Roo's productivity can be extended beyond relational databases.

When you've finished this book, you will have a deeper understanding of why modern data access is becoming more specialized and fragmented, the major categories of NoSQL data stores, how Spring Data can help Java developers operate effectively in this new environment, and where to look for deeper information on individual topics in which you are particularly interested. Most important, you'll have a great start to your own exploration in code!

—Rod Johnson
Creator, Spring Framework

Preface

Overview of the New Data Access Landscape

The data access landscape over the past seven or so years has changed dramatically. Relational databases, the heart of storing and processing data in the enterprise for over 30 years, are no longer the only game in town. The past seven years have seen the birth —and in some cases the death—of many alternative data stores that are being used in mission-critical enterprise applications. These new data stores have been designed specifically to solve data access problems that relational database can't handle as effectively.

An example of a problem that pushes traditional relational databases to the breaking point is scale. How do you store hundreds or thousands of terabytes (TB) in a relational database? The answer reminds us of the old joke where the patient says, "Doctor, it hurts when I do this," and the doctor says, "Then don't do that!" Jokes aside, what is driving the need to store this much data? In 2001, IDC reported that "the amount of information created and replicated will surpass 1.8 zettabytes and more than double every two years."[1] New data types range from media files to logfiles to sensor data (RFID, GPS, telemetry...) to tweets on Twitter and posts on Facebook. While data that is stored in relational databases is still crucial to the enterprise, these new types of data are not being stored in relational databases.

While general consumer demands drive the need to store large amounts of media files, enterprises are finding it important to store and analyze many of these new sources of data. In the United States, companies in all sectors have at least 100 TBs of stored data and many have more than 1 petabyte (PB).[2] The general consensus is that there are significant bottom-line benefits for businesses to continually analyze this data. For example, companies can better understand the behavior of their products if the products themselves are sending "phone home" messages about their health. To better understand their customers, companies can incorporate social media data into their decision-making processes. This has led to some interesting mainstream media

1. IDC; *Extracting Value from Chaos*. 2011.
2. IDC; US Bureau of Labor Statistics

reports—for example, on why Orbitz shows more expensive hotel options to Mac users and how Target can predict when one of its customers will soon give birth, allowing the company to mail coupon books to the customer's home before public birth records are available.

Big data generally refers to the process in which large quantities of data are stored, kept in raw form, and continually analyzed and combined with other data sources to provide a deeper understanding of a particular domain, be it commercial or scientific in nature.

Many companies and scientific laboratories had been performing this process before the term *big data* came into fashion. What makes the current process different from before is that the value derived from the intelligence of data analytics is higher than the hardware costs. It is no longer necessary to buy a 40K per CPU box to perform this type of data analysis; clusters of commodity hardware now cost $1k per CPU. For large datasets, the cost of storage area network (SAN) or network area storage (NAS) becomes prohibitive: $1 to $10 per gigabyte, while local disk costs only $0.05 per gigabyte with replication built into the database instead of the hardware. Aggregate data transfer rates for clusters of commodity hardware that use local disk are also significantly higher than SAN- or NAS-based systems—500 times faster for similarly priced systems. On the software side, the majority of the new data access technologies are open source. While open source does not mean zero cost, it certainly lowers the barrier for entry and overall cost of ownership versus the traditional commercial software offerings in this space.

Another problem area that new data stores have identified with relational databases is the relational data model. If you are interested in analyzing the social graph of millions of people, doesn't it sound quite natural to consider using a graph database so that the implementation more closely models the domain? What if requirements are continually driving you to change your relational database management system (RDBMS) schema and object-relational mapping (ORM) layer? Perhaps a "schema-less" document database will reduce the object mapping complexity and provide a more easily evolvable system as compared to the more rigid relational model. While each of the new databases is unique in its own way, you can provide a rough taxonomy across most of them based on their data models. The basic camps they fall into are:

Key/value
: A familiar data model, much like a hashtable.

Column family
: An extended key/value data model in which the value data type can also be a sequence of key/value pairs.

Document
: Collections that contain semistructured data, such as XML or JSON.

Graph
: Based on graph theory. The data model has nodes and edges, each of which may have properties.

The general name under which these new databases have become grouped is “NoSQL databases.” In retrospect, this name, while catchy, isn’t very accurate because it seems to imply that you can’t query the database, which isn’t true. It reflects the basic shift away from the relational data model as well as a general shift away from ACID (atomicity, consistency, isolation, durability) characteristics of relational databases.

One of the driving factors for the shift away from ACID characteristics is the emergence of applications that place a higher priority on scaling writes and having a partially functioning system even when parts of the system have failed. While scaling reads in a relational database can be achieved through the use of in-memory caches that front the database, scaling writes is much harder. To put a label on it, these new applications favor a system that has so-called “BASE” semantics, where the acronym represents *basically available, scalable, eventually consistent*. Distributed data grids with a key/value data model generally have not been grouped into this new wave of NoSQL databases. However, they offer similar features to NoSQL databases in terms of the scale of data they can handle as well as distributed computation features that colocate computing power and data.

As you can see from this brief introduction to the new data access landscape, there is a revolution taking place, which for data geeks is quite exciting. Relational databases are not dead; they are still central to the operation of many enterprises and will remain so for quite some time. The trends, though, are very clear: new data access technologies are solving problems that traditional relational databases can’t, so we need to broaden our skill set as developers and have a foot in both camps.

The Spring Framework has a long history of simplifying the development of Java applications, in particular for writing RDBMS-based data access layers that use Java database connectivity (JDBC) or object-relational mappers. In this book we aim to help developers get a handle on how to effectively develop Java applications across a wide range of these new technologies. The Spring Data project directly addresses these new technologies so that you can extend your existing knowledge of Spring to them, or perhaps learn more about Spring as a byproduct of using Spring Data. However, it doesn’t leave the relational database behind. Spring Data also provides an extensive set of new features to Spring’s RDBMS support.

How to Read This Book

This book is intended to give you a hands-on introduction to the Spring Data project, whose core mission is to enable Java developers to use state-of-the-art data processing and manipulation tools but also use traditional databases in a state-of-the-art manner. We’ll start by introducing you to the project, outlining the primary motivation of SpringSource and the team. We’ll also describe the domain model of the sample projects that accommodate each of the later chapters, as well as how to access and set up the code (Chapter 1).

We'll then discuss the general concepts of Spring Data repositories, as they are a common theme across the various store-specific parts of the project (Chapter 2). The same applies to Querydsl, which is discussed in general in Chapter 3. These two chapters provide a solid foundation to explore the store specific integration of the repository abstraction and advanced query functionality.

To start Java developers in well-known terrain, we'll then spend some time on traditional persistence technologies like JPA (Chapter 4) and JDBC (Chapter 5). Those chapters outline what features the Spring Data modules add on top of the already existing JPA and JDBC support provided by Spring.

After we've finished that, we introduce some of the NoSQL stores supported by the Spring Data project: MongoDB as an example of a document database (Chapter 6), Neo4j as an example of a graph database (Chapter 7), and Redis as an example of a key/value store (Chapter 8). HBase, a column family database, is covered in a later chapter (Chapter 12). These chapters outline mapping domain classes onto the store-specific data structures, interacting easily with the store through the provided application programming interface (API), and using the repository abstraction.

We'll then introduce you to the Spring Data REST exporter (Chapter 10) as well as the Spring Roo integration (Chapter 9). Both projects build on the repository abstraction and allow you to easily export Spring Data–managed entities to the Web, either as a representational state transfer (REST) web service or as backing to a Spring Roo–built web application.

The book next takes a tour into the world of big data—Hadoop and Spring for Apache Hadoop in particular. It will introduce you to using cases implemented with Hadoop and show how the Spring Data module eases working with Hadoop significantly (Chapter 11). This leads into a more complex example of building a big data pipeline using Spring Batch and Spring Integration—projects that come nicely into play in big data processing scenarios (Chapter 12 and Chapter 13).

The final chapter discusses the Spring Data support for Gemfire, a distributed data grid solution (Chapter 14).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Spring Data* by Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger (O'Reilly). Copyright 2013 Mark Pollack, Oliver Gierke, Thomas Risberg, Jonathan L. Brisbin, and Michael Hunger, 978-1-449-32395-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

The code samples are posted on GitHub.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/spring-data-1e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We would like to thank Rod Johnson and Emil Eifrem for starting what was to become the Spring Data project.

A big thank you goes to David Turanski for pitching in and helping out with the GemFire chapter. Thank you to Richard McDougall for the big data statistics used in the introduction, and to Costin Leau for help with writing the Hadoop sample applications.

We would also like to thank O'Reilly Media, especially Meghan Blanchette for guiding us through the project, production editor Kristen Borg, and copyeditor Rachel Monaghan. Thanks to Greg Turnquist, Joris Kuipers, Johannes Hiemer, Joachim Arrasz, Stephan Hochdörfer, Mark Spritzler, Jim Webber, Lasse Westh-Nielsen, and all other technical reviewers for their feedback. Thank you to the community around the project for sending feedback and issues so that we could constantly improve. Last but not least, thanks to our friends and families for their patience, understanding, and support.

PART I
Background

CHAPTER 1

The Spring Data Project

The Spring Data project was coined at Spring One 2010 and originated from a hacking session of Rod Johnson (SpringSource) and Emil Eifrem (Neo Technologies) early that year. They were trying to integrate the Neo4j graph database with the Spring Framework and evaluated different approaches. The session created the foundation for what would eventually become the very first version of the Neo4j module of Spring Data, a new SpringSource project aimed at supporting the growing interest in NoSQL data stores, a trend that continues to this day.

Spring has provided sophisticated support for traditional data access technologies from day one. It significantly simplified the implementation of data access layers, regardless of whether JDBC, Hibernate, TopLink, JDO, or iBatis was used as persistence technology. This support mainly consisted of simplified infrastructure setup and resource management as well as exception translation into Spring's `DataAccessException`s. This support has matured over the years and the latest Spring versions contained decent upgrades to this layer of support.

The traditional data access support in Spring has targeted relational databases only, as they were the predominant tool of choice when it came to data persistence. As NoSQL stores enter the stage to provide reasonable alternatives in the toolbox, there's room to fill in terms of developer support. Beyond that, there are yet more opportunities for improvement even for the traditional relational stores. These two observations are the main drivers for the Spring Data project, which consists of dedicated modules for NoSQL stores as well as JPA and JDBC modules with additional support for relational databases.

NoSQL Data Access for Spring Developers

Although the term NoSQL is used to refer to a set of quite young data stores, all of the stores have very different characteristics and use cases. Ironically, it's the nonfeature (the lack of support for running queries using SQL) that actually named this group of databases. As these stores have quite different traits, their Java drivers have completely different APIs to leverage the stores' special traits and features. Trying to abstract away their differences would actually remove the benefits each NoSQL data store offers. A graph database should be chosen to store highly interconnected data. A document database should be used for tree and aggregate-like data structures. A key/value store should be chosen if you need cache-like functionality and access patterns.

With the JPA, the Java EE (Enterprise Edition) space offers a persistence API that could have been a candidate to front implementations of NoSQL databases. Unfortunately, the first two sentences of the specification already indicate that this is probably not working out:

> This document is the specification of the Java API for the management of persistence and object/relational mapping with Java EE and Java SE. The technical objective of this work is to provide an object/relational mapping facility for the Java application developer using a Java domain model to manage a relational database.

This theme is clearly reflected in the specification later on. It defines concepts and APIs that are deeply connected to the world of relational persistence. An `@Table` annotation would not make a lot of sense for NoSQL databases, nor would `@Column` or `@JoinColumn`. How should one implement the transaction API for stores like MongoDB, which essentially do not provide transactional semantics spread across multidocument manipulations? So implementing a JPA layer on top of a NoSQL store would result in a profile of the API at best.

On the other hand, all the special features NoSQL stores provide (geospatial functionality, map-reduce operations, graph traversals) would have to be implemented in a proprietary fashion anyway, as JPA simply does not provide abstractions for them. So we would essentially end up in a worst-of-both-worlds scenario—the parts that can be implemented behind JPA plus additional proprietary features to reenable store-specific features.

This context rules out JPA as a potential abstraction API for these stores. Still, we would like to see the programmer productivity and programming model consistency known from various Spring ecosystem projects to simplify working with NoSQL stores. This led the Spring Data team to declare the following mission statement:

> Spring Data provides a familiar and consistent Spring-based programming model for NoSQL and relational stores while retaining store-specific features and capabilities.

So we decided to take a slightly different approach. Instead of trying to abstract all stores behind a single API, the Spring Data project provides a consistent programming model across the different store implementations using patterns and abstractions already known from within the Spring Framework. This allows for a consistent experience when you're working with different stores.

General Themes

A core theme of the Spring Data project available for all of the stores is support for configuring resources to access the stores. This support is mainly implemented as XML namespace and support classes for Spring JavaConfig and allows us to easily set up access to a Mongo database, an embedded Neo4j instance, and the like. Also, integration with core Spring functionality like JMX is provided, meaning that some stores will expose statistics through their native API, which will be exposed to JMX via Spring Data.

Most of the NoSQL Java APIs do not provide support to map domain objects onto the stores' data abstractions (documents in MongoDB; nodes and relationships for Neo4j). So, when working with the native Java drivers, you would usually have to write a significant amount of code to map data onto the domain objects of your application when reading, and vice versa on writing. Thus, a very core part of the Spring Data modules is a mapping and conversion API that allows obtaining metadata about domain classes to be persistent and enables the actual conversion of arbitrary domain objects into store-specific data types.

On top of that, we'll find opinionated APIs in the form of template pattern implementations already well known from Spring's `JdbcTemplate`, `JmsTemplate`, etc. Thus, there is a `RedisTemplate`, a `MongoTemplate`, and so on. As you probably already know, these templates offer helper methods that allow us to execute commonly needed operations like persisting an object with a single statement while automatically taking care of appropriate resource management and exception translation. Beyond that, they expose callback APIs that allow you to access the store-native APIs while still getting exceptions translated and resources managed properly.

These features already provide us with a toolbox to implement a data access layer like we're used to with traditional databases. The upcoming chapters will guide you through this functionality. To ease that process even more, Spring Data provides a repository abstraction on top of the template implementation that will reduce the effort to implement data access objects to a plain interface definition for the most common scenarios like performing standard CRUD operations as well as executing queries in case the store supports that. This abstraction is actually the topmost layer and blends the APIs of the different stores as much as reasonably possible. Thus, the store-specific implementations of it share quite a lot of commonalities. This is why you'll find a dedicated chapter (Chapter 2) introducing you to the basic programming model.

Now let's take a look at our sample code and the domain model that we will use to demonstrate the features of the particular store modules.

The Domain

To illustrate how to work with the various Spring Data modules, we will be using a sample domain from the ecommerce sector (see Figure 1-1). As NoSQL data stores usually have a dedicated sweet spot of functionality and applicability, the individual chapters might tweak the actual implementation of the domain or even only partially implement it. This is not to suggest that you have to model the domain in a certain way, but rather to emphasize which store might actually work better for a given application scenario.

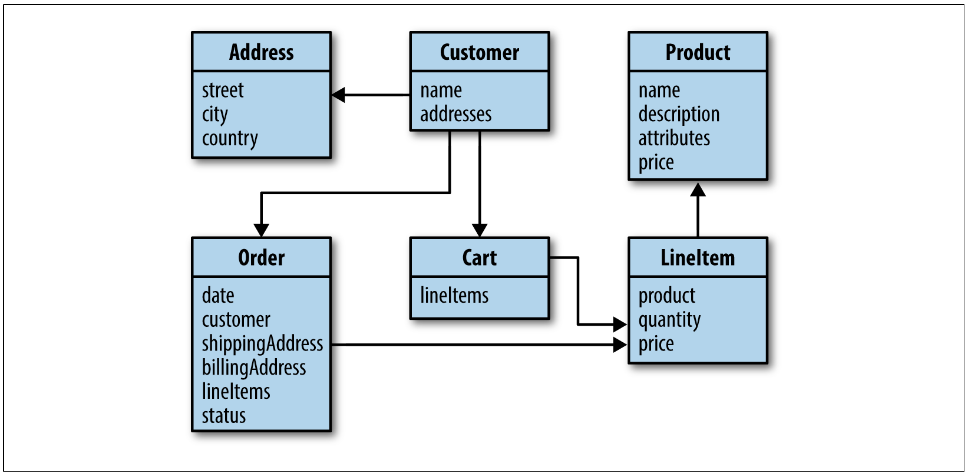


Figure 1-1. The domain model

At the core of our model, we have a customer who has basic data like a first name, a last name, an email address, and a set of addresses in turn containing street, city, and country. We also have products that consist of a name, a description, a price, and arbitrary attributes. These abstractions form the basis of a rudimentary CRM (customer relationship management) and inventory system. On top of that, we have orders a customer can place. An order contains the customer who placed it, shipping and billing addresses, the date the order was placed, an order status, and a set of line items. These line items in turn reference a particular product, the number of products to be ordered, and the price of the product.

The Sample Code

The sample code for this book can be found on GitHub. It is a Maven project containing a module per chapter. It requires either a Maven 3 installation on your machine or an IDE capable of importing Maven projects such as the Spring Tool Suite (STS). Getting the code is as simple as cloning the repository:

```
$ cd ~/dev
$ git clone https://github.com/SpringSource/spring-data-book.git
Cloning into 'spring-data-book'...
remote: Counting objects: 253, done.
remote: Compressing objects: 100% (137/137), done.
Receiving objects: 100% (253/253), 139.99 KiB | 199 KiB/s, done.
remote: Total 253 (delta 91), reused 219 (delta 57)
Resolving deltas: 100% (91/91), done.
$ cd spring-data-book
```

You can now build the code by executing Maven from the command line as follows:

```
$ mvn clean package
```

This will cause Maven to resolve dependencies, compile and test code, execute tests, and package the modules eventually.

Importing the Source Code into Your IDE

STS/Eclipse

STS ships with the m2eclipse plug-in to easily work with Maven projects right inside your IDE. So, if you have it already downloaded and installed (have a look at Chapter 3 for details), you can choose the Import option of the File menu. Select the Existing Maven Projects option from the dialog box, shown in Figure 1-2.

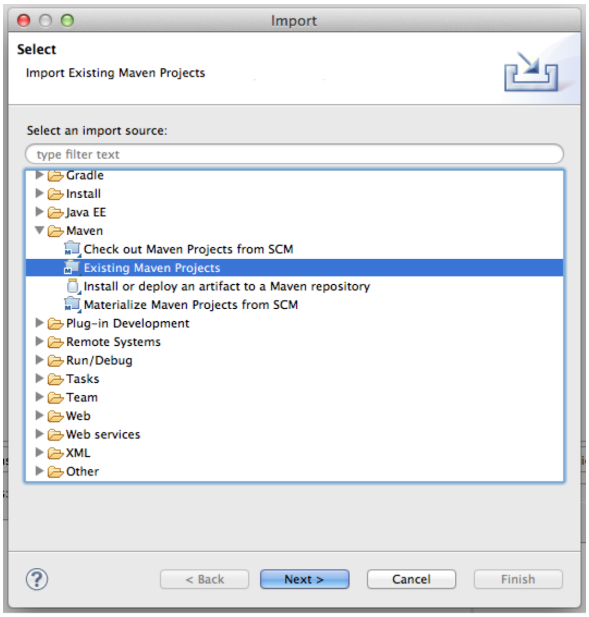
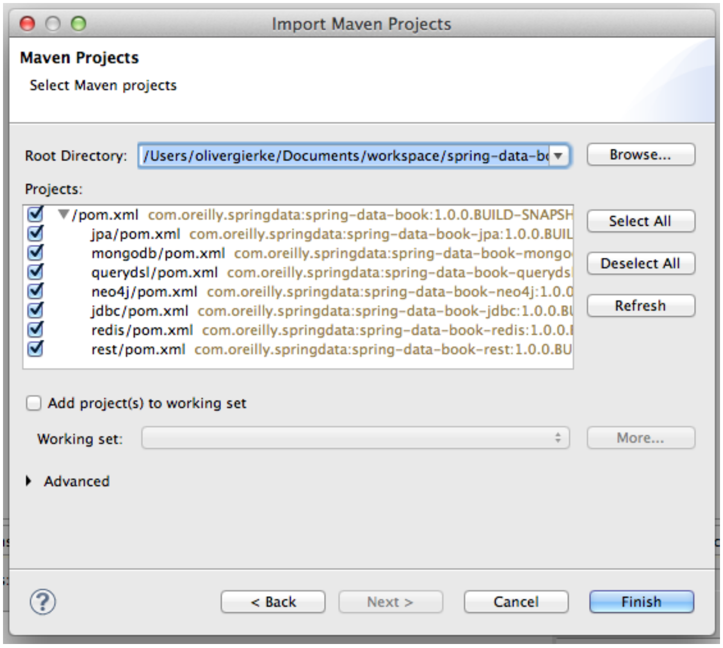


Figure 1-2. Importing Maven projects into Eclipse (step 1 of 2)

In the next window, select the folder in which you've just checked out the project using the Browse button. After you've done so, the pane right below should fill with the individual Maven modules listed and checked (Figure 1-3). Proceed by clicking on Finish, and STS will import the selected Maven modules into your workspace. It will also resolve the necessary dependencies and source folder according to the *pom.xml* file in the module's root directory.



```
☑ ▼/pom.xml  com.oreilly.springdata:spring-data-book:1.0.0.BUILD-SNAPSH
☑    jpa/pom.xml  com.oreilly.springdata:spring-data-book-jpa:1.0.0.BUIL
☑    mongodb/pom.xml  com.oreilly.springdata:spring-data-book-mongo
☑    querydsl/pom.xml  com.oreilly.springdata:spring-data-book-queryds
☑    neo4j/pom.xml  com.oreilly.springdata:spring-data-book-neo4j:1.0.0
☑    jdbc/pom.xml  com.oreilly.springdata:spring-data-book-jdbc:1.0.0.BU
☑    redis/pom.xml  com.oreilly.springdata:spring-data-book-redis:1.0.0.
☑    rest/pom.xml  com.oreilly.springdata:spring-data-book-rest:1.0.0.BU
```

Figure 1-3. Importing Maven projects into Eclipse (step 2 of 2)

You should eventually end up with a Package or Project Explorer looking something like Figure 1-4. The projects should compile fine and contain no red error markers.

The projects using Querydsl (see Chapter 5 for details) might still carry a red error marker. This is due to the m2eclipse plug-in needing additional information about when to execute the Querydsl-related Maven plug-ins in the IDE build life cycle. The integration for that can be installed from the m2e-querydsl extension update site; you'll find the most recent version of it at the project home page. Copy the link to the latest version listed there (0.0.3, at the time of this writing) and add it to the list of available update sites, as shown in Figure 1-5. Installing the feature exposed through that update site, restarting Eclipse, and potentially updating the Maven project configuration (right-click on the project→Maven→Update Project) should let you end up with all the projects without Eclipse error markers and building just fine.

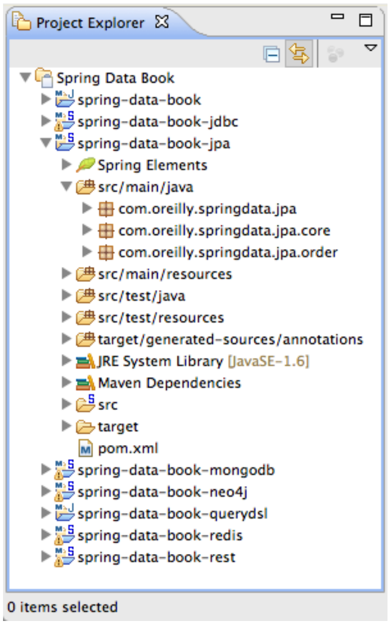
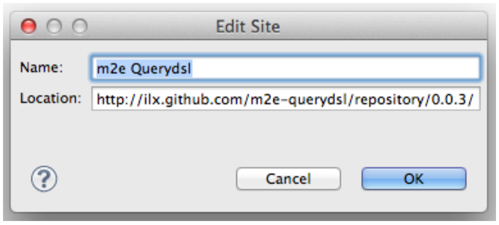


Figure 1-4. Eclipse Project Explorer with import finished



Name: m2e Querydsl

Location: http://ilx.github.com/m2e-querydsl/repository/0.0.3/

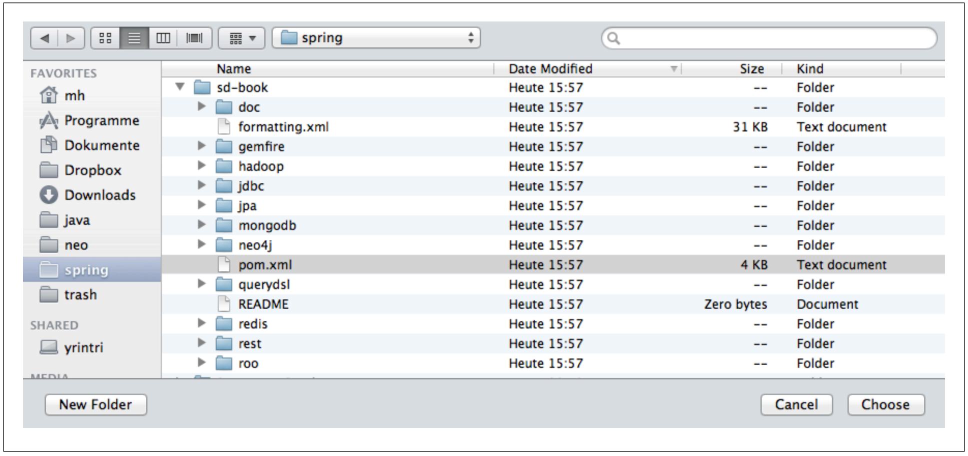
Figure 1-5. Adding the m2e-querydsl update site

IntelliJ IDEA

IDEA is able to open Maven project files directly without any further setup needed. Select the Open Project menu entry to show the dialog box (see Figure 1-6).

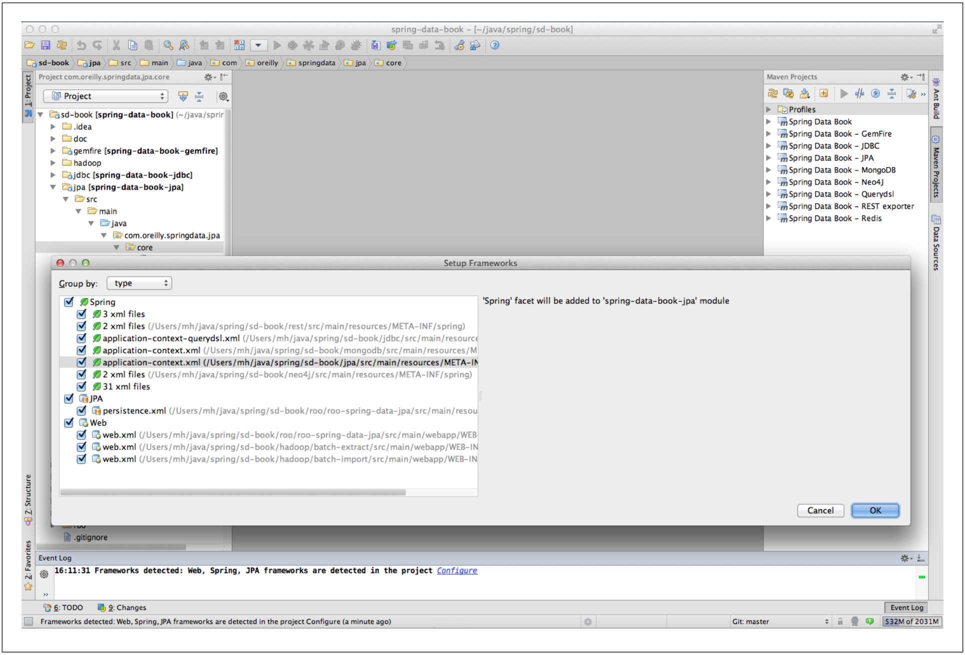
The IDE opens the project and fetches needed dependencies. In the next step (shown in Figure 1-7), it detects used frameworks (like the Spring Framework, JPA, WebApp); use the Configure link in the pop up or the Event Log to configure them.

The project is then ready to be used. You will see the Project view and the Maven Projects view, as shown in Figure 1-8. Compile the project as usual.



Name	Date Modified	Size	Kind
sd-book	Heute 15:57	--	Folder
doc	Heute 15:57	--	Folder
formatting.xml	Heute 15:57	31 KB	Text document
gemfire	Heute 15:57	--	Folder
hadoop	Heute 15:57	--	Folder
jdbc	Heute 15:57	--	Folder
jpa	Heute 15:57	--	Folder
mongodb	Heute 15:57	--	Folder
neo4j	Heute 15:57	--	Folder
pom.xml	Heute 15:57	4 KB	Text document
querydsl	Heute 15:57	--	Folder
README	Heute 15:57	Zero bytes	Document
redis	Heute 15:57	--	Folder
rest	Heute 15:57	--	Folder
roo	Heute 15:57	--	Folder

Figure 1-6. Importing Maven projects into IDEA (step 1 of 2)



Setup Frameworks

Group by: type

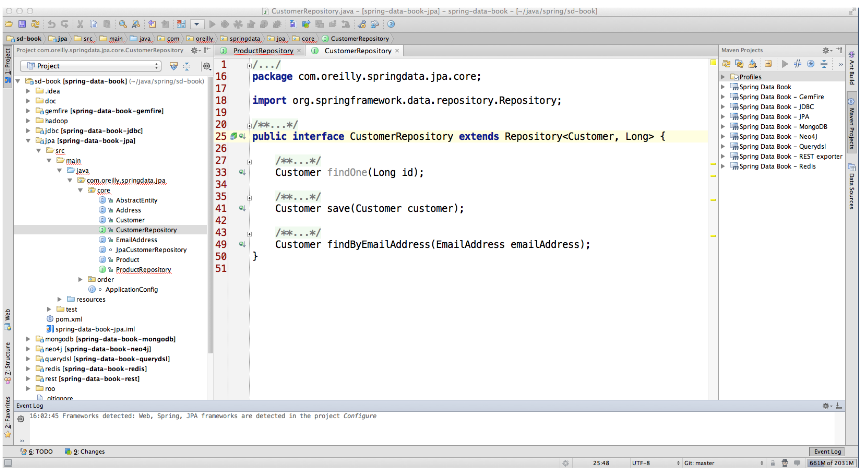
- [✓] Spring
  - [✓] 3 xml files
  - [✓] 2 xml files (/Users/mh/java/spring/sd-book/rest/src/main/resources/META-INF/spring)
  - [✓] application-context-querydsl.xml (/Users/mh/java/spring/sd-book/jdbc/src/main/resource
  - [✓] application-context.xml (/Users/mh/java/spring/sd-book/mongodb/src/main/resources/M
  - [✓] application-context.xml (/Users/mh/java/spring/sd-book/jpa/src/main/resources/META-IN
  - [✓] 2 xml files (/Users/mh/java/spring/sd-book/neo4j/src/main/resources/META-INF/spring)
  - [✓] 31 xml files
- [✓] JPA
  - [✓] persistence.xml (/Users/mh/java/spring/sd-book/roo/roo-spring-data-jpa/src/main/resou
- [✓] Web
  - [✓] web.xml (/Users/mh/java/spring/sd-book/roo/roo-spring-data-jpa/src/main/webapp/WEB
  - [✓] web.xml (/Users/mh/java/spring/sd-book/hadoop/batch-extract/src/main/webapp/WEB-IN
  - [✓] web.xml (/Users/mh/java/spring/sd-book/hadoop/batch-import/src/main/webapp/WEB-IN

'Spring' facet will be added to 'spring-data-book-jpa' module

```
16:11:31 Frameworks detected: Web, Spring, JPA frameworks are detected in the project Configure
```

Frameworks detected: Web, Spring, JPA frameworks are detected in the project Configure (a minute ago)

Figure 1-7. Importing Maven projects into IDEA (step 2 of 2)

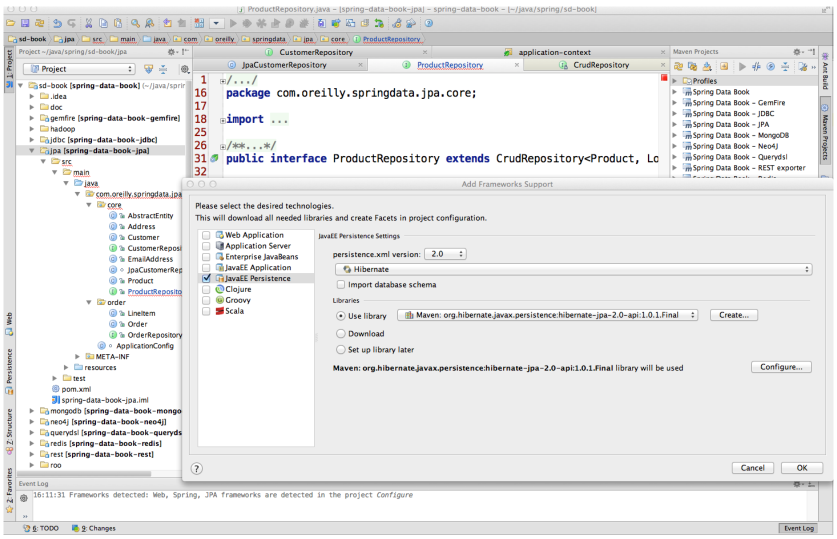


```
/.../
package com.oreilly.springdata.jpa.core;

import org.springframework.data.repository.Repository;

/**...*/
public interface CustomerRepository extends Repository<Customer, Long> {

    /**...*/
    Customer findOne(Long id);

    /**...*/
    Customer save(Customer customer);

    /**...*/
    Customer findByEmailAddress(EmailAddress emailAddress);
}
```

```
16:02:45 Frameworks detected: Web, Spring, JPA frameworks are detected in the project Configure
```

Figure 1-8. IDEA with the Spring Data Book project opened

Next you must add JPA support in the Spring Data JPA module to enable finder method completion and error checking of repositories. Just right-click on the module and choose Add Framework. In the resulting dialog box, check JavaEE Persistence support and select Hibernate as the persistence provider (Figure 1-9). This will create a *src/main/java/resources/META-INF/persistence.xml* file with just a persistence-unit setup.



```
/.../
package com.oreilly.springdata.jpa.core;

import ...

/**...*/
public interface ProductRepository extends CrudRepository<Product, Lo
```

Add Frameworks Support

Please select the desired technologies.
This will download all needed libraries and create Facets in project configuration.

- [ ] Web Application
- [ ] Application Server
- [ ] Enterprise JavaBeans
- [ ] JavaEE Application
- [x] JavaEE Persistence
- [ ] Clojure
- [ ] Groovy
- [ ] Scala

JavaEE Persistence Settings

persistence.xml version: 2.0

Hibernate

- [ ] Import database schema

Libraries

[✓ Use library] Maven: org.hibernate.javax.persistence:hibernate-jpa-2.0-api:1.0.1.Final

Download

Set up library later

Maven: org.hibernate.javax.persistence:hibernate-jpa-2.0-api:1.0.1.Final library will be used

```
16:11:31 Frameworks detected: Web, Spring, JPA frameworks are detected in the project Configure
```

Figure 1-9. Enable JPA support for the Spring Data JPA module

CHAPTER 2

Repositories: Convenient Data Access Layers

Implementing the data access layer of an application has been cumbersome for quite a while. Too much boilerplate code had to be written. Domain classes were anemic and not designed in a real object-oriented or domain-driven manner. The goal of the repository abstraction of Spring Data is to reduce the effort required to implement data access layers for various persistence stores significantly. The following sections will introduce the core concepts and interfaces of Spring Data repositories. We will use the Spring Data JPA module as an example and discuss the basic concepts of the repository abstraction. For other stores, make sure you adapt the examples accordingly.

Quick Start

Let's take the `Customer` domain class from our domain that will be persisted to an arbitrary store. The class might look something like Example 2-1.

Example 2-1. The Customer domain class

```
public class Customer {

  private Long id;
  private String firstname;
  private String lastname;
  private EmailAddress emailAddress;
  private Address address;

  …
}
```

A traditional approach to a data access layer would now require you to at least implement a repository class that contains necessary CRUD (Create, Read, Update, and Delete) methods as well as query methods to access subsets of the entities stored by applying restrictions on them. The Spring Data repository approach allows you to get rid of most of the implementation code and instead start with a plain interface definition for the entity's repository, as shown in Example 2-2.

Example 2-2. The CustomerRepository interface definition

```
public interface CustomerRepository extends Repository<Customer, Long> {
  …
}
```

As you can see, we extend the Spring Data `Repository` interface, which is just a generic marker interface. Its main responsibility is to allow the Spring Data infrastructure to pick up all user-defined Spring Data repositories. Beyond that, it captures the type of the domain class managed alongside the type of the ID of the entity, which will come in quite handy at a later stage. To trigger the autodiscovery of the interfaces declared, we use either the `<repositories />` element of the store-specific XML namespace (Example 2-3) or the related `@Enable…Repositories` annotation in case we're using JavaConfig (Example 2-4). In our sample case, we will use JPA. We just need to configure the XML element's `base-package` attribute with our root package so that Spring Data will scan it for repository interfaces. The annotation can also get a dedicated package configured to scan for interfaces. Without any further configuration given, it will simply inspect the package of the annotated class.

Example 2-3. Activating Spring Data repository support using XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <jpa:repositories base-package="com.acme.**.repository" />

</beans>
```

Example 2-4. Activating Spring Data repository support using Java Config

```
@Configuration
@EnableJpaRepositories
class ApplicationConfig {

}
```

Both the XML and JavaConfig configuration will need to be enriched with store-specific infrastructure bean declarations, such as a JPA `EntityManagerFactory`, a `DataSource`, and the like. For other stores, we simply use the corresponding namespace elements or annotations. The configuration snippet, shown in Example 2-5, will now cause the Spring Data repositories to be found, and Spring beans will be created that actually consist of proxies that will implement the discovered interface. Thus a client could now go ahead and get access to the bean by letting Spring simply autowire it.

Example 2-5. Using a Spring Data repository from a client

```
@Component
public class MyRepositoryClient {

  private final CustomerRepository repository;

  @Autowired
  public MyRepositoryClient(CustomerRepository repository) {
    Assert.notNull(repository);
    this.repository = repository;
  }

  …
}
```

With our `CustomerRepository` interface set up, we are ready to dive in and add some easy-to-declare query methods. A typical requirement might be to retrieve a `Customer` by its email address. To do so, we add the appropriate query method (Example 2-6).

Example 2-6. Declaring a query method

```
public interface CustomerRepository extends Repository<Customer, Long> {

  Customer findByEmailAddress(EmailAddress email);
}
```

The namespace element will now pick up the interface at container startup time and trigger the Spring Data infrastructure to create a Spring bean for it. The infrastructure will inspect the methods declared inside the interface and try to determine a query to be executed on method invocation. If you don't do anything more than declare the method, Spring Data will derive a query from its name. There are other options for query definition as well; you can read more about them in "Defining Query Methods" on page 16.

In Example 2-6, the query can be derived because we followed the naming convention of the domain object's properties. The `Email` part of the query method name actually refers to the `Customer` class's `emailAddress` property, and thus Spring Data will automatically derive `select C from Customer c where c.emailAddress = ?1` for the method declaration if you were using the JPA module. It will also check that you have valid property references inside your method declaration, and cause the container to fail to start on bootstrap time if it finds any errors. Clients can now simply execute the method, causing the given method parameters to be bound to the query derived from the method name and the query to be executed (Example 2-7).

Example 2-7. Executing a query method

```
@Component
public class MyRepositoryClient {

  private final CustomerRepository repository;

  …

  public void someBusinessMethod(EmailAddress email) {

    Customer customer = repository.findByEmailAddress(email);
  }
}
```

Defining Query Methods

Query Lookup Strategies

The interface we just saw had a simple query method declared. The method declaration was inspected by the infrastructure and parsed, and a store-specific query was derived eventually. However, as the queries become more complex, the method names would just become awkwardly long. For more complex queries, the keywords supported by the method parser wouldn't even suffice. Thus, the individual store modules ship with an `@Query` annotation, demonstrated in Example 2-8, that takes a query string in the store-specific query language and potentially allows further tweaks regarding the query execution.

Example 2-8. Manually defining queries using the @Query annotation

```
public interface CustomerRepository extends Repository<Customer, Long> {

  @Query("select c from Customer c where c.emailAddress = ?1")
  Customer findByEmailAddress(EmailAddress email);
}
```

Here we use JPA as an example and manually define the query that would have been derived anyway.

The queries can even be externalized into a properties file—*`$store`-named-queries.properties*, located in *META-INF*—where *`$store`* is a placeholder for *jpa*, *mongo*, *neo4j*, etc. The key has to follow the convention of *`$domainType.$methodName`*. Thus, to back our existing method with a externalized named query, the key would have to be `Customer.findByEmailAddress`. The `@Query` annotation is not needed if named queries are used.

Query Derivation

The query derivation mechanism built into the Spring Data repository infrastructure, shown in Example 2-9, is useful to build constraining queries over entities of the repository. We will strip the prefixes findBy, readBy, and getBy from the method and start parsing the rest of it. At a very basic level, you can define conditions on entity properties and concatenate them with And and Or.

Example 2-9. Query derivation from method names

```
public interface CustomerRepository extends Repository<Customer, Long> {

  List<Customer> findByEmailAndLastname(EmailAddress email, String lastname);
}
```

The actual result of parsing that method will depend on the data store we use. There are also some general things to notice. The expressions are usually property traversals combined with operators that can be concatenated. As you can see in Example 2-9, you can combine property expressions with And and Or. Beyond that, you also get support for various operators like Between, LessThan, GreaterThan, and Like for the property expressions. As the operators supported can vary from data store to data store, be sure to look at each store's corresponding chapter.

Property expressions

Property expressions can just refer to a direct property of the managed entity (as you just saw in Example 2-9). On query creation time, we already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. As seen above, Customers have Addresses with ZipCodes. In that case, a method name of:

```
List<Customer> findByAddressZipCode(ZipCode zipCode);
```

will create the property traversal x.address.zipCode. The resolution algorithm starts with interpreting the entire part (AddressZipCode) as a property and checks the domain class for a property with that name (with the first letter lowercased). If it succeeds, it just uses that. If not, it starts splitting up the source at the camel case parts from the right side into a head and a tail and tries to find the corresponding property (e.g., AddressZip and Code). If it finds a property with that head, we take the tail and continue building the tree down from there. Because in our case the first split does not match, we move the split point further to the left (from "AddressZip, Code" to "Address, Zip Code").

Although this should work for most cases, there might be situations where the algorithm could select the wrong property. Suppose our Customer class has an addressZip property as well. Then our algorithm would match in the first split, essentially choosing the wrong property, and finally fail (as the type of addressZip probably has no code property). To resolve this ambiguity, you can use an underscore (_) inside your method name to manually define traversal points. So our method name would end up like so:

```
List<Customer> findByAddress_ZipCode(ZipCode zipCode);
```

Pagination and Sorting

If the number of results returned from a query grows significantly, it might make sense to access the data in chunks. To achieve that, Spring Data provides a pagination API that can be used with the repositories. The definition for what chunk of data needs to be read is hidden behind the `Pageable` interface alongside its implementation `PageRequest`. The data returned from accessing it page by page is held in a `Page`, which not only contains the data itself but also metainformation about whether it is the first or last page, how many pages there are in total, etc. To calculate this metadata, we will have to trigger a second query as well as the initial one.

We can use the pagination functionality with the repository by simply adding a `Pageable` as a method parameter. Unlike the others, this will not be bound to the query, but rather used to restrict the result set to be returned. One option is to have a return type of `Page`, which will restrict the results, but require another query to calculate the metainformation (e.g., the total number of elements available). Our other option is to use `List`, which will avoid the additional query but won't provide the metadata. If you don't need pagination functionality, but plain sorting only, add a `Sort` parameter to the method signature (see Example 2-10).

Example 2-10. Query methods using Pageable and Sort

```
Page<Customer> findByLastname(String lastname, Pageable pageable);

List<Customer> findByLastname(String lastname, Sort sort);

List<Customer> findByLastname(String lastname, Pageable pageable);
```

The first method allows you to pass a `Pageable` instance to the query method to dynamically add paging to your statically defined query. Sorting options can either be handed into the method by the `Sort` parameter explicitly, or embedded in the `PageRequest` value object, as you can see in Example 2-11.

Example 2-11. Using Pageable and Sort

```
Pageable pageable = new PageRequest(2, 10, Direction.ASC, "lastname", "firstname");
Page<Customer> result = findByLastname("Matthews", pageable);

Sort sort = new Sort(Direction.DESC, "Matthews");
List<Customer> result = findByLastname("Matthews", sort);
```

Defining Repositories

So far, we have seen repository interfaces with query methods derived from the method name or declared manually, depending on the means provided by the Spring Data module for the actual store. To derive these queries, we had to extend a Spring Data–specific marker interface: `Repository`. Apart from queries, there is usually quite a bit of functionality that you need to have in your repositories: the ability to store objects, to delete them, look them up by ID, return all entities stored, or access them page by page. The easiest way to expose this kind of functionality through the repository interfaces is by using one of the more advanced repository interfaces that Spring Data provides:

`Repository`
: A plain marker interface to let the Spring Data infrastructure pick up user-defined repositories

`CrudRepository`
: Extends `Repository` and adds basic persistence methods like saving, finding, and deleting entities

`PagingAndSortingRepositories`
: Extends `CrudRepository` and adds methods for accessing entities page by page and sorting them by given criteria

Suppose we want to expose typical CRUD operations for the `CustomerRepository`. All we need to do is change its declaration as shown in Example 2-12.

Example 2-12. CustomerRepository exposing CRUD methods

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

  List<Customer> findByEmailAndLastname(EmailAddress email, String lastname);
}
```

The `CrudRepository` interface now looks something like Example 2-13. It contains methods to save a single entity as well as an `Iterable` of entities, finder methods for a single entity or all entities, and `delete(…)` methods of different flavors.

Example 2-13. CrudRepository

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {

  <S extends T> save(S entity);
  <S extends T> Iterable<S> save(Iterable<S> entities);

  T findOne(ID id);
  Iterable<T> findAll();

  void delete(ID id);
  void delete(T entity);
  void deleteAll();
}
```

Each of the Spring Data modules supporting the repository approach ships with an implementation of this interface. Thus, the infrastructure triggered by the namespace element declaration will not only bootstrap the appropriate code to execute the query methods, but also use an instance of the generic repository implementation class to back the methods declared in `CrudRepository` and eventually delegate calls to `save(…)`, `findAll()`, etc., to that instance. `PagingAndSortingRepository` (Example 2-14) now in turn extends `CrudRepository` and adds methods to allow handing instances of `Pageable` and `Sort` into the generic `findAll(…)` methods to actually access entities page by page.

Example 2-14. PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

To pull that functionality into the `CustomerRepository`, you'd simply extend `PagingAndSortingRepository` instead of `CrudRepository`.

Fine-Tuning Repository Interfaces

As we've just seen, it's very easy to pull in chunks of predefined functionality by extending the appropriate Spring Data repository interface. The decision to implement this level of granularity was actually driven by the trade-off between the number of interfaces (and thus complexity) we would expose in the event that we had separator interfaces for all find methods, all save methods, and so on, versus the ease of use for developers.

However, there might be scenarios in which you'd like to expose only the reading methods (the R in CRUD) or simply prevent the delete methods from being exposed in your repository interfaces. Spring Data now allows you to tailor a custom base repository with the following steps:

1. Create an interface either extending `Repository` or annotated with `@RepositoryDefinition`.
2. Add the methods you want to expose to it and make sure they actually match the signatures of methods provided by the Spring Data base repository interfaces.
3. Use this interface as a base interface for the interface declarations for your entities.

To illustrate this, let's assume we'd like to expose only the `findAll(…)` method taking a `Pageable` as well as the save methods. The base repository interface would look like Example 2-15.

Example 2-15. Custom base repository interface

```
@NoRepositoryBean
public interface BaseRepository<T, ID extends Serializable> extends Repository<T, ID> {

  Iterable<T> findAll(Pageable sort);

  <S extends T> S save(S entity);

  <S extends T> S save(Iterable<S> entities);
}
```

Note that we additionally annotated the interface with `@NoRepositoryBean` to make sure the Spring Data repository infrastructure doesn't actually try to create a bean instance for it. Letting your `CustomerRepository` extend this interface will now expose exactly the API you defined.

It's perfectly fine to come up with a variety of base interfaces (e.g., a `ReadOnlyRepository` or a `SaveOnlyRepository`) or even a hierarchy of them depending on the needs of your project. We usually recommend starting with locally defined CRUD methods directly in the concrete repository for an entity and then moving either to the Spring Data–provided base repository interfaces or tailor-made ones if necessary. That way, you keep the number of artifacts naturally growing with the project's complexity.

Manually Implementing Repository Methods

So far we have seen two categories of methods on a repository: CRUD methods and query methods. Both types are implemented by the Spring Data infrastructure, either by a backing implementation or the query execution engine. These two cases will probably cover a broad range of data access operations you'll face when building applications. However, there will be scenarios that require manually implemented code. Let's see how we can achieve that.

We start by implementing just the functionality that actually needs to be implemented manually, and follow some naming conventions with the implementation class (as shown in Example 2-16).

Example 2-16. Implementing custom functionality for a repository

```
interface CustomerRepositoryCustom {

  Customer myCustomMethod(…);
}


class CustomerRepositoryImpl implements CustomerRepositoryCustom {

  // Potentially wire dependencies

  public Customer myCustomMethod(…) {
    // custom implementation code goes here
  }
}
```

Neither the interface nor the implementation class has to know anything about Spring Data. This works pretty much the way that you would manually implement code with Spring. The most interesting piece of this code snippet in terms of Spring Data is that the name of the implementation class follows the naming convention to suffix the core repository interface's (`CustomerRepository` in our case) name with `Impl`. Also note that we kept both the interface as well as the implementation class as *package private* to prevent them being accessed from outside the package.

The final step is to change the declaration of our original repository interface to extend the just-introduced one, as shown in Example 2-17.

Example 2-17. Including custom functionality in the CustomerRepository

```
public interface CustomerRepository extends CrudRepository<Customer, Long>,
                                            CustomerRepositoryCustom { … }
```

Now we have essentially pulled the API exposed in `CustomerRepositoryCustom` into our `CustomerRepository`, which makes it the central access point of the data access API for `Customers`. Thus, client code can now call `CustomerRepository.myCustomMethod(…)`. But how does the implementation class actually get discovered and brought into the proxy to be executed eventually? The bootstrap process for a repository essentially looks as follows:

1. The repository interface is discovered (e.g., `CustomerRepository`).
2. We're trying to look up a bean definition with the name of the lowercase interface name suffixed by `Impl` (e.g., `customerRepositoryImpl`). If one is found, we'll use that.
3. If not, we scan for a class with the name of our core repository interface suffixed by `Impl` (e.g., `CustomerRepositoryImpl`, which will be picked up in our case). If one is found, we register this class as a Spring bean and use that.
4. The found custom implementation will be wired to the proxy configuration for the discovered interface and act as a potential target for method invocation.

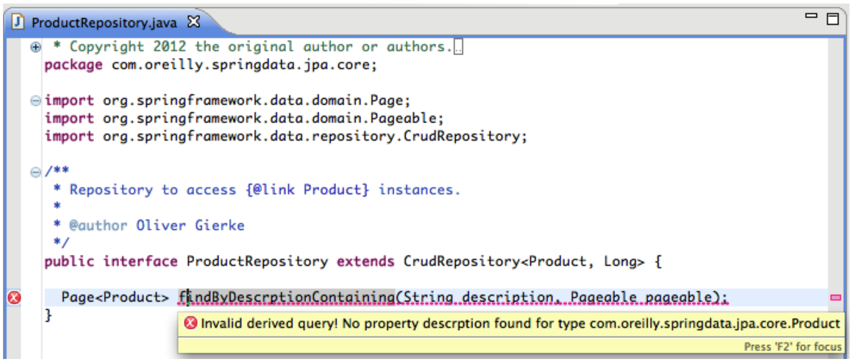
This mechanism allows you to easily implement custom code for a dedicated repository. The suffix used for implementation lookup can be customized on the XML namespace element or an attribute of the repository enabling annotation (see the individual store chapters for more details on that). The reference documentation also contains some material on how to implement custom behavior to be applied to multiple repositories.

IDE Integration

As of version 3.0, the Spring Tool Suite (STS) provides integration with the Spring Data repository abstraction. The core area of support provided for Spring Data by STS is the query derivation mechanism for finder methods. The first thing it helps you with to validate your derived query methods right inside the IDE so that you don't actually have to bootstrap an ApplicationContext, but can eagerly detect typos you introduce into your method names.

STS is a special Eclipse distribution equipped with a set of plug-ins to ease building Spring applications as much as possible. The tool can be downloaded from the project's website or installed into an plain Eclipse distribution by using the STS update site (based on Eclipse 3.8 or 4.2).

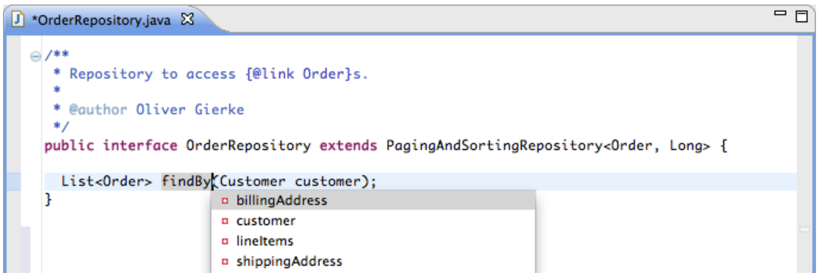
As you can see in Figure 2-1, the IDE detects that Descrption is not valid, as there is no such property available on the Product class. To discover these typos, it will analyze the Product domain class (something that bootstrapping the Spring Data repository infrastructure would do anyway) for properties and parse the method name into a property traversal tree. To avoid these kinds of typos as early as possible, STS's Spring Data support offers code completion for property names, criteria keywords, and concatenators like And and Or (see Figure 2-2).



```
 * Copyright 2012 the original author or authors.
package com.oreilly.springdata.jpa.core;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.CrudRepository;

/**
 * Repository to access {@link Product} instances.
 *
 * @author Oliver Gierke
 */
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescrptionContaining(String description, Pageable pageable);
}
```

Invalid derived query! No property descrption found for type com.oreilly.springdata.jpa.core.Product

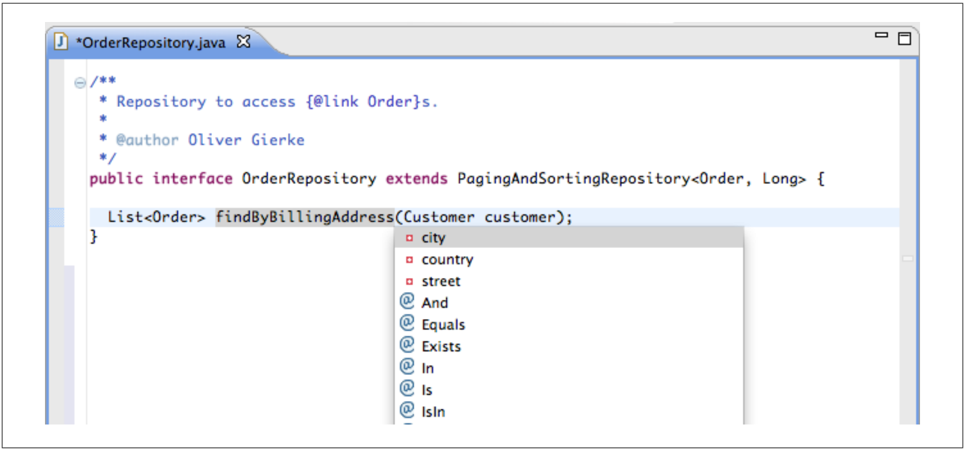
Figure 2-1. Spring Data STS derived query method name validation



```
/**
 * Repository to access {@link Order}s.
 *
 * @author Oliver Gierke
 */
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {

  List<Order> findBy(Customer customer);
}
```

Figure 2-2. Property code completion proposals for derived query methods

The `Order` class has a few properties that you might want to refer to. Assuming we'd like to traverse the `billingAddress` property, another Cmd+Space (or Ctrl+Space on Windows) would trigger a nested property traversal that proposes nested properties, as well as keywords matching the type of the property traversed so far (Figure 2-3). Thus, `String` properties would additionally get `Like` proposed.



```
/**
 * Repository to access {@link Order}s.
 *
 * @author Oliver Gierke
 */
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {

  List<Order> findByBillingAddress(Customer customer);
}
```

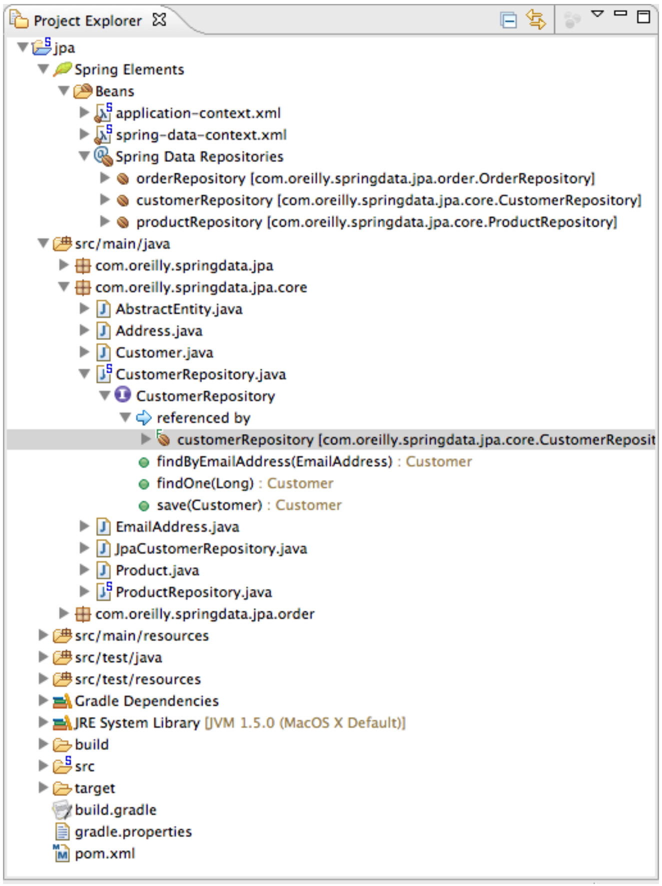
Figure 2-3. Nested property and keyword proposals

To put some icing on the cake, the Spring Data STS will make the repositories first-class citizens of your IDE navigator, marking them with the well-known Spring bean symbol. Beyond that, the Spring Elements node in the navigator will contain a dedicated Spring Data Repositories node to contain all repositories found in your application's configuration (see Figure 2-4).

As you can see, you can discover the repository interfaces at a quick glance and trace which configuration element they actually originate from.

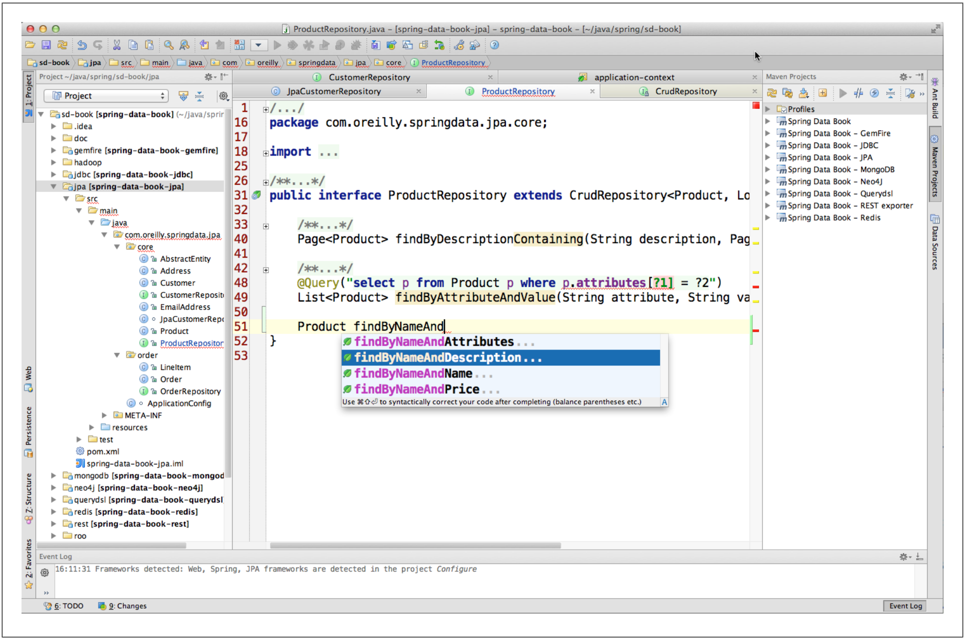
IntelliJ IDEA

Finally, with the JPA support enabled, IDEA offers repository finder method completion derived from property names and the available keyword, as shown in Figure 2-5.



```
Project Explorer
▼ jpa
  ▼ Spring Elements
    ▼ Beans
      ▶ application-context.xml
      ▶ spring-data-context.xml
      ▼ Spring Data Repositories
        ▶ orderRepository [com.oreilly.springdata.jpa.order.OrderRepository]
        ▶ customerRepository [com.oreilly.springdata.jpa.core.CustomerRepository]
        ▶ productRepository [com.oreilly.springdata.jpa.core.ProductRepository]
  ▼ src/main/java
    ▶ com.oreilly.springdata.jpa
    ▼ com.oreilly.springdata.jpa.core
      ▶ AbstractEntity.java
      ▶ Address.java
      ▶ Customer.java
      ▼ CustomerRepository.java
        ▼ CustomerRepository
          ▼ referenced by
            ▶ customerRepository [com.oreilly.springdata.jpa.core.CustomerReposit
          findByEmailAddress(EmailAddress) : Customer
          findOne(Long) : Customer
          save(Customer) : Customer
      ▶ EmailAddress.java
      ▶ JpaCustomerRepository.java
      ▶ Product.java
      ▶ ProductRepository.java
    ▶ com.oreilly.springdata.jpa.order
  ▶ src/main/resources
  ▶ src/test/java
  ▶ src/test/resources
  ▶ Gradle Dependencies
  ▶ JRE System Library [JVM 1.5.0 (MacOS X Default)]
  ▶ build
  ▶ src
  ▶ target
    build.gradle
    gradle.properties
    pom.xml
```

Figure 2-4. Eclipse Project Explorer with Spring Data support in STS



```
/.../
package com.oreilly.springdata.jpa.core;

import ...

/**...*/
public interface ProductRepository extends CrudRepository<Product, Lo

    /**...*/
    Page<Product> findByDescriptionContaining(String description, Pag

    /**...*/
    @Query("select p from Product p where p.attributes[?1] = ?2")
    List<Product> findByAttributeAndValue(String attribute, String va

    Product findByNameAnd
}
```

```
findByNameAndAttributes...
findByNameAndDescription...
findByNameAndName...
findByNameAndPrice...
Use ⌘⇧⏎ to syntactically correct your code after completing (balance parentheses etc.)
```

```
16:11:31 Frameworks detected: Web, Spring, JPA frameworks are detected in the project Configure
```

Figure 2-5. Finder method completion in IDEA editor

CHAPTER 3

Type-Safe Querying Using Querydsl

Writing queries to access data is usually done using Java `Strings`. The query languages of choice have been SQL for JDBC as well as HQL/JPQL for Hibernate/JPA. Defining the queries in plain `Strings` is powerful but quite error-prone, as it's very easy to introduce typos. Beyond that, there's little coupling to the actual query source or sink, so column references (in the JDBC case) or property references (in the HQL/JPQL context) become a burden in maintenance because changes in the table or object model cannot be reflected in the queries easily.

The Querydsl project tries to tackle this problem by providing a fluent API to define these queries. The API is derived from the actual table or object model but is highly store- and model-agnostic at the same time, so it allows you to create and use the query API for a variety of stores. It currently supports JPA, Hibernate, JDO, native JDBC, Lucene, Hibernate Search, and MongoDB. This versatility is the main reason why the Spring Data project integrates with Querydsl, as Spring Data also integrates with a variety of stores. The following sections will introduce you to the Querydsl project and its basic concepts. We will go into the details of the store-specific support in the store-related chapters later in this book.

Introduction to Querydsl

When working with Querydsl, you will usually start by deriving a metamodel from your domain classes. Although the library can also use plain `String` literals, creating the metamodel will unlock the full power of Querydsl, especially its type-safe property and keyword references. The derivation mechanism is based on the Java 6 Annotation Processing Tool (APT), which allows for hooking into the compiler and processing the sources or even compiled classes. For details, read up on that topic in "Generating the Query Metamodel" on page 30. To kick things off, we need to define a domain class like the one shown in Example 3-1. We model our `Customer` with a few primitive and nonprimitive properties.

Example 3-1. The Customer domain class

```
@QueryEntity
public class Customer extends AbstractEntity {

  private String firstname, lastname;
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  …
}
```

Note that we annotate the class with `@QueryEntity`. This is the default annotation, from which the Querydsl annotation processor generates the related query class. When you're using the integration with a particular store, the APT processor will be able to recognize the store-specific annotations (e.g., `@Entity` for JPA) and use them to derive the query classes. As we're not going to work with a store for this introduction and thus cannot use a store-specific mapping annotation, we simply stick with `@QueryEntity`. The generated Querydsl query class will now look like Example 3-2.

Example 3-2. The Querydsl generated query class

```
@Generated("com.mysema.query.codegen.EntitySerializer")
public class QCustomer extends EntityPathBase<Customer> {

  public static final QCustomer customer = new QCustomer("customer");
  public final QAbstractEntity _super = new QAbstractEntity(this);

  public final NumberPath<Long> id = _super.id;
  public final StringPath firstname = createString("firstname");
  public final StringPath lastname = createString("lastname");
  public final QEmailAddress emailAddress;

  public final SetPath<Address, QAddress> addresses =
        this.<Address, QAddress>createSet("addresses", Address.class, QAddress.class);

  …
}
```

You can find these classes in the *target/generated-sources/queries* folder of the module's sample project. The class exposes public `Path` properties and references to other query classes (e.g., `QEmailAddress`). This enables your IDE to list the available paths for which you might want to define predicates during code completion. You can now use these `Path` expressions to define reusable predicates, as shown in Example 3-3.

Example 3-3. Using the query classes to define predicates

```
QCustomer customer = QCustomer.customer;

BooleanExpression idIsNull = customer.id.isNull();
BooleanExpression lastnameContainsFragment = customer.lastname.contains("thews");
BooleanExpression firstnameLikeCart = customer.firstname.like("Cart");
```

```
EmailAddress reference = new EmailAddress("dave@dmband.com");
BooleanExpression isDavesEmail = customer.emailAddress.eq(reference);
```

We assign the static `QCustomer.customer` instance to the `customer` variable to be able to concisely refer to its property paths. As you can see, the definition of a predicate is clean, concise, and—most importantly—type-safe. Changing the domain class would cause the query metamodel class to be regenerated. Property references that have become invalidated by this change would become compiler errors and thus give us hints to places that need to be adapted. The methods available on each of the `Path` types take the type of the `Path` into account (e.g., the `like(…)` method makes sense only on `String` properties and thus is provided only on those).

Because predicate definitions are so concise, they can easily be used inside a method declaration. On the other hand, we can easily define predicates in a reusable manner, building up atomic predicates and combining them with more complex ones by using concatenating operators like `And` and `Or` (see Example 3-4).

Example 3-4. Concatenating atomic predicates

```
QCustomer customer = QCustomer.customer;

BooleanExpression idIsNull = customer.id.isNull();

EmailAddress reference = new EmailAddress("dave@dmband.com");
BooleanExpression isDavesEmail = customer.emailAddress.eq(reference);

BooleanExpression idIsNullOrIsDavesEmail = idIsNull.or(isDavesEmail);
```

We can use our newly written predicates to define a query for either a particular store or plain collections. As the support for store-specific query execution is mainly achieved through the Spring Data repository abstraction, have a look at "Integration with Spring Data Repositories" on page 32. We'll use the feature to query collections as an example now to keep things simple. First, we set up a variety of `Product`s to have something we can filter, as shown in Example 3-5.

Example 3-5. Setting up Products

```
Product macBook = new Product("MacBook Pro", "Apple laptop");
Product iPad = new Product("iPad", "Apple tablet");
Product iPod = new Product("iPod", "Apple MP3 player");
Product turntable = new Product("Turntable", "Vinyl player");

List<Product> products = Arrays.asList(macBook, iPad, iPod, turntable);
```

Next, we can use the Querydsl API to actually set up a query against the collection, which is some kind of filter on it (Example 3-6).

Example 3-6. Filtering Products using Querydsl predicates

```
QProduct $ = QProduct.product;
List<Product> result = from($, products).where($.description.contains("Apple")).list($);
```

```
assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

We're setting up a Querydsl `Query` using the `from(…)` method, which is a static method on the `MiniAPI` class of the *querydsl-collections* module. We hand it an instance of the query class for `Product` as well as the source collection. We can now use the `where(…)` method to apply predicates to the source list and execute the query using one of the `list(…)` methods (Example 3-7). In our case, we'd simply like to get back the `Product` instances matching the defined predicate. Handing `$.description` into the `list(…)` method would allow us to project the result onto the product's description and thus get back a collection of `String`s.

Example 3-7. Filtering Products using Querydsl predicates (projecting)

```
QProduct $ = QProduct.product;
BooleanExpression descriptionContainsApple = $.description.contains("Apple");
List<String> result = from($, products).where(descriptionContainsApple).list($.name);

assertThat(result, hasSize(3));
assertThat(result, hasItems("MacBook Pro", "iPad", "iPod"));
```

As we have discovered, Querydsl allows us to define entity predicates in a concise and easy way. These can be generated from the mapping information for a variety of stores as well as for plain Java classes. Querydsl's API and its support for various stores allows us to generate predicates to define queries. Plain Java collections can be filtered with the very same API.

Generating the Query Metamodel

As we've just seen, the core artifacts with Querydsl are the query metamodel classes. These classes are generated via the Annotation Processing Toolkit, part of the `javac` Java compiler in Java 6. The APT provides an API to programmatically inspect existing Java source code for certain annotations, and then call functions that in turn generate Java code. Querydsl uses this mechanism to provide special APT processor implementation classes that inspect annotations. Example 3-1 used Querydsl-specific annotations like `@QueryEntity` and `@QueryEmbeddable`. If we already have domain classes mapped to a store supported by Querydsl, then generating the metamodel classes will require no extra effort. The core integration point here is the annotation processor you hand to the Querydsl APT. The processors are usually executed as a build step.

Build System Integration

To integrate with Maven, Querydsl provides the *maven-apt-plugin*, with which you can configure the actual processor class to be used. In Example 3-8, we bind the `process` goal to the `generate-sources` phase, which will cause the configured processor class to