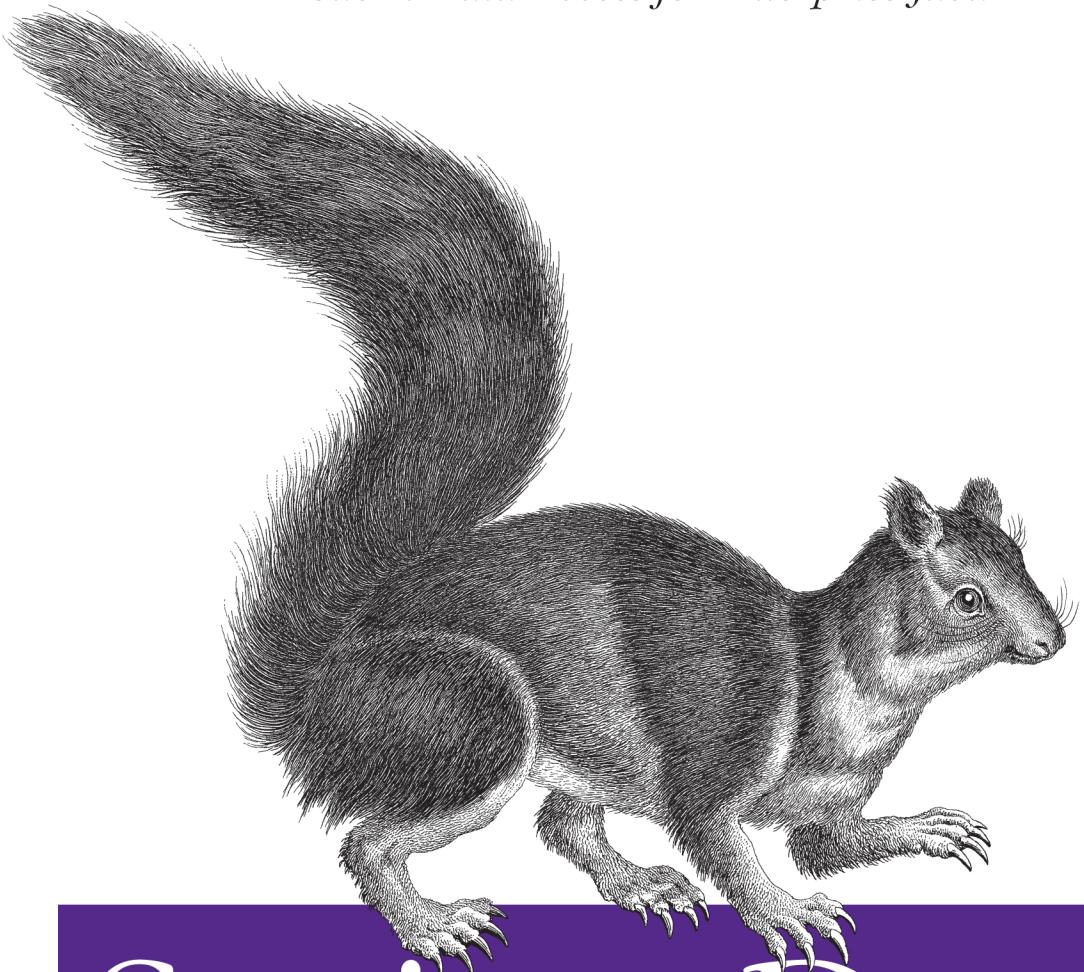


*Modern Data Access for Enterprise Java*



# Spring Data

O'REILLY®

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



---

# Datos de primavera

*Acceso moderno a datos para Java empresarial*

*Mark Pollack, Oliver Gierke, Thomas Risberg,  
Jon Brisbin y Michael Hunger*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokio

**Datos de primavera**

por Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin y Michael Hunger

Derechos de autor © 2013 Mark Pollack, Oliver Gierke, Thomas Risberg, Jonathan L. Brisbin, Michael Hunger. Todos los derechos reservados.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar con fines educativos, comerciales o promocionales de ventas. Las ediciones en línea también están disponibles para la mayoría de los títulos (<http://my.safaribooksonline.com>). Para obtener más información, comuníquese con nuestro departamento de ventas corporativo / institucional: 800-996-9938 o [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editores:** Mike Loukides y Meghan Blanchette

**Indexador:** Lucie Haskins

**Editor de producción:** Kristen Borg

**Diseñador de la portada:** Karen Montgomery

**Corrector de pruebas:** Rachel Monaghan

**Diseñador de interiores:** David Futato

**Ilustrador:** Rebecca Demarest

Octubre 2012: Primera edición.

**Historial de revisiones de la primera edición:**

2012-10-11 Primer lanzamiento

Ver <http://oreilly.com/catalog/errata.csp?isbn=9781449323950> para conocer los detalles del lanzamiento.

Nutshell Handbook, el logotipo del Nutshell Handbook y el logotipo de O'Reilly son marcas comerciales registradas de O'Reilly Media, Inc. *Spring Data*, la imagen de una ardilla gigante y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Muchas de las designaciones utilizadas por los fabricantes y vendedores para distinguir sus productos se reclaman como marcas comerciales. Cuando esas designaciones aparecen en este libro, y O'Reilly Media, Inc., tenía conocimiento de un reclamo de marca comercial, las designaciones se han impreso en mayúsculas o en mayúsculas.

Si bien se han tomado todas las precauciones en la preparación de este libro, el editor y los autores no asumen ninguna responsabilidad por errores u omisiones, o por daños que resulten del uso de la información aquí contenida.

ISBN: 978-1-449-32395-0

[LSI]

1349968177

*Gracias a mi esposa, Daniela, y a mis hijos, Gabriel y Alexandre, cuya paciencia conmigo robando el tiempo libre para "el libro" lo hizo posible.*

- Mark Pollack

*Me gustaría agradecer a mi familia, amigos, compañeros músicos y a todas las personas con las que he tenido el placer de trabajar hasta ahora; todo el equipo de Spring Data y SpringSource por este increíble viaje; y por último, pero en realidad antes que nada, Sabine, por su incansable tible amor y apoyo.*

- Oliver Gierke

*A mi esposa Carol y a mi hijo Alex, gracias por enriquecer mi vida y por todo su apoyo y ánimo.*

- Thomas Risberg



*A mi esposa, Tisha; mis hijos, Jack, Ben y Daniel; y mis hijas, Morgan y Hannah. Gracias por su amor, apoyo y paciencia.*

*Todo esto no valdría la pena sin ti.*

- Jon Brisbin

*Mi agradecimiento especial para Rod y Emil por iniciar el proyecto Spring Data y para Oliver por hacerlo genial. Mi familia siempre me apoya mucho en mi loco trabajo; Estoy muy agradecido de tener tal entender a las mujeres que me rodean.*

- Michael Hunger

*Me gustaría agradecer a mi esposa, Nanette y a mis hijos por su apoyo, paciencia y comprensión. Gracias también a Rod y a mis colegas del equipo de Spring Data por hacer posible todo esto.*

- David Turanski



---

# Tabla de contenido

Prólogo.....	xiii	Prefacio.....	xv
--------------	------	---------------	----

---

## Parte I. Antecedentes

<b>1. El proyecto Spring Data.....</b>	<b>3</b>
Acceso a datos NoSQL para temas generales de desarrolladores	3
de Spring	5
El dominio	6
El código de muestra	6
Importar el código fuente a su IDE	7
<b>2. Repositorios: cómodas capas de acceso a datos.....</b>	<b>13</b>
Inicio rápido	13
Definición de métodos de consulta	dieciséis
Estrategias de búsqueda de consultas	dieciséis
Derivación de consultas	17
Paginación y clasificación	18
Definición de repositorios	19
Interfaces de repositorio de ajuste fino	20
Implementación manual de métodos de repositorio	21
Integración	22
de IDE	22
IntelliJ IDEA	24
<b>3. Consulta de tipo seguro mediante Querydsl.....</b>	<b>27</b>
Introducción a Querydsl	27
Generando el metamodelo de consulta	30
Integración del sistema de construcción	30
Procesadores de anotaciones compatibles	31

---

Consultar almacenes mediante Querydsl	32
Integración con los repositorios de datos de Spring	32
Ejecución de predicados	33
Implementación manual de repositorios	34

---

## Parte II. Bases de datos relacionales

<b>4. Repositorios JPA.....</b>	<b>37</b>
El proyecto de muestra	37
El enfoque tradicional	42
Bootstrapping del código de muestra usando	44
repositorios de datos Spring	47
Transaccionalidad	50
Integración de Repository Querydsl	51
<b>5. Programación JDBC de tipo seguro con Querydsl SQL.....</b>	<b>53</b>
El proyecto de muestra y la configuración	53
La base de datos HyperSQL	54
El módulo SQL de la integración del sistema de	54
compilación de Querydsl	58
El esquema de la base de datos	59
La implementación de dominio del proyecto de muestra La	60
plantilla QueryDslJdbc	63
Ejecutando consultas	64
El comienzo de la implementación del repositorio Consulta	64
de un solo objeto	sesenta y cinco
La clase abstracta OneToManyResultSetExtractor La	67
implementación CustomerListExtractor	68
Las implementaciones para RowMappers que buscan	69
una lista de objetos	71
Operaciones de inserción, actualización y eliminación	71
Insertar con SQLInsertClause Actualizar con	71
SQLUpdateClause Eliminar filas con	72
SQLDeleteClause	73

---

## Parte III. NoSQL

<b>6. MongoDB: un almacén de documentos.....</b>	<b>77</b>
MongoDB en pocas palabras	77
Configuración de MongoDB	78
Usando el MongoDB Shell	79

---

El controlador Java de MongoDB	80
Configuración de la infraestructura utilizando el espacio de nombres Spring El	81
subsistema de mapeo	83
El modelo de dominio	83
Configuración de la indexación de la infraestructura de	89
mapeo	91
Conversión personalizada	91
MongoTemplate	94
Repositorios de Mongo	96
Configuración de infraestructura	96
Repositorios en detalle	97
Integración de Mongo Querydsl	99
<b>7. Neo4j: una base de datos de gráficos. . . . .</b>	<b>101</b>
Bases de datos de gráficos	101
Neo4j	102
Descripción general de Spring Data Neo4j Modelado	105
del dominio como un gráfico	106
Objetos de dominio persistentes con Spring Data Neo4j	111
Neo4jTemplate	112
Combinando gráficos y poder de repositorio	113
Operaciones básicas del repositorio de gráficos	115
Métodos de búsqueda derivados y anotados	116
Casos de uso de gráficos avanzados en el dominio de ejemplo	119
Varias funciones para un solo nodo	119
Categorías de productos y etiquetas como ejemplos de índices en gráfico	120
Aprovechamiento de intereses similares (filtrado colaborativo)	121
Recomendaciones	121
Transacciones, ciclo de vida de la entidad y estrategias de recuperación	122
Modo de mapeo avanzado	123
Trabajar con el servidor Neo4j a	124
partir de aquí	125
<b>8. Redis: un almacén de clave / valor. . . . .</b>	<b>127</b>
Redis en pocas palabras	127
Configuración de Redis	127
Uso del Shell de Redis	128
Conexión a Redis	129
Conversión de objetos	130
Mapeo de objetos	132
Contadores atómicos	134
Funcionalidad Pub / Sub	135
Escuchar y responder mensajes	135

---

**Parte IV. Desarrollo rápido de aplicaciones**

<b>9. Capas de persistencia con Spring Roo.....</b>	<b>141</b>
Una breve introducción a las capas de	141
persistencia de Roo	143
Inicio rápido	143
Usando Roo desde la línea de comandos	143
Usando Roo con Spring Tool Suite	145
Un ejemplo de repositorio de Spring Roo JPA	147
Creando el Proyecto	147
Configuración de la persistencia JPA	148
Creación de las entidades	148
Definición de repositorios	150
Creación de la capa web	150
ejecutando el ejemplo	151
Un ejemplo de repositorio Spring Roo MongoDB	152
Creando el Proyecto	153
Configurar la persistencia de MongoDB	153
Crear las entidades	153
Definición de repositorios	154
Creación de la capa web	154
ejecutando el ejemplo	154

<b>10. Exportador de repositorios REST.....</b>	<b>157</b>
---	------------

El proyecto de muestra	158
Interactuar con el exportador REST	160
Acceder a los productos	162
Acceder a los clientes	165
Acceso a pedidos	169

---

**Parte V. Big Data**

<b>11. Primavera para Apache Hadoop.....</b>	<b>175</b>
--	------------

Desafíos que se desarrollan con Hadoop Hello	176
World	177
Hola mundo revelado	179
Hola mundo usando Spring para Apache Hadoop Scripting	183
HDFS en la JVM	187
Combinación de secuencias de comandos HDFS y envío de trabajos	190

Programación de trabajos	191
Programación de trabajos de MapReduce con un TaskScheduler	191
Programación de trabajos de MapReduce con Quartz	192
<b>12. Análisis de datos con Hadoop. . . . .</b>	<b>195</b>
Usando Hive	195
Hola Mundo	196
Ejecución de un servidor Hive	197
Uso del cliente Hive Thrift mediante el cliente Hive JDBC	198
Análisis de archivos de registro de Apache mediante Hive mediante Pig	201
Hola Mundo	202
Ejecutando un PigServer	204
Control de la ejecución del script en tiempo de ejecución	205
Llamar a los scripts de Pig dentro de Spring Integration Data Pipelines	207
Análisis de archivos de registro de Apache usando Pig	209
Usando HBase	211
Hola Mundo	212
Uso del cliente Java HBase	214
<b>13. Creación de Big Data Pipelines con Spring Batch y Spring Integration. . . . .</b>	<b>219</b>
Recopilación y carga de datos en HDFS	219
Una introducción a la integración de Spring copiando archivos de registro	220
Flujos de eventos	222
Reenvío de eventos	226
administración	229
Introducción a Spring Batch	230
Procesamiento y carga de datos desde una base de datos	232
trabajo de Hadoop	234
Soporte de Spring Batch para Hadoop Wordcount como una aplicación de Spring Batch Hive y Pig Steps	238
Exportación de datos desde HDFS	240
De HDFS a JDBC De HDFS a MongoDB	242
Recopilación y carga de datos en Splunk	243

---

## Parte VI. Cuadrículas de datos

<b>14. GemFire: una cuadrícula de datos distribuidos.</b> . . . . .	<b>255</b>
GemFire en pocas palabras	255
Caches y regiones	257
Cómo conseguir GemFire	257
Configuración de GemFire con el espacio de nombres Spring XML	258
Configuración de caché	258
Configuración de la región	263
Configuración del cliente de caché	265
Configuración del servidor de caché	267
Configuración WAN	267
Configuración de almacenamiento en disco	268
Acceso a datos con el uso del repositorio	269
GemfireTemplate	271
POJOMapping	271
Crear un repositorio	272
Serialización PDX	272
Soporte de consulta continua	273
<b>Bibliografía.</b> . . . . .	<b>275</b>
	<b>Índice.</b> . . . . .
	<b>277</b>

---

# Prefacio

Vivimos en tiempos interesantes. Los nuevos procesos comerciales están generando nuevos requisitos. Los supuestos familiares están amenazados, entre ellos, que la base de datos relacional debería ser la opción predeterminada para la persistencia. Si bien esto está ahora ampliamente aceptado, no está nada claro cómo avanzar con eficacia en el nuevo mundo.

La proliferación de opciones de almacenamiento de datos crea fragmentación. Muchas tiendas más nuevas requieren más esfuerzo de los desarrolladores de lo que los desarrolladores de Java están acostumbrados con respecto al acceso a los datos, introduciendo en la aplicación cosas que normalmente se hacen en una base de datos relacional.

Este libro le ayuda a comprender esta nueva realidad. Proporciona una excelente descripción general del mundo del almacenamiento actual en el contexto del hardware actual y explica por qué las tiendas NoSQL son importantes para resolver los problemas comerciales modernos.

Debido a la identificación del lenguaje con el mercado empresarial a menudo conservador (y quizás también debido a la sofisticación de las soluciones de mapeo relacional de objetos de Java [ORM]), los desarrolladores de Java tradicionalmente han recibido un servicio pobre en el espacio NoSQL. Afortunadamente, esto está cambiando, lo que lo convierte en un libro importante y oportuno. Spring Data es un proyecto importante, con el potencial de ayudar a los desarrolladores a superar nuevos desafíos.

Muchos de los valores que han convertido a Spring en la plataforma preferida para los desarrolladores empresariales de Java ofrecen un beneficio particular en un mundo de soluciones de persistencia fragmentadas. Parte del valor de Spring es cómo aporta consistencia (sin descender a un mínimo común denominador) en su enfoque de las diferentes tecnologías con las que se integra. Una "forma primaveral" distinta ayuda a acortar la curva de aprendizaje para los desarrolladores y simplifica el mantenimiento del código. Si ya está familiarizado con Spring, encontrará que Spring Data facilita su exploración y adopción de tiendas desconocidas. Si aún no está familiarizado con Spring, esta es una buena oportunidad para ver cómo Spring puede simplificar su código y hacerlo más consistente.

Los autores están calificados de manera única para explicar Spring Data, siendo los líderes del proyecto. Aportan una combinación de profundo conocimiento y participación de Spring y una experiencia íntima con una variedad de almacenes de datos modernos. Hacen un buen trabajo al explicar la motivación de Spring Data y cómo continúa la misión que Spring ha perseguido durante mucho tiempo con respecto al acceso a los datos. Existe una valiosa cobertura de cómo Spring Data funciona con otras partes de

Spring, como Spring Integration y Spring Batch. El libro también proporciona mucho valor que va más allá de Spring, por ejemplo, las discusiones sobre el concepto de repositorio, los méritos de las consultas con seguridad de tipos y por qué la API de persistencia de Java (JPA) no es apropiada como una solución general de acceso a datos.

Si bien este es un libro sobre el acceso a los datos en lugar de trabajar con NoSQL, muchos de ustedes encontrarán el material NoSQL más valioso, ya que presenta temas y códigos con los que probablemente estén menos familiarizados. Todo el contenido está actualizado al minuto y los temas importantes incluyen bases de datos de documentos, bases de datos de gráficos, almacenes de clave / valor, Hadoop y la estructura de datos de Gemfire.

Los programadores somos criaturas prácticas y aprendemos mejor cuando podemos ser prácticos. El libro tiene una inclinación práctica bienvenida. Al principio, los autores muestran cómo hacer que el código de muestra funcione en los dos entornos de desarrollo integrados (IDE) de Java líderes, incluidas prácticas capturas de pantalla. Explican los requisitos relacionados con los controladores de la base de datos y la configuración básica de la base de datos. Aplaudo su elección de alojar el código de muestra en GitHub, haciéndolo accesible y navegable universalmente. Dados los muchos temas que cubre el libro, los ejemplos bien diseñados ayudan mucho a unir las cosas.

El énfasis en el desarrollo práctico también es evidente en el capítulo sobre Spring Roo, la solución de desarrollo rápido de aplicaciones (RAD) del equipo de Spring. La mayoría de los usuarios de Roo están familiarizados con cómo se puede utilizar Roo con una arquitectura JPA tradicional; los autores muestran cómo la productividad de Roo puede extenderse más allá de las bases de datos relacionales.

Cuando haya terminado este libro, tendrá una comprensión más profunda de por qué el acceso a los datos modernos se está volviendo más especializado y fragmentado, las principales categorías de almacenes de datos NoSQL, cómo Spring Data puede ayudar a los desarrolladores de Java a operar de manera efectiva en este nuevo entorno y dónde para buscar información más profunda sobre temas individuales en los que está particularmente interesado. Lo más importante es que tendrás un gran comienzo para tu propia exploración del código.

- Rod Johnson  
Creador, Spring Framework

---

# Prefacio

## Descripción general del panorama de acceso a nuevos datos

El panorama del acceso a los datos durante los últimos siete años ha cambiado drásticamente. Las bases de datos relacionales, el corazón del almacenamiento y procesamiento de datos en la empresa durante más de 30 años, ya no son el único juego en la ciudad. Los últimos siete años han visto el nacimiento

Y en algunos casos la muerte, de muchos almacenes de datos alternativos que se utilizan en aplicaciones empresariales de misión crítica. Estos nuevos almacenes de datos se han diseñado específicamente para resolver problemas de acceso a datos que las bases de datos relacionales no pueden manejar con tanta eficacia.

Un ejemplo de un problema que lleva las bases de datos relacionales tradicionales al límite es la escala. ¿Cómo se almacenan cientos o miles de terabytes (TB) en una base de datos relacional? La respuesta nos recuerda la vieja broma en la que el paciente dice: "Doctor, me duele cuando hago esto", y el médico dice: "¡Entonces no haga eso!" Dejando de lado las bromas, ¿qué está impulsando la necesidad de almacenar tanta información? En 2001, IDC informó que "la cantidad de información creada y replicada superará los 1,8 zettabytes y más del doble cada dos años".<sup>1</sup> Los nuevos tipos de datos van desde archivos multimedia hasta archivos de registro, datos de sensores (RFID, GPS, telemetría ...), tweets en Twitter y publicaciones en Facebook. Si bien los datos que se almacenan en bases de datos relacionales siguen siendo cruciales para la empresa, estos nuevos tipos de datos no se almacenan en bases de datos relacionales.

Si bien las demandas generales de los consumidores impulsan la necesidad de almacenar grandes cantidades de archivos multimedia, las empresas encuentran importante almacenar y analizar muchas de estas nuevas fuentes de datos. En los Estados Unidos, las empresas de todos los sectores tienen al menos 100 TB de datos almacenados y muchas tienen más de 1 petabyte (PB).<sup>2</sup> El consenso general es que existen importantes beneficios finales para que las empresas analicen continuamente estos datos. Por ejemplo, las empresas pueden comprender mejor el comportamiento de sus productos si los productos mismos envían mensajes de "teléfono a casa" sobre su salud. Para comprender mejor a sus clientes, las empresas pueden incorporar datos de redes sociales en sus procesos de toma de decisiones. Esto ha llevado a algunos medios importantes

---

1. IDC; *Extraer valor del caos*. 2011.

2. IDC; Oficina de Estadísticas Laborales de EE. UU.

informes, por ejemplo, sobre por qué Orbitz muestra más [opciones de hotel caras para los usuarios de Mac](#) y cómo [Target puede predecir cuándo una de sus clientes dará a luz pronto](#), lo que permite a la empresa enviar por correo talonarios de cupones a la casa del cliente antes de que los registros públicos de nacimiento estén disponibles.

*Big data* generalmente se refiere al proceso en el cual se almacenan grandes cantidades de datos, se mantienen en forma cruda y se analizan y combinan continuamente con otras fuentes de datos para proporcionar una comprensión más profunda de un dominio en particular, ya sea de naturaleza comercial o científica.

Muchas empresas y laboratorios científicos venían realizando este proceso antes del término *big data* se puso de moda. Lo que hace que el proceso actual sea diferente al anterior es que el valor derivado de la inteligencia del análisis de datos es mayor que los costos de hardware. Ya no es necesario comprar 40K por CPUbox para realizar este tipo de análisis de datos; Los grupos de hardware básico ahora cuestan \$ 1k por CPU. Para conjuntos de datos grandes, el costo de la red de área de almacenamiento (SAN) o el almacenamiento de área de red (NAS) se vuelve prohibitivo: \$ 1 a \$ 10 por gigabyte, mientras que el disco local cuesta solo \$ 0.05 por gigabyte con la replicación incorporada en la base de datos en lugar del hardware. Las tasas de transferencia de datos agregadas para clústeres de hardware básico que usan disco local también son significativamente más altas que los sistemas basados en SAN o NAS, 500 veces más rápido para sistemas de precio similar. Por el lado del software, la mayoría de las nuevas tecnologías de acceso a datos son de código abierto.

Otra área problemática que los nuevos almacenes de datos han identificado con las bases de datos relacionales es el modelo de datos relacionales. Si está interesado en analizar el gráfico social de millones de personas, ¿no parece bastante natural considerar el uso de una base de datos de gráficos para que la implementación modele más el dominio? ¿Qué sucede si los requisitos lo impulsan continuamente a cambiar su sistema de administración de base de datos relacional (RDBMS) esquema y capa de mapeo relacional de objetos (ORM)? Quizás una base de datos de documentos "sin esquema" reducirá la complejidad del mapeo de objetos y proporcionará un sistema más fácilmente evolutivo en comparación con el modelo relacional más rígido. Si bien cada una de las nuevas bases de datos es única a su manera, puede proporcionar una taxonomía aproximada en la mayoría de ellos basada en su modelos de datos. Los campos básicos en los que caen son:

#### *Valor clave*

Un modelo de datos familiar, muy parecido a una tabla hash.

#### *Familia de columnas*

Un modelo de datos de clave / valor ampliado en el que el tipo de datos de valor también puede ser una secuencia de pares de clave / valor.

#### *Documento*

Colecciones que contienen datos semiestructurados, como XML o JSON.

#### *Grafico*

Basado en la teoría de grafos. El modelo de datos tiene nodos y bordes, cada uno de los cuales puede tener propiedades.

El nombre general con el que se han agrupado estas nuevas bases de datos es "Bases de datos NoSQL". En retrospectiva, este nombre, aunque es pegadizo, no es muy exacto porque parece implicar que no puede consultar la base de datos, lo cual no es cierto. Refleja el cambio básico que se aleja del modelo de datos relacionales, así como un cambio general que se aleja de las características ACID (atomicidad, consistencia, aislamiento, durabilidad) de las bases de datos relacionales.

Uno de los factores que impulsan el alejamiento de las características de ACID es la aparición de aplicaciones que dan mayor prioridad al escalado de escrituras y al funcionamiento parcial del sistema, incluso cuando algunas partes del sistema han fallado. Si bien el escalado de las lecturas en una base de datos relacional se puede lograr mediante el uso de cachés en memoria que están al frente de la base de datos, escalar las escrituras es mucho más difícil. Para ponerle una etiqueta, estas nuevas aplicaciones favorecen un sistema que tiene la semántica denominada "BASE", donde la sigla representa

*básicamente disponible, escalable, eventualmente consistente*. Las cuadrículas de datos distribuidos con un modelo de datos clave / valor generalmente no se han agrupado en esta nueva ola de bases de datos NoSQL. Sin embargo, ofrecen características similares a las bases de datos NoSQL en términos de la escala de datos que pueden manejar, así como características de computación distribuida que colocan la potencia de computación y los datos.

Como puede ver en esta breve introducción al nuevo panorama de acceso a datos, se está produciendo una revolución, que para los fanáticos de los datos es bastante emocionante. Las bases de datos relacionales no están muertas; siguen siendo fundamentales para el funcionamiento de muchas empresas y lo seguirán siendo durante bastante tiempo. Las tendencias, sin embargo, son muy claras: las nuevas tecnologías de acceso a datos están resolviendo problemas que las bases de datos relacionales tradicionales no pueden, por lo que necesitamos ampliar nuestro conjunto de habilidades como desarrolladores y tener un pie en ambos campos.

Spring Framework tiene una larga historia en la simplificación del desarrollo de aplicaciones Java, en particular para escribir capas de acceso a datos basadas en RDBMS que utilizan conectividad de base de datos Java (JDBC) o mapeadores relacionales de objetos. En este libro, nuestro objetivo es ayudar a los desarrolladores a entender cómo desarrollar de forma eficaz aplicaciones Java en una amplia gama de estas nuevas tecnologías. El proyecto Spring Data aborda directamente estas nuevas tecnologías para que pueda extender su conocimiento existente de Spring a ellas, o tal vez aprender más sobre Spring como un subproducto del uso de Spring Data. Sin embargo, no deja atrás la base de datos relacional. Spring Data también proporciona un amplio conjunto de nuevas características para el soporte RDBMS de Spring.

## Cómo leer este libro

Este libro está destinado a brindarle una introducción práctica al proyecto Spring Data, cuya misión principal es permitir que los desarrolladores de Java utilicen herramientas de procesamiento y manipulación de datos de última generación, pero también usen bases de datos tradicionales en un estado de -la manera del arte. Comenzaremos presentándole el proyecto, describiendo la motivación principal de SpringSource y el equipo. También describiremos el modelo de dominio de los proyectos de muestra que se adaptan a cada uno de los capítulos posteriores, así como cómo acceder y configurar el código ([Capítulo 1](#)).

Luego, discutiremos los conceptos generales de los repositorios de Spring Data, ya que son un tema común en las diversas partes del proyecto específicas de la tienda ([Capítulo 2](#)). Lo mismo se aplica a QueryDSL, que se analiza en general en [Capítulo 3](#). Estos dos capítulos proporcionan una base sólida para explorar la integración específica de la tienda de la abstracción del repositorio y la funcionalidad de consulta avanzada.

Para iniciar a los desarrolladores de Java en un terreno conocido, luego dedicaremos algún tiempo a tecnologías de persistencia tradicionales como JPA ([Capítulo 4](#)) y JDBC ([Capítulo 5](#)). Esos capítulos describen las características que agregan los módulos Spring Data además del soporte JPA y JDBC ya existente proporcionado por Spring.

Una vez que hayamos terminado, presentamos algunas de las tiendas NoSQL compatibles con el proyecto Spring Data: MongoDB como ejemplo de una base de datos de documentos ([Capítulo 6](#)), Neo4j como ejemplo de una base de datos gráfica ([Capítulo 7](#)) y Redis como ejemplo de almacén de clave / valor ([Capítulo 8](#)). HBase, una base de datos de familias de columnas, se trata en un capítulo posterior ([Capítulo 12](#)). Estos capítulos describen la asignación de clases de dominio a las estructuras de datos específicas de la tienda, interactuando fácilmente con la tienda a través de la interfaz de programación de aplicaciones (API) proporcionada y utilizando la abstracción del repositorio.

Luego, le presentaremos el exportador REST de Spring Data ([Capítulo 10](#)) así como la integración de Spring Roo ([Capítulo 9](#)). Ambos proyectos se basan en la abstracción del repositorio y le permiten exportar fácilmente entidades administradas por Spring Data a la Web, ya sea como un servicio web de transferencia de estado representacional (REST) o como respaldo a una aplicación web construida por Spring Roo.

A continuación, el libro hace un recorrido por el mundo de los macrodatos, en particular Hadoop y Spring para Apache Hadoop. Le presentará el uso de casos implementados con Hadoop y le mostrará cómo el módulo Spring Data facilita el trabajo con Hadoop de manera significativa ([Capítulo 11](#)). Esto conduce a un ejemplo más complejo de cómo construir una canalización de big data utilizando Spring Batch y Spring Integration: proyectos que entran en juego muy bien en escenarios de procesamiento de big data ([Capítulo 12](#) y [Capítulo 13](#)).

El capítulo final analiza el soporte de Spring Data para Gemfire, una solución de cuadrícula de datos distribuidos ([Capítulo 14](#)).

## Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

### *Itálico*

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivo.

### Ancho constante

Se utiliza para listas de programas, así como dentro de los párrafos para referirse a elementos del programa como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

#### Ancho constante en negrita

Muestra comandos u otro texto que el usuario debe escribir literalmente.

#### Cursiva de ancho constante

Muestra texto que debe reemplazarse por valores proporcionados por el usuario o por valores determinados por el contexto.



Este icono significa un consejo, sugerencia o nota general.



Este icono indica una advertencia o precaución.

## Usar ejemplos de código

Este libro está aquí para ayudarlo a hacer su trabajo. En general, puede utilizar el código de este libro en sus programas y documentación. No es necesario que se comunique con nosotros para obtener permiso a menos que esté reproduciendo una parte significativa del código. Por ejemplo, escribir un programa que utiliza varios fragmentos de código de este libro no requiere permiso. Vender o distribuir un CD-ROM de ejemplos de libros de O'Reilly requiere permiso. Responder una pregunta citando este libro y citando un código de ejemplo no requiere permiso. La incorporación de una cantidad significativa de código de ejemplo de este libro en la documentación de su producto requiere permiso.

Agradecemos la atribución, pero no la exigimos. Una atribución generalmente incluye el título, el autor, el editor y el ISBN. Por ejemplo: " *Datos de primavera* por Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin y Michael Hunger (O'Reilly). Copyright 2013 Mark Pollack, Oliver Gierke, Thomas Risberg, Jonathan L. Brisbin y Michael Hunger, 978-1-449-32395-0 ".

Si cree que el uso que hace de los ejemplos de código está fuera del uso legítimo o del permiso otorgado anteriormente, no dude en contactarnos en [permissions@oreilly.com](mailto:permissions@oreilly.com) .

Los ejemplos de código se publican en [GitHub](#) .

## Libros en línea de Safari®



Safari BooksOnline ([www.safaribooksonline.com](http://www.safaribooksonline.com)) es una biblioteca digital bajo demanda que ofrece **contenido** en formato de libro y video de los principales autores del mundo en tecnología y negocios.

Los profesionales de la tecnología, los desarrolladores de software, los diseñadores web y los profesionales creativos y empresariales utilizan Safari Books Online como su recurso principal para la investigación, la resolución de problemas, el aprendizaje y la formación de certificación.

Safari Books Online ofrece una variedad de [mezclas de productos](#) y programas de precios para [organizaciones](#), [agencias gubernamentales](#) y [individuos](#). Los suscriptores tienen acceso a miles de libros, videos de capacitación y manuscritos prepublicados en una base de datos de editores como O'ReillyMedia, PrenticeHall 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 y decenas [más](#). Para obtener más información sobre Safari BooksOnline, visítenos [en línea](#).

## Cómo contactarnos

Dirija sus comentarios y preguntas sobre este libro al editor:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (en los Estados Unidos o Canadá)  
707-829-0515 (internacional o local)  
707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos erratas, ejemplos y cualquier información adicional. Puede acceder a esta página en <http://oreil.ly/spring-data-1e>.

Para comentar o hacer preguntas técnicas sobre este libro, envíe un correo electrónico a [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

Para obtener más información sobre nuestros libros, cursos, conferencias y noticias, visite nuestro sitio web en <http://www.oreilly.com>

Encuentranos en Facebook: <http://facebook.com/oreilly>

Síguenos en Twitter: <http://twitter.com/oreillymedia>

Míranos en YouTube: <http://www.youtube.com/oreillymedia>

## **Expresiones de gratitud**

Nos gustaría agradecer a Rod Johnson y Emil Eifrem por iniciar lo que se convertiría en el proyecto Spring Data.

Un gran agradecimiento a David Turanski por colaborar y ayudar con el capítulo de GemFire. Gracias a Richard McDougall por las estadísticas de macrodatos utilizadas en la introducción y a Costin Leau por su ayuda para escribir las aplicaciones de muestra de Hadoop.

También nos gustaría agradecer a O'ReillyMedia, especialmente a Meghan Blanchette por guiarnos a través del proyecto, a la editora de producción Kristen Borg y a la correctora de estilo Rachel Monaghan. Gracias a Greg Turnquist, Joris Kuipers, Johannes Hiemer, Joachim Arrasz, Stephan Hochdörfer, Mark Spritzler, JimWebber, Lasse Westh-Nielsen y todos los demás revisores técnicos por sus comentarios. Gracias a la comunidad que rodea al proyecto por enviar comentarios y problemas para que podamos mejorar constantemente. Por último, pero no menos importante, gracias a nuestros amigos y familiares por su paciencia, comprensión y apoyo.



## **Antecedentes**



# El proyecto Spring Data

El proyecto Spring Data se acuñó en Spring One 2010 y se originó a partir de una sesión de piratería informática de Rod Johnson (SpringSource) y Emil Eifrem (Neo Technologies) a principios de ese año. Intentaban integrar la base de datos de gráficos de Neo4j con Spring Framework y evaluaron diferentes enfoques. La sesión sentó las bases de lo que eventualmente se convertiría en la primera versión del módulo Neo4j de Spring Data, un nuevo proyecto SpringSource destinado a respaldar el creciente interés en los almacenes de datos NoSQL, una tendencia que continúa hasta el día de hoy.

Spring ha proporcionado un soporte sofisticado para las tecnologías tradicionales de acceso a datos desde el primer día. Simplificó significativamente la implementación de capas de acceso a datos, independientemente de si se utilizó JDBC, Hibernate, TopLink, JDO o iBatis como tecnología de persistencia. Este soporte consistió principalmente en la configuración simplificada de la infraestructura y la administración de recursos, así como en la traducción de excepciones a Spring's DataAccessExceptions. Este soporte ha madurado a lo largo de los años y las últimas versiones de Spring contenían actualizaciones decentes para esta capa de soporte.

El soporte de acceso a datos tradicional en Spring se ha dirigido solo a las bases de datos relacionales, ya que eran la herramienta predominante de elección cuando se trataba de la persistencia de datos. A medida que las tiendas NoSQL ingresan al escenario para proporcionar alternativas razonables en la caja de herramientas, hay espacio para completar en términos de soporte para desarrolladores. Más allá de eso, hay aún más oportunidades de mejora incluso para las tiendas relacionales tradicionales. Estas dos observaciones son los principales impulsores del proyecto Spring Data, que consta de módulos dedicados para tiendas NoSQL, así como módulos JPA y JDBC con soporte adicional para bases de datos relacionales.

## Acceso a datos NoSQL para desarrolladores de Spring

Aunque el término NoSQL se usa para referirse a un conjunto de almacenes de datos bastante jóvenes, todas las tiendas tienen características y casos de uso muy diferentes. Irónicamente, es la no característica (la falta de soporte para ejecutar consultas usando SQL) lo que realmente nombró a este grupo de bases de datos. Como estas tiendas tienen características bastante diferentes, sus controladores Java tienen

diferentes API para aprovechar las características y características especiales de las tiendas. Tratar de abstraerse de sus diferencias eliminaría los beneficios que ofrece cada almacén de datos NoSQL. Se debe elegir una base de datos de gráficos para almacenar datos altamente interconectados. Se debe utilizar una base de datos de documentos para estructuras de datos de tipo árbol y agregado. Se debe elegir un almacén de clave / valor si necesita patrones de acceso y funcionalidad similar a la de la caché.

Con JPA, el espacio Java EE (Enterprise Edition) ofrece una API de persistencia que podría haber sido candidata a implementaciones frontales de bases de datos NoSQL. Desafortunadamente, las dos primeras oraciones de la especificación ya indican que esto probablemente no esté funcionando:

Este documento es la especificación de la API de Java para la gestión de la persistencia y el mapeo de objetos / relacionales con Java EE y Java SE. El objetivo técnico de este trabajo es proporcionar una facilidad de mapeo relacional / objeto para el desarrollador de aplicaciones Java utilizando un modelo de dominio Java para administrar una base de datos relacional.

Este tema se refleja claramente en la especificación más adelante. Define conceptos y API que están profundamente conectados con el mundo de la persistencia relacional. Un @ Mesa la anotación no tendría mucho sentido para las bases de datos NoSQL, ni @ Columna o @ JoinCol umn. ¿Cómo se debe implementar la API de transacciones para tiendas como MongoDB, que esencialmente no proporcionan una semántica transaccional distribuida entre manipulaciones de múltiples documentos? Por lo tanto, implementar una capa JPA sobre una tienda NoSQL daría como resultado un perfil de la API en el mejor de los casos.

Por otro lado, todas las características especiales que proporcionan las tiendas NoSQL (funcionalidad geoespacial, operaciones de reducción de mapas, recorridos de gráficos) tendrían que implementarse de forma patentada de todos modos, ya que JPA simplemente no proporciona abstracciones para ellos. Por lo tanto, básicamente terminaríamos en el peor escenario de ambos mundos: las partes que se pueden implementar detrás de JPA más características patentadas adicionales para volver a habilitar características específicas de la tienda.

Este contexto descarta JPA como una API de abstracción potencial para estas tiendas. Aún así, nos gustaría ver la productividad del programador y la consistencia del modelo de programación conocida de varios proyectos del ecosistema Spring para simplificar el trabajo con las tiendas NoSQL. Esto llevó al equipo de Spring Data a declarar la siguiente declaración de misión:

Spring Data proporciona un modelo de programación familiar y consistente basado en Spring para NoSQL y tiendas relacionales al tiempo que conserva las características y capacidades específicas de la tienda.

Así que decidimos adoptar un enfoque ligeramente diferente. En lugar de intentar abstraer todas las tiendas detrás de una única API, el proyecto Spring Data proporciona un modelo de programación coherente en las diferentes implementaciones de tiendas utilizando patrones y abstracciones ya conocidos dentro de Spring Framework. Esto permite una experiencia uniforme cuando trabaja con diferentes tiendas.

## Temas generales

Un tema central del proyecto Spring Data disponible para todas las tiendas es el soporte para configurar recursos para acceder a las tiendas. Este soporte se implementa principalmente como espacio de nombres XML y clases de soporte para Spring JavaConfig y nos permite configurar fácilmente el acceso a una base de datos Mongo, una instancia incorporada de Neo4j y similares. Además, se proporciona integración con la funcionalidad principal de Spring como JMX, lo que significa que algunas tiendas expondrán estadísticas a través de su API nativa, que estará expuesta a JMX a través de Spring Data.

La mayoría de las API de NoSQL Java no brindan soporte para mapear objetos de dominio en las abstracciones de datos de las tiendas (documentos en MongoDB; nodos y relaciones para Neo4j). Por lo tanto, al trabajar con los controladores nativos de Java, normalmente tendría que escribir una cantidad significativa de código para copiar datos en los objetos de dominio de su aplicación al leer, y viceversa al escribir. Por lo tanto, una parte fundamental de los módulos de Spring Data es una API de mapeo y conversión que permite obtener metadatos sobre clases de dominio para ser persistentes y permitir la conversión real de objetos arbitrarios en tipos de datos específicos de la tienda.

Además de eso, encontraremos API obstinadas en forma de implementaciones de patrones de plantilla ya conocidas de Spring JdbcTemplate, JmsTemplate, etc. Por lo tanto, hay una RedisTemplate, una MongoTemplate, y así. Como probablemente ya sepa, estas plantillas ofrecen métodos auxiliares que nos permiten ejecutar operaciones comúnmente necesarias, como conservar un objeto con una sola declaración, mientras se encarga automáticamente de la gestión adecuada de recursos y la traducción de excepciones. Más allá de eso, exponen las API de devolución de llamada que le permiten acceder a las API nativas de la tienda y, al mismo tiempo, obtener las excepciones traducidas y los recursos administrados correctamente.

Estas características ya nos brindan una caja de herramientas para implementar una capa de acceso a datos como estamos acostumbrados a las bases de datos tradicionales. Los capítulos siguientes le guiarán a través de esta funcionalidad. Para facilitar aún más ese proceso, Spring Data proporciona una abstracción de repositorio además de la implementación de la plantilla que reducirá el esfuerzo de implementar objetos de acceso a datos a una definición de interfaz simple para los escenarios más comunes, como realizar operaciones CRUD estándar y ejecutar consultas en caso de que la tienda lo admita. Esta abstracción es en realidad la capa superior y combina las API de las diferentes tiendas tanto como sea razonablemente posible. Por lo tanto, las implementaciones específicas de la tienda comparten muchos puntos en común. Es por eso que encontrará un capítulo dedicado ([Capítulo 2](#)) presentándole el modelo básico de programación.

Ahora echemos un vistazo a nuestro código de muestra y el modelo de dominio que usaremos para demostrar las características de los módulos de la tienda en particular.

## El dominio

Para ilustrar cómo trabajar con los distintos módulos de Spring Data, usaremos un dominio de muestra del sector de comercio electrónico (ver [Figura 1-1](#)). Como los almacenes de datos NoSQL suelen tener un punto óptimo dedicado de funcionalidad y aplicabilidad, los capítulos individuales pueden modificar la implementación real del dominio o incluso implementarlo solo parcialmente. No se trata de sugerir que deba modelar el dominio de una determinada manera, sino de enfatizar qué tienda podría funcionar mejor para un escenario de aplicación determinado.

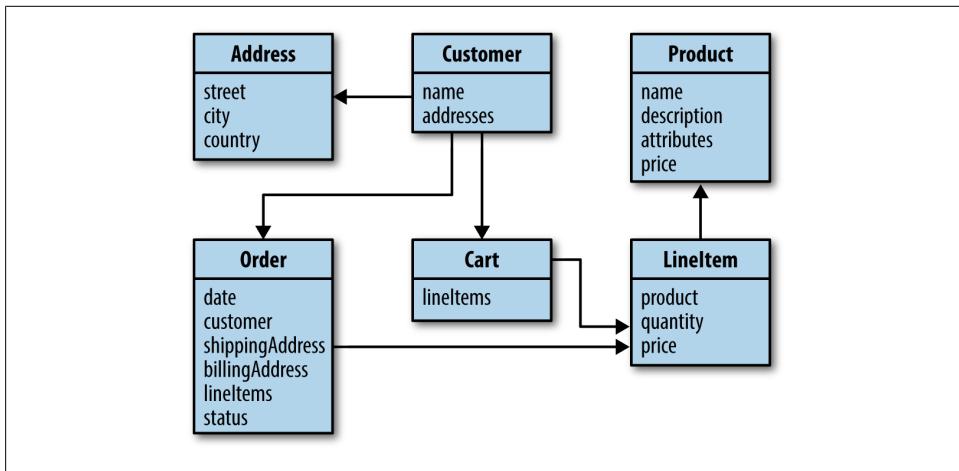


Figura 1-1. El modelo de dominio

En el núcleo de nuestro modelo, tenemos un cliente que tiene datos básicos como un nombre, un apellido, una dirección de correo electrónico y un conjunto de direcciones que, a su vez, contienen la calle, la ciudad y el país. También tenemos productos que consisten en un nombre, una descripción, un precio y atributos arbitrarios. Estas abstracciones forman la base de un CRM (gestión de relaciones con el cliente) rudimentario y un sistema de inventario. Además de eso, tenemos pedidos que puede realizar un cliente. Un pedido contiene el cliente que lo realizó, las direcciones de envío y facturación, la fecha en que se realizó el pedido, el estado del pedido y un conjunto de artículos de línea. Estos elementos de línea, a su vez, hacen referencia a un producto en particular, la cantidad de productos que se pedirán y el precio del producto.

## El código de muestra

El código de muestra de este libro se puede encontrar en [GitHub](#). Es un proyecto de Maven que contiene un módulo por capítulo. Requiere una instalación de Maven 3 en su máquina o un IDE capaz de importar proyectos de Maven como Spring Tool Suite (STS). Obtener el código es tan simple como clonar el repositorio:

```
$ cd ~/dev  
$ git clone https://github.com/SpringSource/spring-data-book.git Clonación en 'spring-data-book' ...  
  
remoto: Contando objetos: 253, hecho.  
remoto: Comprimir objetos: 100% (137/137), listo.  
Recepción de objetos: 100% (253/253), 139,99 KiB | 199 KiB / s, hecho. remoto: Total 253 (delta 91),  
reutilizado 219 (delta 57)  
Resolución de deltas: 100% (91/91), hecho. $ cd libro de  
datos de primavera
```

Ahora puede compilar el código ejecutando Maven desde la línea de comando de la siguiente manera:

```
paquete limpio $ mvn
```

Esto hará que Maven resuelva dependencias, compile y pruebe el código, ejecute pruebas y eventualmente empaquete los módulos.

## Importar el código fuente a su IDE

### STS / Eclipse

STS se envía con el complemento m2eclipse para trabajar fácilmente con proyectos de Maven directamente dentro de su IDE. Entonces, si ya lo tiene descargado e instalado (eche un vistazo a [Capítulo 3](#) para obtener más detalles), puede elegir la opción Importar del menú Archivo. Seleccione la opción Proyectos existentes de Maven en el cuadro de diálogo, que se muestra en [Figura 1-2](#).

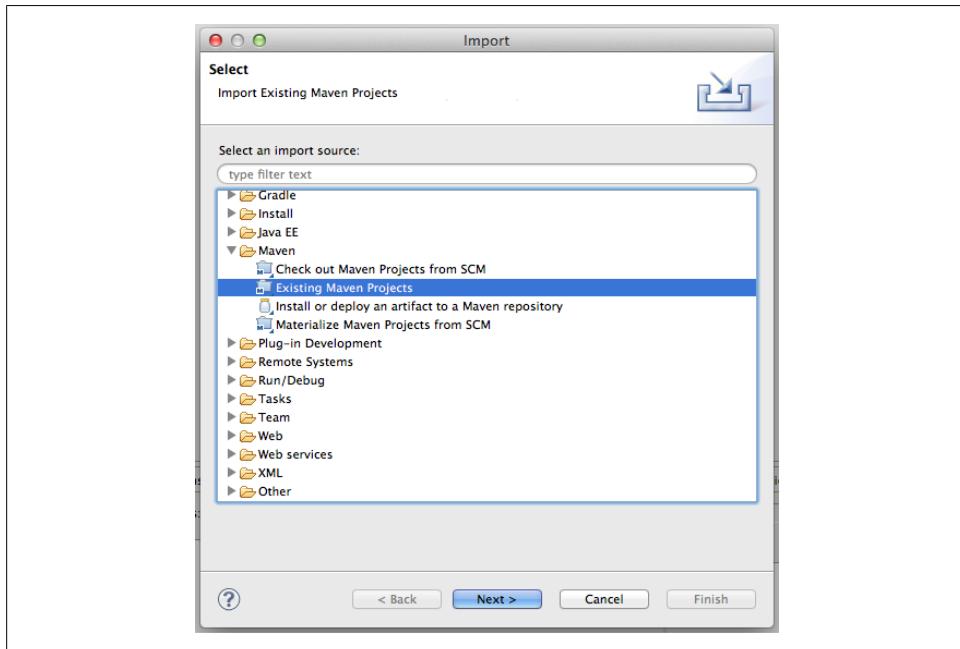
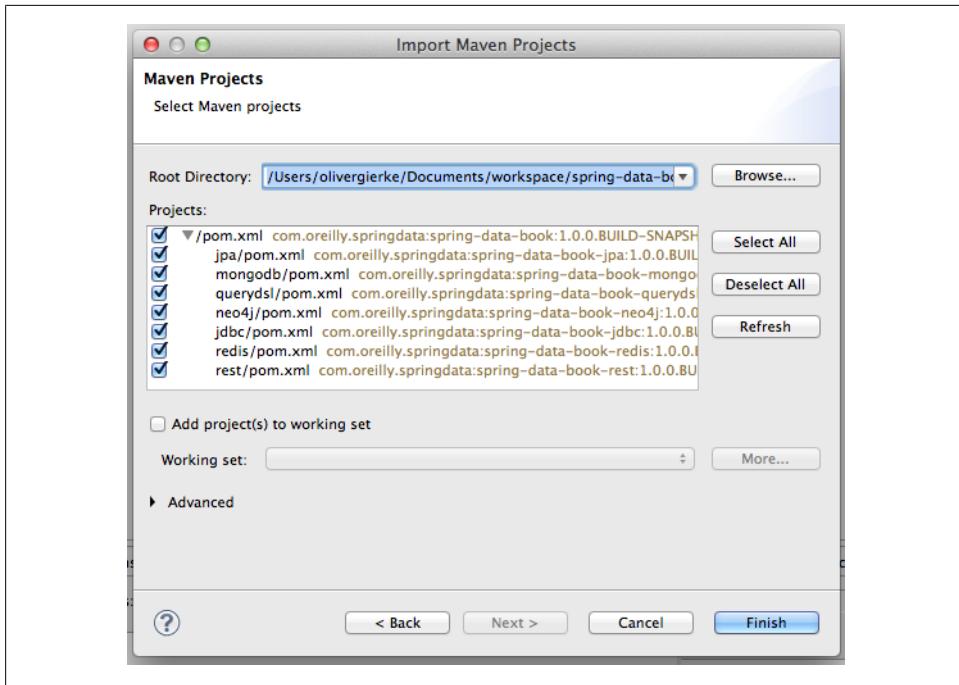


Figura 1-2. Importar proyectos de Maven a Eclipse (paso 1 de 2)

En la siguiente ventana, seleccione la carpeta en la que acaba de retirar el proyecto usando el botón Examinar. Una vez que lo haya hecho, el panel que se encuentra debajo debería llenarse con los módulos individuales de Maven enumerados y marcados ( [Figura 1-3](#) ). Continúe haciendo clic en Finalizar y STS importará los módulos Maven seleccionados a su espacio de trabajo. También resolverá las dependencias necesarias y la carpeta de origen de acuerdo con el `pom.xml` archivo en el directorio raíz del módulo.



*Figura 1-3. Importar proyectos de Maven a Eclipse (paso 2 de 2)*

Eventualmente debería terminar con un Explorador de paquetes o proyectos que se parezca a [Figura 1-4](#) . Los proyectos deben compilarse bien y no contener marcadores de error rojos.

Los proyectos que usan QueryDSL (ver [Capítulo 5](#) para obtener más detalles) aún puede llevar un marcador de error rojo. Esto se debe a que el complemento m2eclipse necesita información adicional sobre cuándo ejecutar los complementos de Maven relacionados con QueryDSL en el ciclo de vida de la compilación del IDE. La integración para eso se puede instalar desde el sitio de actualización de la extensión them2e-querydsl; encontrará la versión más reciente en el [página de inicio del proyecto](#) . Copie el enlace a la última versión que aparece allí (0.0.3, en el momento de escribir este artículo) y agréguelo a la lista de sitios de actualización disponibles, como se muestra en [Figura 1-5](#) . Instalar la característica expuesta a través de ese sitio de actualización, reiniciar Eclipse y posiblemente actualizar la configuración del proyecto Maven (haga clic con el botón derecho en el proyecto → Maven → Actualizar proyecto) debería permitirle terminar con todos los proyectos sin marcadores de error de Eclipse y construir bien.

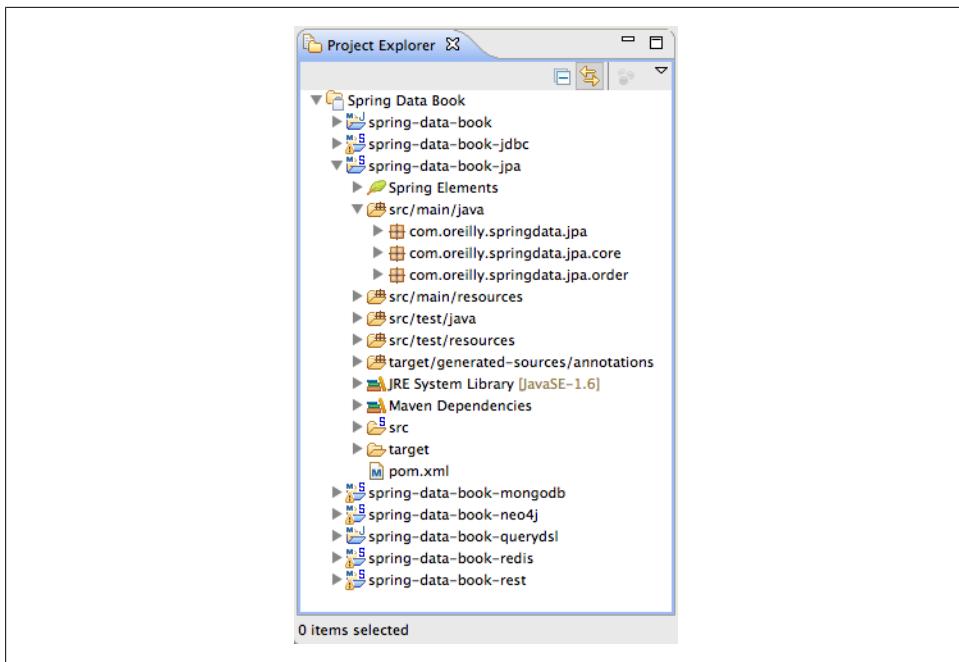


Figura 1-4. Explorador de proyectos de Eclipse con importación finalizada

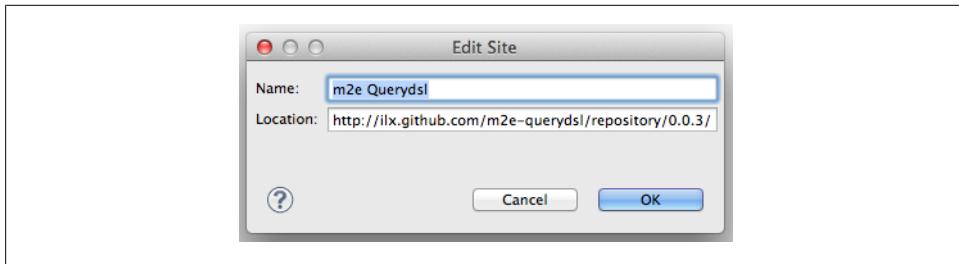


Figura 1-5. Agregar el sitio de actualización m2e-querydsl

### IntelliJ IDEA

IDEA puede abrir archivos de proyectos de Maven directamente sin necesidad de configuración adicional. Seleccione la entrada del menú Abrir proyecto para mostrar el cuadro de diálogo (consulte [Figura 1-6](#) ).

El IDE abre el proyecto y busca las dependencias necesarias. En el siguiente paso (mostrado en [Figura 1-7](#) ), detecta marcos usados (como Spring Framework, JPA, WebApp); utilice el enlace Configurar en la ventana emergente o el Registro de eventos para configurarlos.

Entonces, el proyecto está listo para ser utilizado. Verá la vista Proyecto y la vista Proyectos Maven, como se muestra en [Figura 1-8](#). Compile el proyecto como de costumbre.

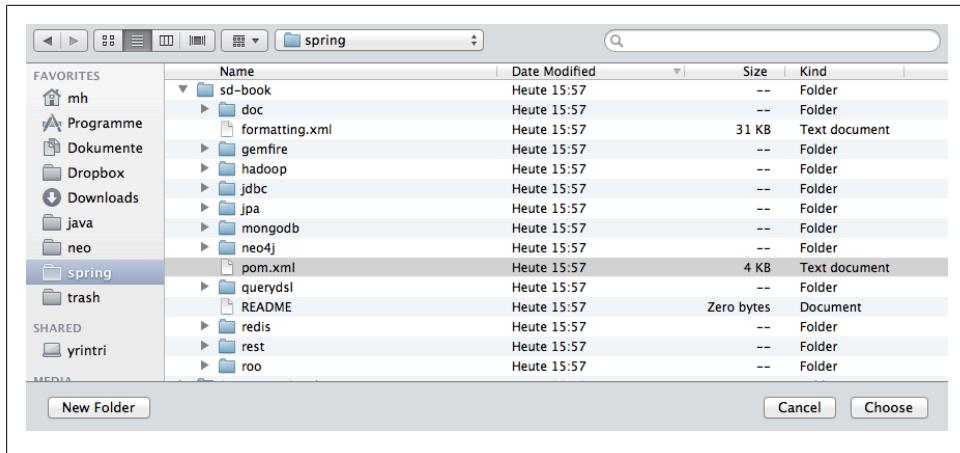


Figura 1-6. Importar proyectos de Maven a IDEA (paso 1 de 2)

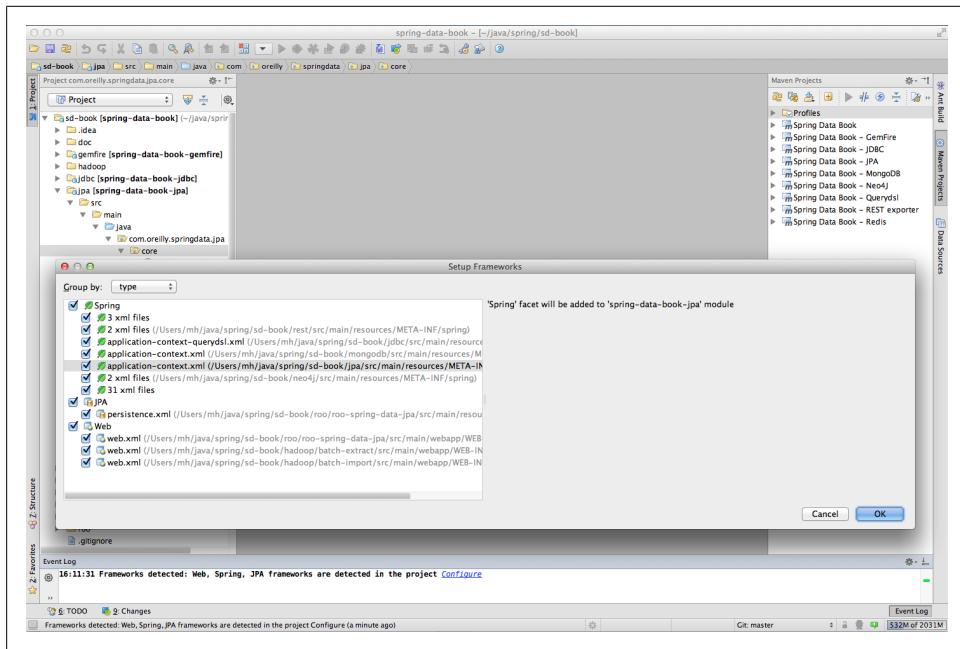


Figura 1-7. Importación de proyectos de Maven en IDEA (paso 2 de 2)

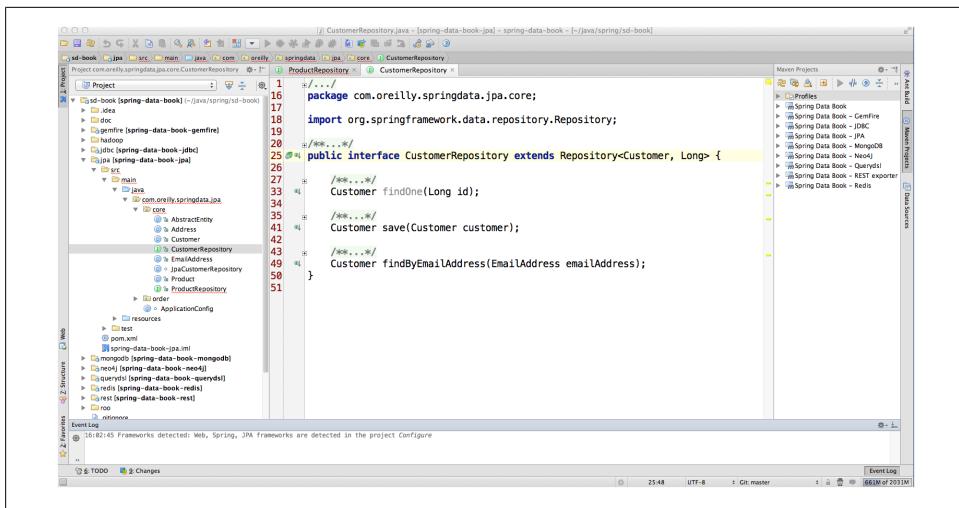


Figura 1-8. IDEA con el proyecto Spring Data Book abierto

A continuación, debe agregar compatibilidad con JPA en SpringData JPA module para permitir la finalización del método de búsqueda y la verificación de errores de los repositorios. Simplemente haga clic con el botón derecho en el módulo y elija Agregar marco. En el cuadro de diálogo resultante, marque el soporte de persistencia JavaEE y seleccione Hibernate como proveedor de persistencia ( Figura 1-9 ). Esto creará un `src/main/java/resources/META-INF/persistence.xml` archivo con solo una configuración de unidad de persistencia.

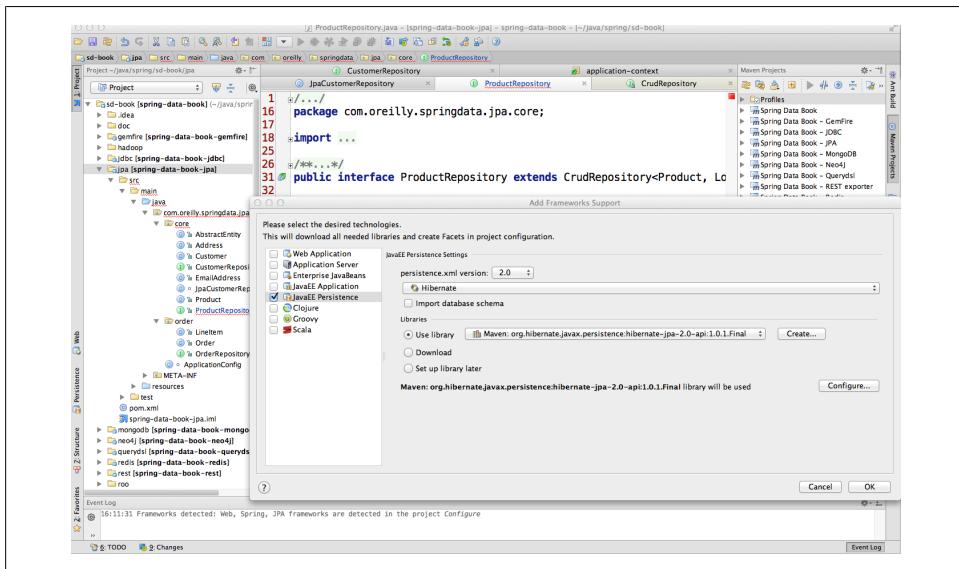


Figura 1-9. Habilite la compatibilidad con JPA para el módulo Spring Data JPA



# Repositorios: datos prácticos

## Capas de acceso

Implementar la capa de acceso a datos de una aplicación ha sido engorroso durante bastante tiempo. Se tuvo que escribir demasiado código repetitivo. Las clases de dominio eran anémicas y no estaban diseñadas de una manera real orientada a objetos o impulsada por dominios. El objetivo de la abstracción del repositorio de SpringData es reducir significativamente el esfuerzo requerido para implementar capas de acceso a datos para varios almacenes de persistencia. Las siguientes secciones presentarán los conceptos básicos y las interfaces de los repositorios de Spring Data. Usaremos el módulo Spring Data JPA como ejemplo y discutiremos los conceptos básicos de la abstracción del repositorio. Para otras tiendas, asegúrese de adaptar los ejemplos en consecuencia.

### Inicio rápido

Tomemos el Cliente clase de dominio de nuestro dominio que se conservará en un almacén arbitrario. La clase puede parecerse a [Ejemplo 2-1](#).

*Ejemplo 2-1. La clase de dominio del cliente*

```
clase pública Cliente {  
  
    privado Identificación larga ;  
    privado Nombre de la cadena ;  
    privado Apellido de cadena ;  
    privado EmailAddress emailAddress ;  
    privado Dirección Dirección ;  
  
    ...  
}
```

Un enfoque tradicional para una capa de acceso a datos ahora requeriría que implemente al menos una clase de repositorio que contenga los métodos CRUD (Crear, Leer, Actualizar y Eliminar) necesarios, así como métodos de consulta para acceder a subconjuntos de las entidades almacenadas mediante la aplicación de restricciones en ellos. El enfoque del repositorio de Spring Data le permite obtener

deshacerse de la mayor parte del código de implementación y, en su lugar, comenzar con una definición de interfaz sencilla para el repositorio de la entidad, como se muestra en [Ejemplo 2-2](#).

*Ejemplo 2-2. La definición de la interfaz CustomerRepository*

```
interfaz pública CustomerRepository extiende Repositorio < Cliente , Largo > {  
    ...  
}
```

Como puede ver, ampliamos Spring Data Repositorio interfaz, que es solo una interfaz de marcador genérica. Su principal responsabilidad es permitir que la infraestructura de Spring Data recoja todos los repositorios Spring Data definidos por el usuario. Más allá de eso, captura el tipo de clase de dominio administrada junto con el tipo de ID de la entidad, lo que será bastante útil en una etapa posterior. Para desencadenar el descubrimiento automático de las interfaces declaradas, usamos el < repositorios /> elemento del espacio de nombres XML específico de la tienda ([Ejemplo 2-3](#)) o el relacionado @ Habilitar ... Repositorios anotación en caso de que estemos usando JavaConfig ([Ejemplo 2-4](#)). En nuestro caso de muestra, usaremos JPA. Solo necesitamos configurar el elemento XML paquete base atributo con nuestro paquete raíz para que Spring Data lo escanee en busca de interfaces de repositorio. La anotación también puede obtener un paquete dedicado configurado para buscar interfaces. Sin ninguna configuración adicional dada, simplemente inspeccionará el paquete de la clase anotada.

*Ejemplo 2-3. Activando el soporte del repositorio de Spring Data usando XML*

```
<? xml version = "1.0" encoding = "UTF-8"?>  
<frijoles: frijoles 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: repositorios paquete-base = "com.acme. **. repositorio" />  
  
</beans>
```

*Ejemplo 2-4. Activando el soporte del repositorio de Spring Data usando Java Config*

```
@Configuración  
@EnableJpaRepositories  
clase ApplicationConfig {  
  
}
```

Tanto la configuración XML como JavaConfig necesitarán enriquecerse con declaraciones de beans de infraestructura específicas de la tienda, como un JPA. EntityManagerFactory, una Fuente de datos, y similares. Para otras tiendas, simplemente usamos las anotaciones o elementos del espacio de nombres correspondientes. El fragmento de configuración, que se muestra en [Ejemplo 2-5](#), ahora hará que se encuentren los repositorios de Spring Data, y se crearán Spring beans que en realidad

constan de proxies que implementarán la interfaz descubierta. Por lo tanto, un cliente ahora podría seguir adelante y obtener acceso al bean permitiendo que Spring simplemente lo conecte automáticamente.

*Ejemplo 2-5. Usando un repositorio de Spring Data de un cliente*

```
@Componente
clase pública MyRepositoryClient {

    final privada Repositorio CustomerRepository ;

    @Autowired
    público MyRepositoryClient ( Repositorio CustomerRepository ) {
        Afirmar . no nulo ( repositorio );
        esta . repositorio = repositorio ;
    }

    ...
}
```

Con nuestro CustomerRepository configurada la interfaz, estamos listos para sumergirnos y agregar algunos métodos de consulta fáciles de declarar. Un requisito típico podría ser recuperar un Cliente por su dirección de correo electrónico. Para hacerlo, agregamos el método de consulta apropiado ([Ejemplo 2-6](#)).

*Ejemplo 2-6. Declarar un método de consulta*

```
interfaz pública CustomerRepository extiende Repositorio < Cliente , Largo > {

    Cliente findByEmailAddress ( Dirección de correo electrónico );
}
```

El elemento de espacio de nombres ahora recogerá la interfaz en el momento de inicio del contenedor y activará la infraestructura de Spring Data para crear un bean Spring para él. La infraestructura inspeccionará los métodos declarados dentro de la interfaz e intentará determinar una consulta que se ejecutará en la invocación del método. Si no hace nada más que declarar el método, Spring Data derivará una consulta de su nombre. También hay otras opciones para la definición de consultas; puedes leer más sobre ellos en “[Definición de métodos de consulta](#)” en la [página 16](#).

En [Ejemplo 2-6](#), la consulta se puede derivar porque seguimos la convención de nomenclatura de las propiedades del objeto de dominio. El Correo electrónico parte del nombre del método de consulta en realidad se refiere al Cliente clase dirección de correo electrónico propiedad, y así Spring Data se auto-derivará matemáticamente seleccione C de Cliente c donde c.emailAddress =? 1 por el método declaración si estaba utilizando el módulo JPA. También verificará que tenga referencias de propiedad válidas dentro de la declaración de su método y hará que el contenedor no se inicie en el tiempo de arranque si encuentra algún error. Los clientes ahora pueden simplemente ejecutar el método, haciendo que los parámetros del método dados estén vinculados a la consulta derivada del nombre del método y la consulta que se ejecutará ([Ejemplo 2-7](#)).

*Ejemplo 2-7. Ejecutando un método de consulta*

```
@Componente
clase pública MyRepositoryClient {

    final privada Repositorio CustomerRepository ;

    ...

    público vacio someBusinessMethod ( Dirección de correo electrónico ) {

        Cliente cliente = repositorio . findByEmailAddress ( correo electrónico );
    }
}
```

## Definición de métodos de consulta

### Estrategias de búsqueda de consultas

La interfaz que acabamos de ver tenía un método de consulta simple declarado. La infraestructura inspeccionó y analizó la declaración del método, y finalmente se derivó una consulta específica de la tienda. Sin embargo, a medida que las consultas se vuelven más complejas, los nombres de los métodos se vuelven torpemente largos. Para consultas más complejas, las palabras clave admitidas por el analizador de métodos ni siquiera serían suficientes. Por lo tanto, los módulos de tienda individuales se envían con una @ Consulta anotación, demostrado en [Ejemplo 2-8](#), que toma una cadena de consulta en el lenguaje de consulta específico de la tienda y potencialmente permite más ajustes con respecto a la ejecución de la consulta.

*Ejemplo 2-8. Definición manual de consultas usando la anotación @Query*

```
interfaz pública CustomerRepository extiende Repositorio < Cliente , Largo > {

    @Consulta ( "seleccione c de Cliente c donde c.emailAddress =? 1" )
    Cliente findByEmailAddress ( Dirección de correo electrónico );
}
```

Aquí usamos JPA como ejemplo y definimos manualmente la consulta que se habría derivado de todos modos.

Las consultas incluso se pueden externalizar en un archivo de propiedades: \$ Tienda- named-queries.properties, situado en META-INF —Donde \$ Tienda es un marcador de posición para *jpa*, *mongo*, *neo4j*, etc. La clave tiene que seguir la convención de \$ domainType. \$ methodName. Por lo tanto, para respaldar nuestro método existente con una consulta con nombre externalizada, la clave debería ser

Customer.findByEmailAddress. Los @ Consulta la anotación no es necesaria si se utilizan consultas con nombre.

## Derivación de consultas

El mecanismo de derivación de consultas integrado en la infraestructura del repositorio de Spring Data, que se muestra en [Ejemplo 2-9](#), es útil para crear consultas restrictivas sobre entidades del repositorio. Despojaremos los prefijos findBy, readBy, y arreglárselas del método y comience a analizar el resto. En un nivel muy básico, puede definir condiciones en propiedades de entidad y concatenarlas con Y o O.

*Ejemplo 2-9. Derivación de consultas a partir de nombres de métodos*

```
interfaz pública CustomerRepository extiende Repositorio < Cliente , Largo > {  
  
    Lista < Cliente > findByEmailAndLastname ( Dirección de correo electrónico , Apellido de cadena );  
}
```

El resultado real de analizar ese método dependerá del almacén de datos que usemos. También hay algunas cosas generales a tener en cuenta. Las expresiones suelen ser recorridos de propiedades combinados con operadores que se pueden concatenar. Como puedes ver en [Ejemplo 2-9](#), puede combinar expresiones de propiedad con Y y O. Más allá de eso, también obtienes soporte para varios operadores como Entre, menos que, mayor que, y Me gusta para las expresiones de propiedad. Como los operadores admitidos pueden variar de un almacén de datos a otro, asegúrese de consultar el capítulo correspondiente de cada tienda.

## Expresiones de propiedad

Las expresiones de propiedad pueden referirse simplemente a una propiedad directa de la entidad administrada (como acaba de ver en [Ejemplo 2-9](#)). En el momento de la creación de la consulta, ya nos aseguramos de que la propiedad analizada sea una propiedad de la clase de dominio administrado. Sin embargo, también puede definir restricciones atravesando propiedades anidadas. Como se vio arriba, Cliente tengo Habla a es con Código postal s. En ese caso, un nombre de método de:

```
Lista < Cliente > findByAddressZipCode ( ZipCode zipCode );
```

creará la propiedad transversal x.address.zipCode. El algoritmo de resolución comienza con la interpretación de la parte completa (AddressZipCode) como una propiedad y verifica la clase de dominio para una propiedad con ese nombre (con la primera letra en minúscula). Si tiene éxito, simplemente lo usa. De lo contrario, comienza a dividir la fuente en las partes de la caja del camello del lado derecho en una cabeza y una cola y trata de encontrar la propiedad correspondiente (por ejemplo,

AddressZip y Código). Si encuentra una propiedad con esa cabeza, tomamos la cola y continuamos construyendo el árbol desde allí. Debido a que en nuestro caso la primera división no coincide, movemos el punto de división más hacia la izquierda (desde "Dirección Postal, Código" a "Dirección, código postal Código").

Aunque esto debería funcionar en la mayoría de los casos, puede haber situaciones en las que el algoritmo pueda seleccionar la propiedad incorrecta. Supongamos nuestro Cliente la clase tiene un addressZip propiedad también. Entonces nuestro algoritmo coincidiría en la primera división, esencialmente eligiendo la propiedad incorrecta, y finalmente fallaría (como el tipo de addressZip probablemente no tiene código

propiedad). Para resolver esta ambigüedad, puede usar un subrayado (\_) dentro del nombre de su método para definir manualmente los puntos transversales. Entonces, el nombre de nuestro método terminaría así:

```
Lista < Cliente > findByAddress_ZipCode ( ZipCode zipCode );
```

## Paginación y clasificación

Si la cantidad de resultados devueltos por una consulta aumenta significativamente, podría tener sentido acceder a los datos en fragmentos. Para lograr eso, Spring Data proporciona una API de paginación que se puede usar con los repositorios. La definición de qué fragmento de datos debe leerse está oculta detrás del Pageable interfaz junto con su implementación Búsqueda de PageRe. Los datos devueltos al acceder a él página por página se mantienen en un Página, que no solo contiene los datos en sí, sino también metainformación sobre si es la primera o la última página, cuántas páginas hay en total, etc. Para calcular estos metadatos tendremos que disparar una segunda consulta además de la inicial.

Podemos usar la funcionalidad de paginación con el repositorio simplemente agregando un Pageable como parámetro de método. A diferencia de los demás, esto no estará vinculado a la consulta, sino que se utilizará para restringir el conjunto de resultados que se devolverá. Una opción es tener un tipo de retorno de Página, lo que restringirá los resultados, pero requerirá otra consulta para calcular la metainformación (por ejemplo, el número total de elementos disponibles). Nuestra otra opción es usar

Lista, que evitará la consulta adicional pero no proporcionará los metadatos. Si no necesita la funcionalidad de paginación, solo ordenación simple, agregue un Ordenar parámetro a la firma del método (ver [Ejemplo 2-10](#) ).

*Ejemplo 2-10. Métodos de consulta que utilizan paginable y ordenar*

```
Página < Cliente > findByLastname ( Apellido de cadena , Paginable paginable );
```

```
Lista < Cliente > findByLastname ( Apellido de cadena , Ordenar ordenar );
```

```
Lista < Cliente > findByLastname ( Apellido de cadena , Paginable paginable );
```

El primer método le permite pasar un Pageable instancia al método de consulta para agregar dinámicamente la paginación a su consulta definida estéticamente. Las opciones de clasificación se pueden transferir al método mediante el Ordenar parámetro explícitamente, o incrustado en el Búsqueda de PageRe objeto de valor, como puede ver en [Ejemplo 2-11](#) .

*Ejemplo 2-11. Usar paginable y ordenar*

```
Paginable paginable = nuevo PageRequest ( 2 , 10 , Dirección . ASC , "apellido" , "nombre de pila" );
Página < Cliente > resultado = findByLastname ( "Matthews" , paginable );
```

```
Ordenar ordenar = nuevo Ordenar ( Dirección . DESC , "Matthews" );
Lista < Cliente > resultado = findByLastname ( "Matthews" , ordenar );
```

## Definición de repositorios

Hasta ahora, hemos visto interfaces de repositorio con métodos de consulta derivados del nombre del método o declarados manualmente, según los medios proporcionados por el módulo Spring Data para la tienda real. Para衍生 estas consultas, tuvimos que extender una interfaz de marcador específica de Spring Data: Repositorio. Aparte de las consultas, suele haber bastante funcionalidad que necesita tener en sus repositorios: la capacidad de almacenar objetos, eliminarlos, buscarlos por ID, devolver todas las entidades almacenadas o acceder a ellas página por página. La forma más fácil de exponer este tipo de funcionalidad a través de las interfaces del repositorio es utilizando una de las interfaces de repositorio más avanzadas que proporciona Spring Data:

### Repositorio

Una interfaz de marcador simple para permitir que la infraestructura de Spring Data seleccione los repositorios definidos por el usuario

### CrudRepository

Se extiende Repositorio y agrega métodos de persistencia básicos como guardar, buscar y eliminar entidades

### PagingAndSortingRepositories

Se extiende CrudRepository y agrega métodos para acceder a las entidades página por página y ordenarlas según los criterios dados

Suponga que queremos exponer operaciones CRUD típicas para el CustomerRepository. Todo lo que tenemos que hacer es cambiar su declaración como se muestra en [Ejemplo 2-12](#) .

#### Ejemplo 2-12. CustomerRepository exponiendo métodos CRUD

```
interfaz pública CustomerRepository extiende CrudRepository < Cliente , Largo > {  
    Lista < Cliente > findByEmailAndLastname ( Dirección de correo electrónico , Apellido de cadena );  
}
```

los CrudRepository la interfaz ahora se parece a [Ejemplo 2-13](#) . Contiene métodos para guardar una sola entidad, así como una Iterable de entidades, métodos de búsqueda para una sola entidad o todas las entidades, y Eliminar(...) métodos de diferentes sabores.

#### Ejemplo 2-13. CrudRepository

```
interfaz pública CrudRepository < T , CARNÉ DE IDENTIDAD extiende Serializable > extiende Repositorio < T , CARNÉ DE IDENTIDAD > {  
  
< S extiende T > salvar ( Entidad S ); < S extiende T > Iterable < S > salvar ( Iterable < S > entidades  
);  
  
T Encuentra uno ( Yo hice );  
Iterable < T > encontrar todos ();  
  
vacío Eliminar ( Yo hice );  
vacío Eliminar ( Entidad T );  
vacío eliminar todos ();  
}
```

Cada uno de los módulos de Spring Data que admiten el enfoque de repositorio se envía con una implementación de esta interfaz. Por lo tanto, la infraestructura activada por la declaración del elemento del espacio de nombres no solo arrancará el código apropiado para ejecutar los métodos de consulta, sino que también usará una instancia de la clase de implementación del repositorio genérico para respaldar los métodos declarados en CrudRepository y eventualmente delegar llamadas a guardar (...), buscar todos (), etc., a esa instancia. PagingAndSortingRepository ( [Ejemplo 2-14](#) ) ahora a su vez se extiende CrudRepository y agrega métodos para permitir la gestión de instancias de Pageable y Ordenar en lo genérico encuentra todos(...) métodos para acceder a las entidades página por página.

#### *Ejemplo 2-14. PagingAndSortingRepository*

```
interfaz pública PagingAndSortingRepository < T , CARNÉ DE IDENTIDAD extiende Serializable >
extiende CrudRepository < T , CARNÉ DE IDENTIDAD > {

    Iterable < T > encuentra todos ( Ordenar ordenar );

    Página < T > encuentra todos ( Paginable paginable );
}
```

Para llevar esa funcionalidad al CustomerRepository, simplemente extenderías PagingY OrdenarRepository en vez de CrudRepository.

### Interfaces de repositorio de ajuste fino

Como acabamos de ver, es muy fácil incorporar trozos de funcionalidad predefinida al extender la interfaz de repositorio de Spring Data apropiada. La decisión de implementar este nivel de granularidad fue en realidad impulsada por la compensación entre el número de interfaces (y por lo tanto la complejidad) que expondríamos en el caso de que tuviéramos interfaces de separación para todos los métodos de búsqueda, todos los métodos de guardado, etc. frente a la facilidad de uso para los desarrolladores.

Sin embargo, puede haber escenarios en los que le gustaría exponer solo los métodos de lectura (la R en CRUD) o simplemente evitar que los métodos de eliminación se expongan en las interfaces de su repositorio. Spring Data ahora le permite personalizar un repositorio base personalizado con los siguientes pasos:

1. Cree una interfaz que se extienda Repository o anotado con @ Repository Definición.
2. Agregue los métodos que desea exponer y asegúrese de que coincidan con las firmas de los métodos proporcionados por las interfaces del repositorio de la base de datos de Spring.
3. Utilice esta interfaz como interfaz base para las declaraciones de interfaz para sus entidades.

Para ilustrar esto, supongamos que nos gustaría exponer solo el encuentra todos(...) método tomando un Pageable así como los métodos de guardado. La interfaz del repositorio base se vería así

#### [Ejemplo 2-15](#) .

*Ejemplo 2-15. Interfaz de repositorio base personalizada*

```
@NoRepositoryBean  
interfaz pública BaseRepository < T , CARNÉ DE IDENTIDAD extiende Serializable > extiende Repositorio < T , CARNÉ DE IDENTIDAD > {  
  
    Iterable < T > encontrar todos ( Orden paginable );  
  
    < S extiende T > S guardar ( Entidad S ); < S extiende T > S guardar ( Iterable  
  
        < S > entidades );  
}
```

Tenga en cuenta que además anotamos la interfaz con @ NoRepositoryBean para asegurarse de que la infraestructura del repositorio de Spring Data no intente crear una instancia de bean para él. Dejando tu CustomerRepository extender esta interfaz ahora expondrá exactamente la API que definió.

Está perfectamente bien crear una variedad de interfaces base (por ejemplo, una ReadOnlyRepositorio o un SaveOnlyRepository) o incluso una jerarquía de ellos en función de las necesidades de su proyecto. Por lo general, recomendamos comenzar con métodos CRUD definidos localmente directamente en el repositorio concreto de una entidad y luego pasar a las interfaces del repositorio base proporcionadas por Spring Data o a las personalizadas si es necesario. De esa forma, mantendrá el número de artefactos creciendo naturalmente con la complejidad del proyecto.

## Implementación manual de métodos de repositorio

Hasta ahora hemos visto dos categorías de métodos en un repositorio: métodos CRUD y métodos de consulta. Ambos tipos son implementados por la infraestructura de Spring Data, ya sea mediante una implementación de respaldo o el motor de ejecución de consultas. Estos dos casos probablemente cubrirán una amplia gama de operaciones de acceso a datos a las que se enfrentará al crear aplicaciones. Sin embargo, habrá escenarios que requieran código implementado manualmente. Veamos cómo podemos lograrlo.

Comenzamos implementando solo la funcionalidad que realmente necesita implementarse manualmente y seguimos algunas convenciones de nomenclatura con la clase de implementación (como se muestra en [Ejemplo 2-16](#) ).

*Ejemplo 2-16. Implementar funcionalidad personalizada para un repositorio*

```
interfaz CustomerRepositoryCustom {  
  
    Cliente myCustomMethod ( ... );  
}  
  
clase CustomerRepositoryImpl implementos CustomerRepositoryCustom {  
  
    // Potencialmente cablear dependencias  
  
    público Cliente myCustomMethod ( ... ) {  
        // el código de implementación personalizado va aquí
```

```
}
```

Ni la interfaz ni la clase de implementación tienen que saber nada sobre Spring Data. Esto funciona prácticamente de la misma manera que implementaría código manualmente con Spring. La parte más interesante de este fragmento de código en términos de Spring Data es que el nombre de la clase de implementación sigue la convención de nomenclatura para agregar el sufijo de la interfaz del repositorio central (`CustomerRepository` en nuestro caso) nombre con `Impl`. También tenga en cuenta que mantuvimos tanto la interfaz como la clase de implementación como *paquete privado* para evitar que se acceda a ellos desde fuera del paquete.

El paso final es cambiar la declaración de nuestra interfaz de repositorio original para extender la recién introducida, como se muestra en [Ejemplo 2-17](#).

*Ejemplo 2-17. Incluyendo funcionalidad personalizada en CustomerRepository*

```
interfaz pública CustomerRepository extiende CrudRepository < Cliente , Largo >,  
CustomerRepositoryCustom { ... }
```

Ahora, básicamente, hemos extraído la API expuesta en `CustomerRepositoryCustom` en nuestro `CustomerRepository`, lo que lo convierte en el punto de acceso central de la API de acceso a datos para `Cliente`s. Por lo tanto, el código del cliente ahora puede llamar `CustomerRepository.myCustomMethod(...)`. Pero ¿Cómo se descubre realmente la clase de implementación y se lleva al proxy para que se ejecute finalmente? El proceso de arranque para un repositorio se ve esencialmente de la siguiente manera:

1. Se descubre la interfaz del repositorio (p. Ej., `CustomerRepository`).
2. Estamos tratando de buscar una definición de bean con el nombre del nombre de la interfaz en minúsculas con el sufijo `Impl` ( p.ej, `customerRepositoryImpl`). Si se encuentra uno, lo usaremos.
3. Si no es así, buscamos una clase con el nombre de nuestra interfaz de repositorio principal con el sufijo `Impl` ( p.ej, `CustomerRepositoryImpl`, que será recogido en nuestro caso). Si se encuentra uno, registramos esta clase como un bean Spring y lo usamos.
4. La implementación personalizada encontrada se conectará a la configuración del proxy para la interfaz descubierta y actuará como un objetivo potencial para la invocación del método.

Este mecanismo le permite implementar fácilmente un código personalizado para un repositorio dedicado. El sufijo utilizado para la búsqueda de implementación se puede personalizar en el elemento del espacio de nombres XML o en un atributo del repositorio que habilita la anotación (consulte los capítulos individuales de la tienda para obtener más detalles al respecto). Los [documentación de referencia](#) también contiene material sobre cómo implementar un comportamiento personalizado para aplicarlo a múltiples repositorios.

## Integración IDE

A partir de la versión 3.0, Spring Tool Suite (STS) proporciona integración con la abstracción del repositorio de Spring Data. El área principal de soporte proporcionada para Spring Data por STS es el mecanismo de derivación de consultas para los métodos de búsqueda. Lo primero que te ayuda a

valide sus métodos de consulta derivados directamente dentro del IDE para que no tenga que arrancar un ApplicationContext, pero puede detectar con entusiasmo los errores tipográficos que introduce en los nombres de sus métodos.



STS es una distribución especial de Eclipse equipada con un conjunto de complementos para facilitar la creación de aplicaciones Spring tanto como sea posible. La herramienta se puede descargar desde el [sitio web del proyecto](#) o instalado en una distribución Eclipse simple utilizando el sitio de actualización de STS (basado en [Eclipse 3.8 o 4.2](#)).

Como puedes ver en [Figura 2-1](#), el IDE detecta que Description no es válido, ya que no existe tal propiedad disponible en el Producto clase. Para descubrir estos errores tipográficos, analizará Producto clase de dominio (algo que el arranque de la infraestructura del repositorio de Spring Data haría de todos modos) para las propiedades y analizar el nombre del método en un árbol transversal de propiedades. Para evitar este tipo de errores tipográficos lo antes posible, el soporte Spring Data de STS ofrece finalización de código para nombres de propiedades, palabras clave de criterios y concatenadores como Y y O (ver [Figura 2-2](#) ).

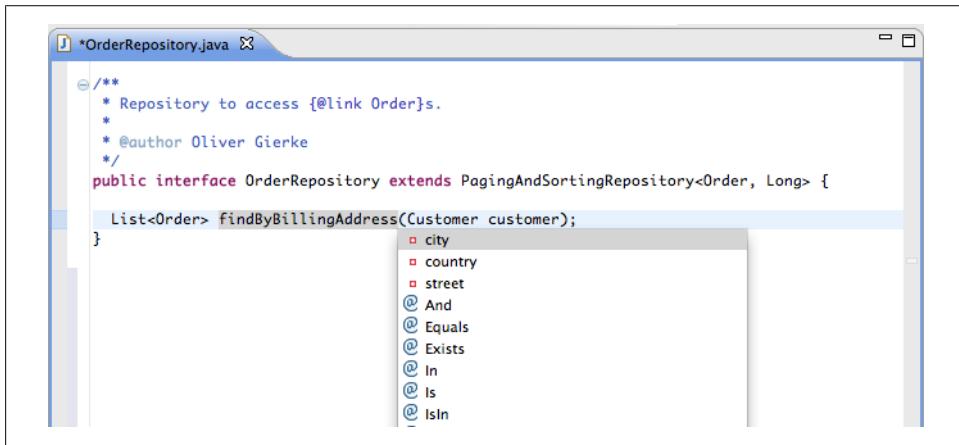
A screenshot of the STS IDE showing a Java code editor for 'ProductRepository.java'. The code defines a public interface 'ProductRepository' extending 'CrudRepository<Product, Long>'. It contains a method 'Page<Product> findByDescriptionContaining(String description, Pageable pageable);'. A red error marker is placed next to the word 'Description' in the method signature, with a tooltip message: 'Invalid derived query! No property description found for type com.oreilly.springdata.jpa.core.Product'. The IDE interface includes standard windows, tabs, and toolbars.

Figura 2-1. Validación del nombre del método de consulta derivado de Spring Data STS

A screenshot of the STS IDE showing a Java code editor for 'OrderRepository.java'. The code defines a public interface 'OrderRepository' extending 'PagingAndSortingRepository<Order, Long>'. It contains a method 'List<Order> findBy(Customer customer);'. A code completion dropdown menu is open over the word 'customer', listing options: 'billingAddress', 'customer', 'lineItems', and 'shippingAddress'. The IDE interface includes standard windows, tabs, and toolbars.

Figura 2-2. Propuestas de finalización de códigos de propiedad para métodos de consulta derivados

los Orden La clase tiene algunas propiedades a las que es posible que desee hacer referencia. Suponiendo que nos gustaría atravesar el Dirección de Envío propiedad, otro Cmd + Espacio (o Ctrl + Espacio en Windows) desencadenaría un recorrido de propiedad anidado que propone propiedades anidadas, así como palabras clave que coincidan con el tipo de propiedad recorrida hasta ahora ([Figura 2-3](#)). Así, Cuerda las propiedades también obtendrían Me gusta propuesto.



*Figura 2-3. Propuestas de propiedades y palabras clave anidadas*

Para poner un poco de guinda al pastel, Spring Data STS hará que los repositorios sean ciudadanos de primera clase de su navegador IDE, marcándolos con el conocido símbolo Spring bean. Más allá de eso, el nodo SpringElements en el navegador contendrá un nodo Spring Data Repositories dedicado para contener todos los repositorios que se encuentran en la configuración de su aplicación (ver [Figura 2-4](#) ).

Como puede ver, puede descubrir las interfaces del repositorio de un vistazo rápido y rastrear de qué elemento de configuración se originan realmente.

## IntelliJ IDEA

Finalmente, con el soporte JPA habilitado, IDEA ofrece la finalización del método de búsqueda de repositorios derivado de los nombres de propiedad y la palabra clave disponible, como se muestra en [Figura 2-5](#) .

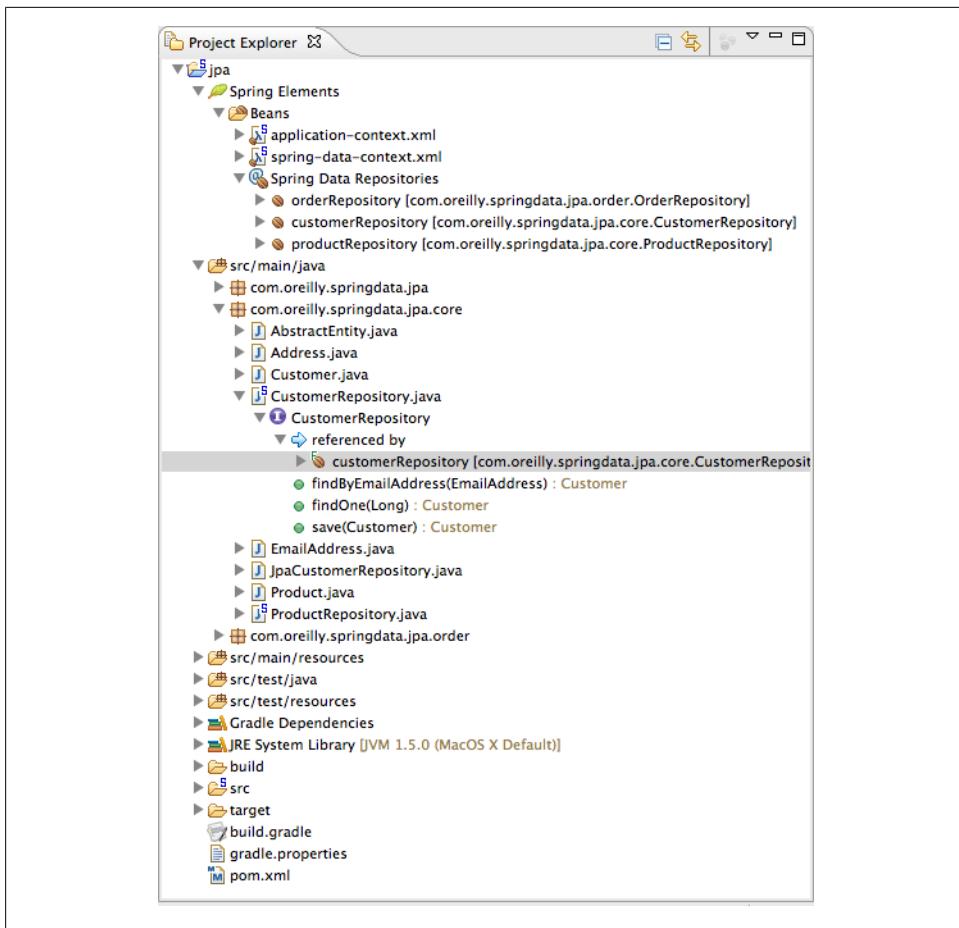


Figura 2-4. Explorador de proyectos Eclipse con soporte de Spring Data en STS

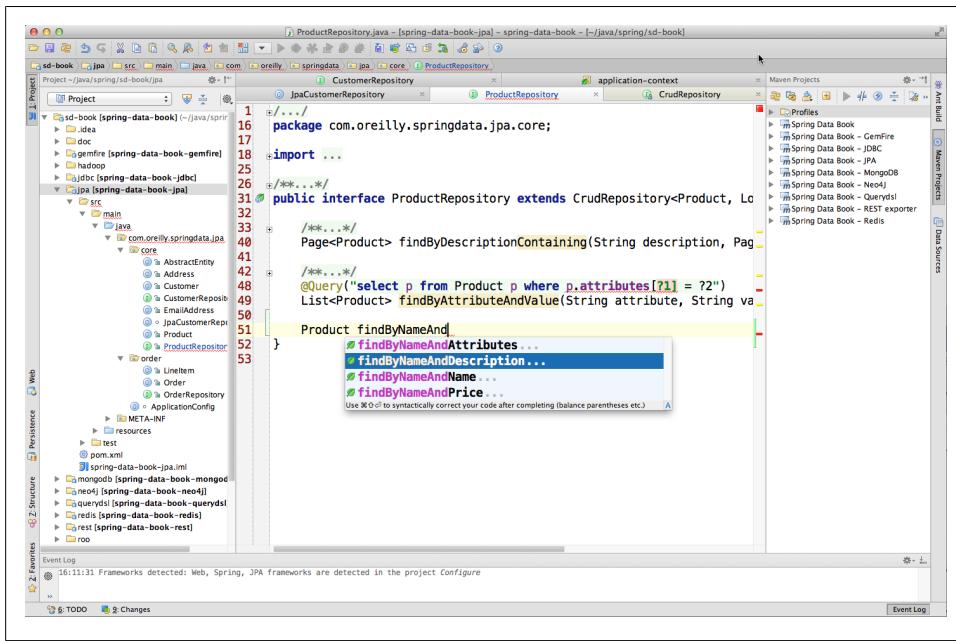


Figura 2-5. Finalización del método del buscador en el editor de IDEA

## Consulta de tipo seguro mediante Querydsl

La escritura de consultas para acceder a los datos generalmente se realiza usando Java Cuerda s. Los lenguajes de consulta elegidos han sido SQL para JDBC y HQL / JPQL para Hibernate / JPA. Definiendo las consultas en plano Cuerda s es potente pero bastante propenso a errores, ya que es muy fácil introducir errores tipográficos. Más allá de eso, hay poco acoplamiento con el origen o receptor de la consulta real, por lo que las referencias de columna (en el caso de JDBC) o las referencias de propiedad (en el contexto HQL / JPQL) se convierten en una carga de mantenimiento porque los cambios en la tabla o el modelo de objeto no se pueden reflejar en las consultas fácilmente.

los [Proyecto Querydsl](#) intenta abordar este problema proporcionando una API fluida para definir estas consultas. La API se deriva de la tabla real o del modelo de objetos, pero es muy independiente de la tienda y el modelo al mismo tiempo, por lo que le permite crear y utilizar la API de consulta para una variedad de tiendas. Actualmente es compatible con JPA, Hibernate, JDO, JDBC nativo, Lucene, Hibernate Search y MongoDB. Esta versatilidad es la razón principal por la que el proyecto Spring Data se integra con Querydsl, ya que Spring Data también se integra con una variedad de tiendas. Las siguientes secciones le presentarán el proyecto Querydsl y sus conceptos básicos. Entraremos en los detalles del soporte específico de la tienda en los capítulos relacionados con la tienda más adelante en este libro.

### Introducción a Querydsl

Cuando trabaje con Querydsl, normalmente empezará derivando un metamodelo de sus clases de dominio. Aunque la biblioteca también puede usar Cuerda literales, la creación del metamodelo desbloqueará todo el poder de Querydsl, especialmente su propiedad de seguridad de tipos y referencias de palabras clave. El mecanismo de derivación se basa en la herramienta de procesamiento de anotaciones (APT) de Java 6, que permite conectarse al compilador y procesar las fuentes o incluso las clases compiladas. Para obtener más información, lea sobre ese tema en ["Generación del metamodelo de consulta" en la página 30](#). Para comenzar, necesitamos definir una clase de dominio como la que se muestra en [Ejemplo 3-1](#). Modelamos nuestro Cliente con algunas propiedades primitivas y no primitivas.

*Ejemplo 3-1. La clase de dominio del cliente*

```
@QueryEntity  
clase pública Cliente extiende AbstractEntity {  
  
    privado Nombre de la cadena , apellido ;  
    privado EmailAddress emailAddress ;  
    privado Conjunto < Habla a > direcciones = nuevo HashSet < Habla a > ();  
  
    ...  
}
```

Tenga en cuenta que anotamos la clase con @ QueryEntity. Esta es la anotación predeterminada, a partir de la cual el procesador de anotaciones Querydsl genera la clase de consulta relacionada. Cuando utiliza la integración con una tienda en particular, el procesador APT podrá reconocer las anotaciones específicas de la tienda (por ejemplo, @ Entidad para JPA) y utilizarlos para derivar las clases de consulta. Como no vamos a trabajar con una tienda para esta introducción y, por lo tanto, no podemos usar una anotación de mapeo específica de la tienda, simplemente nos quedamos con @ QueryEntity.

La clase de consulta Querydsl generada ahora se verá así [Ejemplo 3-2](#) .

*Ejemplo 3-2. La clase de consulta generada por QueryDSL*

```
@Generado ( "com.mysema.query.codegen.EntitySerializer" )  
clase pública QCustomer extiende EntityPathBase < Cliente > {  
  
    final estática pública Cliente QCustomer = nuevo QCustomer ( "cliente" );  
    final pública QAbstractEntity _super = nuevo QAbstractEntity ( esta );  
  
    final pública NumberPath < Largo > carnet de identidad = _super . carnet de identidad ;  
    final pública Nombre de StringPath = createString ( "nombre de pila" );  
    final pública StringPath apellido = createString ( "apellido" );  
    final pública QEmailAddress emailAddress ;  
  
    final pública SetPath < Habla a , QAddress > direcciones =  
        esta . < Habla a , QAddress > createSet ( "direcciones" , Habla a . clase , QAddress . clase );  
  
    ...  
}
```

Puedes encontrar estas clases en el *destino / fuentes-generadas / consultas* carpeta del proyecto de muestra del módulo. La clase expone al público Camino propiedades y referencias a otras clases de consulta (por ejemplo, QEmailAddress). Esto permite que su IDE enumere las rutas disponibles para las que podría querer definir predicados durante la finalización del código. Ahora puedes usar estos

Camino expresiones para definir predicados reutilizables, como se muestra en [Ejemplo 3-3](#) .

*Ejemplo 3-3. Usando las clases de consulta para definir predicados*

```
Cliente QCustomer = QCustomer . cliente ;  
  
BooleanExpression idIsNull = cliente . carnet de identidad . es nulo ();  
BooleanExpression lastnameContainsFragment = cliente . apellido . contiene ( "fuerza muscular" );  
BooleanExpression firstnameLikeCart = cliente . nombre de pila . me gusta ( "Carro" );
```

```
Referencia de dirección de correo electrónico = nuevo Dirección de correo electrónico ( "dave@dmiband.com" );
BooleanExpression isDavesEmail = cliente . dirección de correo electrónico . eq ( referencia );
```

Asignamos la estática QCustomer.customer instancia a la cliente variable para poder hacer referencia de forma concisa a sus rutas de propiedad. Como puede ver, la definición de un predicado es clara, concisa y, lo más importante, segura para los tipos. Cambiar la clase de dominio haría que se regenerara la clase de metamodelo de consulta. Las referencias de propiedad que hayan quedado invalidadas por este cambio se convertirían en errores del compilador y, por lo tanto, nos darían pistas sobre los lugares que deben adaptarse. Los métodos disponibles en cada uno de los Camino tipos toman el tipo de Camino en cuenta (por ejemplo, el me gusta(...)) El método tiene sentido solo en

Cuerda propiedades y, por lo tanto, se proporciona solo en aquellos).

Debido a que las definiciones de predicado son tan concisas, se pueden usar fácilmente dentro de una declaración de método. Por otro lado, podemos definir predicados fácilmente de una manera reutilizable, construyendo predicados atómicos y combinándolos con otros más complejos usando operadores de concatenación como Y y O (ver [Ejemplo 3-4](#) ).

#### *Ejemplo 3-4. Concatenando predicados atómicos*

```
Cliente QCustomer = QCustomer . cliente ;

BooleanExpression idIsNull = cliente . carné de identidad . es nulo ();

Referencia de dirección de correo electrónico = nuevo Dirección de correo electrónico ( "dave@dmiband.com" );
BooleanExpression isDavesEmail = cliente . dirección de correo electrónico . eq ( referencia );

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

Podemos usar nuestros predicados recién escritos para definir una consulta para una tienda en particular o colecciones simples. Como el soporte para la ejecución de consultas específicas de la tienda se logra principalmente a través de la abstracción del repositorio de Spring Data, eche un vistazo a ["Integración con los repositorios de datos de Spring"](#) en la página 32 . Usaremos la función para consultar colecciones como ejemplo ahora para simplificar las cosas. Primero, configuraremos una variedad de Producto s tener algo que podamos filtrar, como se muestra en [Ejemplo 3-5](#) .

#### *Ejemplo 3-5. Configurar productos*

```
MacBook del producto = nuevo Producto ( "Macbook Pro" , "Portátil Apple" );
IPad del producto = nuevo Producto ( "IPad" , "Tableta de Apple" );
IPod del producto = nuevo Producto ( "IPod" , "Reproductor MP3 de Apple" );
Tocadiscos de productos = nuevo Producto ( "Placa giratoria" , "Reproductor de vinilo" );

Lista < Producto > productos = Matrices . asList ( macbook , iPad , iPod , placa giratoria );
```

A continuación, podemos usar la API de Querydsl para configurar realmente una consulta contra la colección, que es una especie de filtro ([Ejemplo 3-6](#) ).

#### *Ejemplo 3-6. Filtrado de productos mediante predicados Querydsl*

```
QProducto $ = QProduct . producto ;
Lista < Producto > resultado = desde ( PS , productos ) . dónde ( PS . descripción . contiene ( "Manzana" )) . lista ( PS );
```

```
afirmar que ( resultado , hasSize ( 3 ));  
afirmar que ( resultado , hasItems ( macbook , iPad , iPod ));
```

Estamos configurando un Querydsl Consulta utilizando la `desde(...)` método, que es un método estático en el MiniAPI clase de la `querydsl-colecciones` módulo. Le entregamos una instancia de la clase de consulta para Producto así como la colección fuente. Ahora podemos usar el `dónde(...)` método para aplicar predicados a la lista fuente y ejecutar la consulta usando uno de los `lista(...)` métodos ([Ejemplo 3-7](#)). En nuestro caso, simplemente nos gustaría recuperar el Producto instancias que coinciden con el predicado definido. Entrega de \$. descripción en el `lista(...)`. El método nos permitiría proyectar el resultado en la descripción del producto y así obtener una colección de Cuerda s.

#### *Ejemplo 3-7. Filtrar productos utilizando predicados Querydsl (proyectar)*

```
QProducto $ = QProducto . producto ;  
BooleanExpression descripciónContainsApple = PS . descripción . contiene ( "Manzana" );  
Lista < Cuerda > resultado = desde ( PS , productos ) . dónde ( descripciónContainsApple ) . lista ( PS . nombre );  
  
afirmar que ( resultado , hasSize ( 3 ));  
afirmar que ( resultado , hasItems ( "Macbook Pro" , "iPad" , "iPod" ));
```

Como hemos descubierto, Querydsl nos permite definir predicados de entidad de una manera concisa y sencilla. Estos se pueden generar a partir de la información de mapeo para una variedad de tiendas, así como para clases simples de Java. La API de Querydsl y su soporte para varias tiendas nos permite generar predicados para definir consultas. Las colecciones simples de Java se pueden filtrar con la misma API.

## Generando el metamodelo de consulta

Como acabamos de ver, los artefactos centrales con Querydsl son las clases de metamodelo de consulta. Estas clases se generan a través del [Kit de herramientas de procesamiento de anotaciones](#), parte de javac Compilador de Java en Java 6. La APT proporciona una API para inspeccionar mediante programación el código fuente de Java existente en busca de ciertas anotaciones y luego llamar a funciones que a su vez generan código Java. Querydsl utiliza este mecanismo para proporcionar clases especiales de implementación del procesador APT que inspeccionan las anotaciones. [Ejemplo 3-1](#) usó anotaciones específicas de Querydsl como `@QueryEntity` y `@QueryEmbeddable`. Si ya tenemos clases de dominio asignadas a una tienda compatible con Querydsl, la generación de las clases de metamodelo no requerirá ningún esfuerzo adicional. El punto de integración principal aquí es el procesador de anotaciones que se entrega a la APT de Querydsl. Los procesadores generalmente se ejecutan como un paso de compilación.

## Integración del sistema de construcción

Para integrarse con Maven, Querydsl proporciona `maven-apt-plugin`, con el que puede configurar la clase de procesador real que se utilizará. En [Ejemplo 3-8](#), atamos el proceso meta al generar-fuentes fase, que hará que la clase de procesador configurada