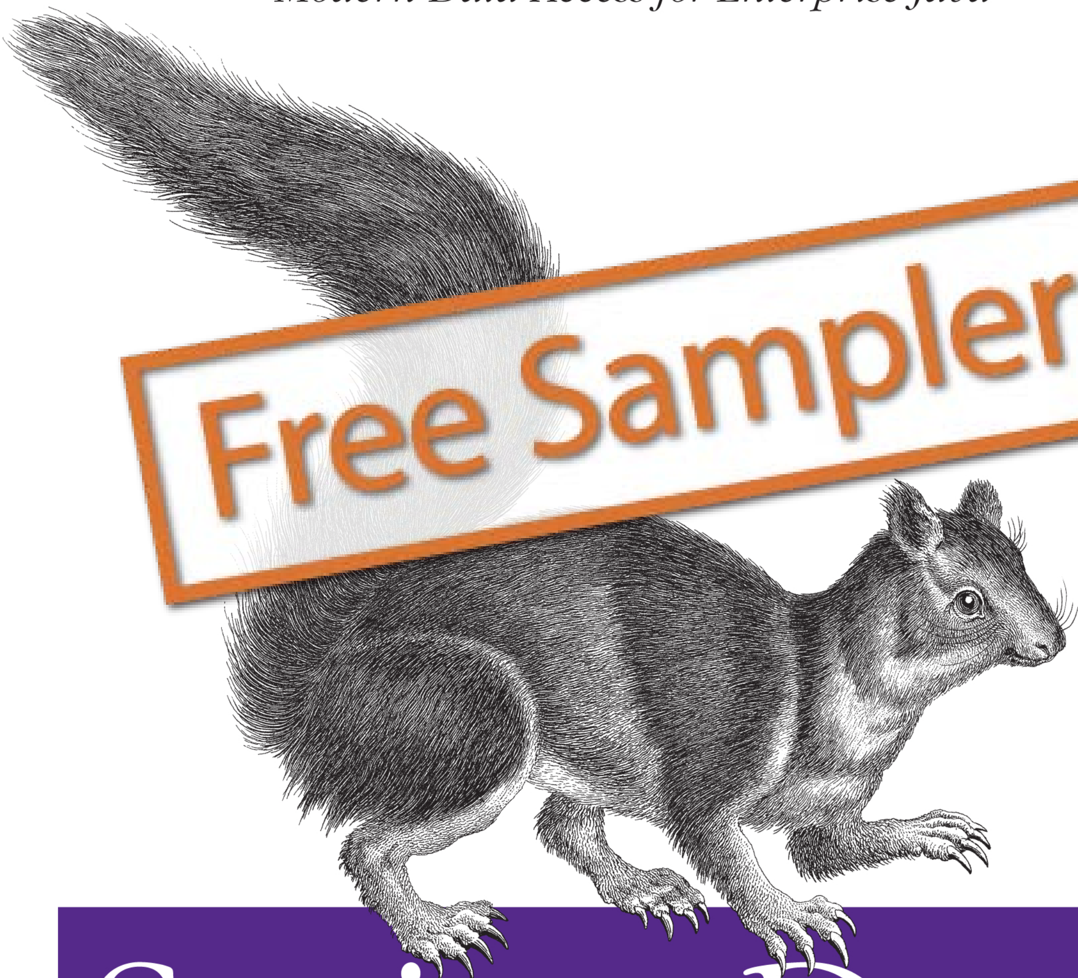
*Modern Data Access for Enterprise Java*



Free Sampler

# Spring Data

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

O'REILLY®

**Spring Data**

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



Printed in the United States of America.

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

# Table of Contents

## Part VI. Data Grids

# 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.

# 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.

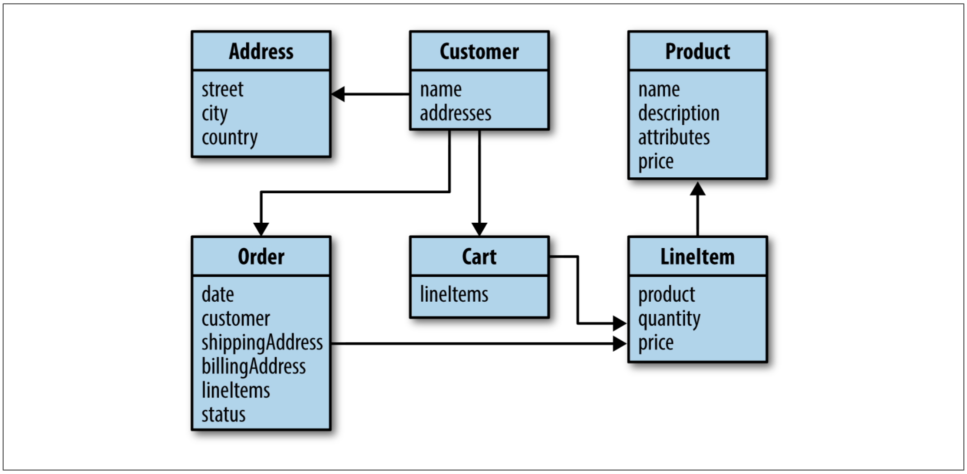



*Figure 1-1. The domain model*

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.

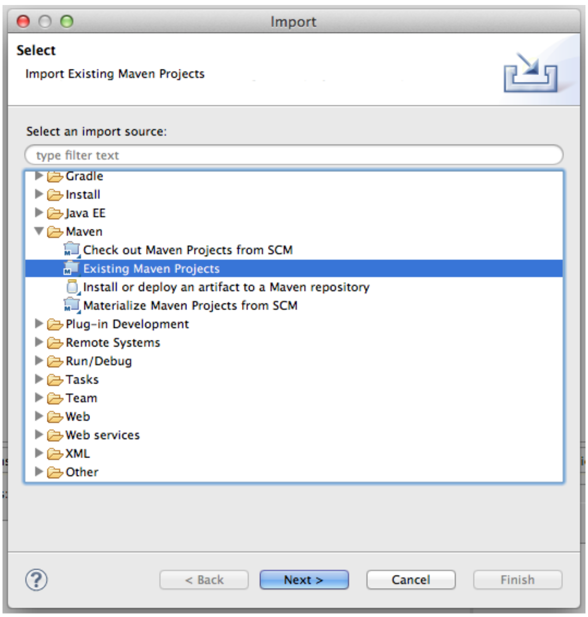



*Figure 1-2. Importing Maven projects into Eclipse (step 1 of 2)*

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.

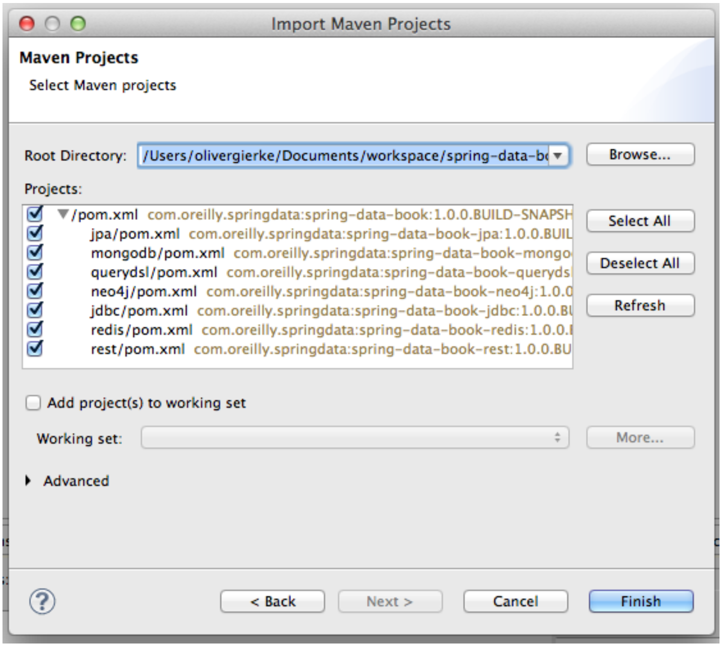



*Figure 1-3. Importing Maven projects into Eclipse (step 2 of 2)*

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.

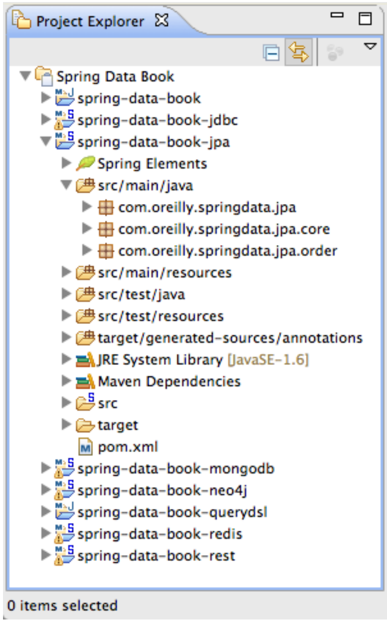



*Figure 1-4. Eclipse Project Explorer with import finished*

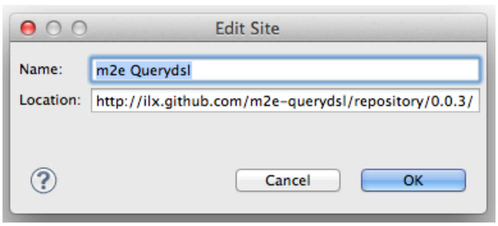



*Figure 1-5. Adding the m2e-querydsl update site*

### 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.

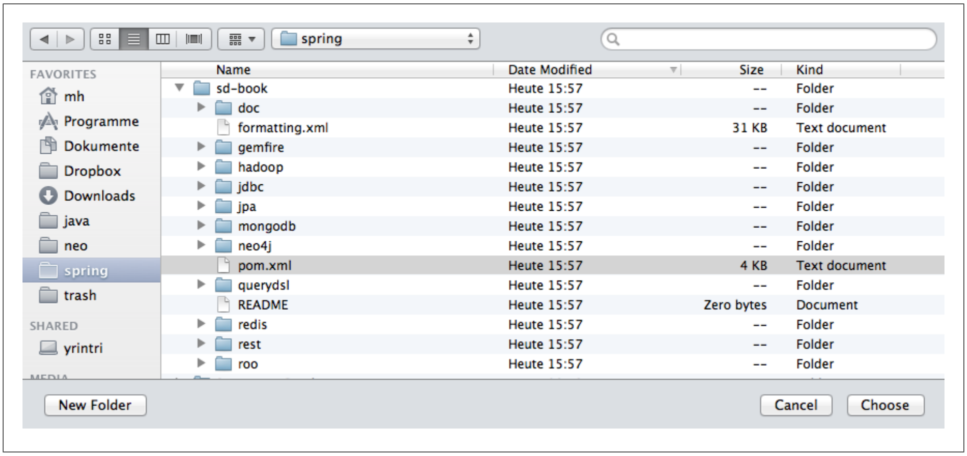



*Figure 1-6. Importing Maven projects into IDEA (step 1 of 2)*

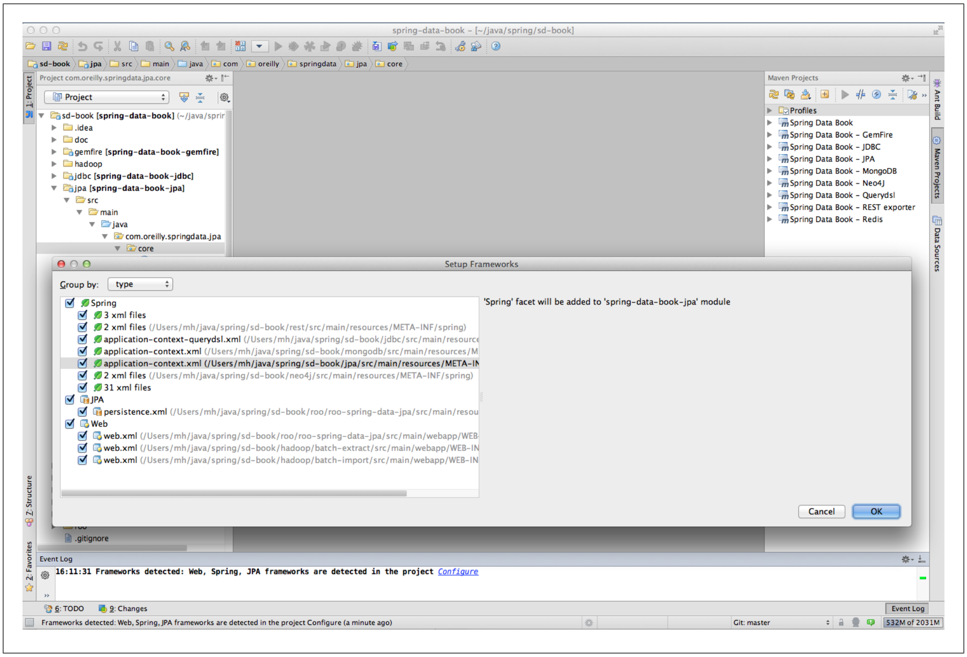



*Figure 1-7. Importing Maven projects into IDEA (step 2 of 2)*

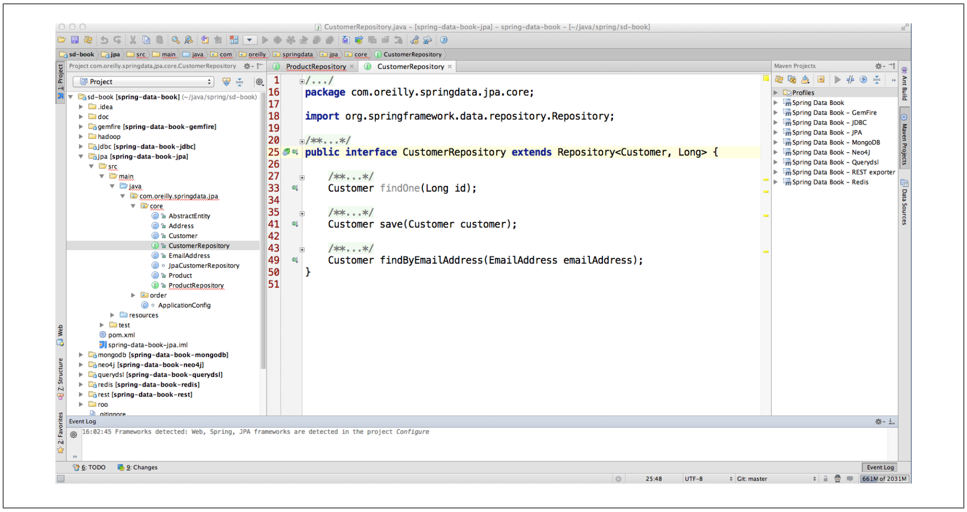



*Figure 1-8. IDEA with the Spring Data Book project opened*

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.

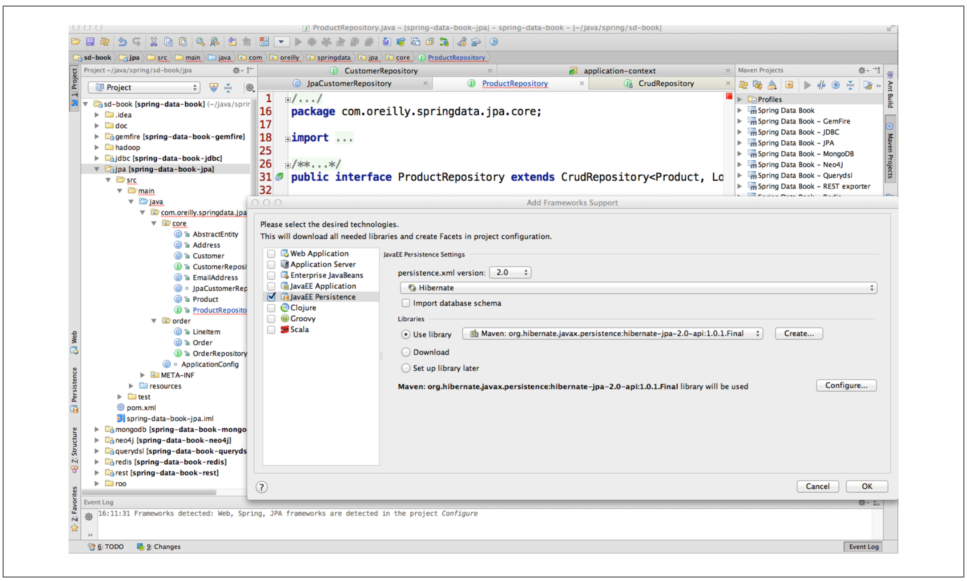



*Figure 1-9. Enable JPA support for the Spring Data JPA module*