against a regular expression and rejects it right away if it doesn't match the expression. This way, clients can be sure they're dealing with a proper email address if they get hold of an `EmailAddress` instance.

Example 6-16. The EmailAddress domain class

```
public class EmailAddress {

  private static final String EMAIL_REGEX = …;
  private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);

  @Field("email")
  private final String value;

  public EmailAddress(String emailAddress) {
    Assert.isTrue(isValid(emailAddress), "Invalid email address!");
    this.value = emailAddress;
  }

  public static boolean isValid(String source) {
    return PATTERN.matcher(source).matches();
  }
}
```

The `value` property is annotated with `@Field`, which allows for customizing the way a property is mapped to a field in a `DBObject`. In our case, we map the rather generic `value` to a more specific `email`. While we could have simply named the property `email` in the first place in our situation, this feature comes in handy in two major scenarios. First, say you want to map classes onto existing documents that might have chosen field keys that you don't want to let leak into your domain objects. `@Field` generally allows decoupling between field keys and property names. Second, in contrast to the relational model, field keys are repeated for every document, so they can make up a large part of the document data, especially if the values you store are small. So you could reduce the space required for keys by defining rather short ones to be used, with the trade-off of slightly reduced readability of the actual JSON representation.

Now that we've set the stage with our basic domain concept implementations, let's have a look at the classes that actually will make up our documents.

Customers

The first thing you'll probably notice about the `Customer` domain class, shown in Example 6-17, is the `@Document` annotation. It is actually an optional annotation to some degree. The mapping subsystem would still be able to convert the class into a `DBObject` if the annotation were missing. So why do we use it here? First, we can configure the mapping infrastructure to scan for domain classes to be persisted. This will pick up only classes annotated with `@Document`. Whenever an object of a type currently unknown to the mapping subsystem is handed to it, the subsystem automatically and immediately inspects the class for mapping information, slightly decreasing the per-

formance of that very first conversion operation. The second reason to use `@Document` is the ability to customize the MongoDB collection in which a domain object is stored. If the annotation is not present at all or the collection attribute is not configured, the collection name will be the simple class name with the first letter lowercased. So, for example, a `Customer` would go into the `customer` collection.

The code might look slightly different in the sample project, because we're going to tweak the model slightly later to improve the mapping. We'd like to keep it simple at this point to ease your introduction, so we will concentrate on general mapping aspects here.

Example 6-17. The Customer domain class

```
@Document
public class Customer extends AbstractDocument {

  private String firstname, lastname;

  @Field("email")
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  public Customer(String firstname, String lastname) {

    Assert.hasText(firstname);
    Assert.hasText(lastname);

    this.firstname = firstname;
    this.lastname = lastname;
  }

  // additional methods and accessors
}
```

The `Customer` class contains two primitive properties to capture first name and last name as well as a property of our `EmailAddress` domain class and a `Set` of `Addresses`. The `emailAddress` property is annotated with `@Field`, which (as noted previously) allows us to customize the key to be used in the MongoDB document.

Note that we don't actually need any annotations to configure the relationship between the `Customer` and `EmailAddress` and the `Addresses`. This is mostly driven from the fact that MongoDB documents can contain complex values (i.e., nested documents). This has quite severe implications for the class design and the persistence of the objects. From a design point of view, the `Customer` becomes an *aggregate root,* in the domain-driven design terminology. `Addresses` and `EmailAddresses` are never accessed individually but rather through a `Customer` instance. We essentially model a tree structure here that maps nicely onto MongoDB's document model. This results in the object-to-document mapping being much less complicated than in an object-relational scenario. From a persistence point of view, storing the entire `Customer` alongside its `Addresses`

and `EmailAddress`es becomes a single—and thus atomic—operation. In a relational world, persisting this object would require an insert for each `Address` plus one for the `Customer` itself (assuming we'd inline the `EmailAddress` into a column of the table the `Customer` ends up in). As the rows in the table are only loosely coupled to each other, we have to ensure the consistency of the insert by using a transaction mechanism. Beyond that, the insert operations have to be ordered correctly to satisfy the foreign key relationships.

However, the document model not only has implications on the writing side of persistence operations, but also on the reading side, which usually makes up even more of the access operations for data. Because the document is a self-contained entity in a collection, accessing it does not require reaching out into other collections, documents or the like. Speaking in relational terms, a document is actually a set of prejoined data. Especially if applications access data of a particular granularity (which is what is usually driving the class design to some degree), it hardly makes sense to tear apart the data on writes and rejoin it on each and every read. A complete customer document would look something like Example 6-18.

Example 6-18. Customer document

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : { email : "dave@dmband.com" },
  addresses : [ { street : "Broadway",
                  city : "New York",
                  country : "United States" } ] }
```

Note that modeling an email address as a value object requires it to be serialized as a nested object, which essentially duplicates the key and makes the document more complex than necessary. We'll leave it as is for now, but we'll see how to improve it in "Customizing Conversion" on page 91.

Products

The `Product` domain class (Example 6-19) again doesn't contain any huge surprises. The most interesting part probably is that `Map`s can be stored natively—once again due to the nature of the documents. The attributes will be just added as a nested document with the `Map` entries being translated into document fields. Note that currently, only `String`s can be used as `Map` keys.

Example 6-19. The Product domain class

```
@Document
public class Product extends AbstractDocument {

  private String name, description;
  private BigDecimal price;
  private Map<String, String> attributes = new HashMap<String, String>();
```

```
  // … additional methods and accessors
}
```

Orders and line items

Moving to the order subsystem of our application, we should look first at the `LineItem` class, shown in Example 6-20.

Example 6-20. The LineItem domain class

```
public class LineItem extends AbstractDocument {

  @DBRef
  private Product product;
  private BigDecimal price;
  private int amount;

  // … additional methods and accessors
}
```

First we see two basic properties, `price` and `amount`, declared without further mapping annotations because they translate into document fields natively. The `product` property, in contrast, is annotated with `@DBRef`. This will cause the `Product` object inside the `LineItem` to not be embedded. Instead, there will be a pointer to a document in the collection that stores `Product`s. This is very close to a foreign key in the world of relational databases.

Note that when we're storing a `LineItem`, the `Product` instance referenced has to be saved already—so currently, there's no cascading of save operations available. When we're reading `LineItem`s from the store, the reference to the `Product` will be resolved eagerly, causing the referenced document to be read and converted into a `Product` instance.

To round things off, the final bit we should have a look at is the `Order` domain class (Example 6-21).

Example 6-21. The Order domain class

```
@Document
public class Order extends AbstractDocument {

  @DBRef
  private Customer customer;
  private Address billingAddress;
  private Address shippingAddress;
  private Set<LineItem> lineItems = new HashSet<LineItem>();

  // - additional methods and parameters
}
```

Here we essentially find a combination of mappings we have seen so far. The class is annotated with `@Document` so it can be discovered and inspected for mapping informa-

tion during application context startup. The `Customer` is referenced using an `@DBRef`, as we'd rather point to one than embedding it into the document. The `Address` properties and the `LineItem`s are embedded as is.

Setting Up the Mapping Infrastructure

As we've seen how the domain class is persisted, now let's have a look at how we actually set up the mapping infrastructure to work for us. In most cases this is pretty simple, and some of the components that use the infrastructure (and which we'll introduce later) will fall back to reasonable default setups that generally enable the mapping subsystem to work as just described. However, if you'd like to customize the setup, you'll need to tweak the configuration slightly. The two core abstractions that come into play here are the `MongoMappingContext` and `MappingMongoConverter`. The former is actually responsible for building up the domain class metamodel to avoid reflection lookups (e.g., to detect the `id` property or determine the field key on each and every persistence operation). The latter is actually performing the conversion using the mapping information provided by the `MappingContext`. You can simply use these two abstractions together to trigger object-to-DBObject-and-back conversion programmatically (see Example 6-22).

Example 6-22. Using the mapping subsystem programmatically

```
MongoMappingContext context = new MongoMappingContext();
MongoDbFactory dbFactory = new SimpleMongoDbFactory(new Mongo(), "database");
MappingMongoConverter converter = new MongoMappingContext(dbFactory, context);

Customer customer = new Customer("Dave", "Matthews");
customer.setEmailAddress(new EmailAddress("dave@dmband.com"));
customer.add(new Address("Broadway", "New York", "United States"));

DBObject sink = new BasicDBObject();
converter.write(customer, sink);

System.out.println(sink.toString());
```

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : { email : "dave@dmband.com" },
  addresses : [ { street : "Broadway",
                  city : "New York",
                  country : "United States" } ] }
```

We set up instances of a `MongoMappingContext` as well as a `SimpleMongoDbFactory`. The latter is necessary to potentially load `@DBRef` annotated documents eagerly. This is not needed in our case, but we still have to set up the `MongoMappingConverter` instance correctly. We then set up a `Customer` instance as well as a `BasicDBObject` and invoke the converter to do its work. After that, the `DBObject` is populated with the data as expected.

Using the Spring namespace

The Spring namespace that ships with Spring Data MongoDB contains a `<mongo:mapping-converter />` element that basically sets up an instance of `MappingMongoConverter` as we've seen before. It will create a `MongoMappingContext` internally and expect a Spring bean named `mongoDbFactory` in the `ApplicationContext`. We can tweak this by using the `db-factory-ref` attribute of the namespace element. See Example 6-23.

Example 6-23. Setting up a MappingMongoConverter in XML

```
<mongo:mapping-converter id="mongoConverter"
                         base-package="com.oreilly.springdata.mongodb" />
```

This configuration snippet configures the `MappingMongoConverter` to be available under the `id` `mongoConverter` in the Spring application context. We point the `base-package` attribute to our project's base package to pick up domain classes and build the persistence metadata at application context startup.

In Spring JavaConfig

To ease the configuration when we're working with Spring JavaConfig classes, Spring Data MongoDB ships with a configuration class that declares the necessary infrastructure components in a default setup and provides callback methods to allow us to tweak them as necessary. To mimic the setup just shown, our configuration class would have to look like Example 6-24.

Example 6-24. Basic MongoDB setup with JavaConfig

```
@Configuration
class ApplicationConfig extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "e-store";
  }

  @Override
  public Mongo mongo() throws Exception {

    Mongo mongo = new Mongo();
    mongo.setWriteConcern(WriteConcern.SAFE);
    return mongo;
  }

  @Override
  protected String getMappingBasePackage() {
    return "com.oreilly.springdata.mongodb"
  }
}
```

The first two methods are required to be implemented by the superclass because it sets up a `SimpleMongoDbFactory` to access a MongoDB already. Beyond these necessary

implementations, we override the `getMappingBasePackage()` method to indicate that the mapping subsystem will inspect this package, and all below it, for the classes annotated with `@Document`. This is not strictly necessary, as the mapping infrastructure will scan the package of the configuration class by default. We just list it here to demonstrate how it could be reconfigured.

Indexing

MongoDB supports indexes just as a relational store does. You can configure the index programatically or by using a mapping annotation. Because we're usually going to retrieve `Customer`s by their email addresses, we'd like to index on those. Thus, we add the `@Index` annotation to the `emailAddress` property of the `Customer` class, as shown in Example 6-25.

Example 6-25. Configuring an index on the emailAddress property of Customer

```
@Document
public class Customer extends AbstractDocument {

  @Index(unique = true)
  private EmailAddress emailAddress;

  …
}
```

We'd like to prevent duplicate email addresses in the system, so we set the `unique` flag to `true`. This will cause MongoDB to prevent `Customer`s from being created or updated with the same email address as another `Customer`. We can define indexes including multiple properties by using the `@CompoundIndex` annotation on the domain class.

Index metadata will be discovered when the class is discovered by the `MappingContext`. As the information is stored alongside the collection, to which the class gets persisted, it will get lost if you drop the collection. To avoid that, remove all documents from the collection.

You can find an example use case of a domain object being rejected in the `CustomerRepositoryIntegrationTests` class of the sample application. Note that we expect a `DuplicateKeyException` to be thrown, as we persist a second customer with the email address obtained from an already existing one.

Customizing Conversion

The mapping subsystem provides a generic way to convert your Java objects into MongoDB `DBObject`s and vice versa. However, you might want to manually implement a conversion of a given type. For example, you've seen previously that the introduction of a value object to capture email addresses resulted in a nested document that you

might want to avoid to keep the document structure simple, especially since we can simply inline the `EmailAddress` value into the customer object. To recap the scenario, Example 6-26 shows where we'd like to start.

Example 6-26. The Customer class and its DBObject representation

```
@Document
public class Customer extends AbstractDocument {

  private String firstname, lastname;

  @Field("email")
  private EmailAddress emailAddress;

  …
}
```

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : { email : "dave@dmband.com" }, … }
```

What we would actually like to end up with is a simpler document looking something like Example 6-27.

Example 6-27. The intended document structure of a Customer

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : "dave@dmband.com", … }
```

Implementing custom converters

The mapping subsystem allows you to manually implement the object-to-document-and-back conversion yourself by leveraging the Spring conversion service's `Converter` abstraction. Since we'd like to turn the complex object into a plain `String`, we essentially need to implement a writing `Converter<EmailAddress, String>` as well as one to construct `EmailAddress` objects from `String`s (i.e., a `Converter<String, EmailAddress>`), as shown in Example 6-28.

Example 6-28. Custom Converter implementations for EmailAddress

```
@Component
class EmailAddressToStringConverter implements Converter<EmailAddress, String> {

  public String convert(EmailAddress source) {
    return source == null ? null : source.value;
  }
}

@Component
class StringToEmailAddressConverter implements Converter<String, EmailAddress> {

  public EmailAddress convert(String source) {
    return StringUtils.hasText(source) ? new EmailAddress(source) : null;
```

```
  }
}
```

Registering custom converters

The just-implemented converters now have to be registered with the mapping subsystem. Both the Spring XML namespace as well as the provided Spring JavaConfig configuration base class make this very easy. In the XML world, it's just a matter of declaring a nested element inside `<mongo:mapping-converter />` and activating component scanning by setting the `base-package` attribute. See Example 6-29.

Example 6-29. Registering custom converters with the XML namespace

```
<mongo:mapping-converter id="mongoConverter" base-package="com.oreilly.springdata.mongodb">
  <mongo:custom-converters base-package="com.oreilly.springdata.mongodb" />
</mongo:mapping-converter>
```

In the JavaConfig world, the configuration base class provides a callback method for you to return an instance of `CustomConversions`. This class is a wrapper around the `Converter` instances you hand it, which we can inspect later to configure the `MappingContext` and `MongoConverter` appropriately, as well as the `ConversionService` to eventually perform the conversions. In Example 6-30, we access the `Converter` instances by enabling component scanning and autowiring them into the configuration class to eventually wrap them into the `CustomConversions` instance.

Example 6-30. Registering custom converters using Spring JavaConfig

```
@Configuration
@ComponentScan
class ApplicationConfig extends AbstractMongoConfiguration {

  @Autowired
  private List<Converter<?, ?>> converters;

  @Override
  public CustomConversions customConversions() {
    return new CustomConversions(converters);
  }
}
```

If we now obtain a `MappingMongoConverter` from the application context and invoke a conversion, as demonstrated in Example 6-22, the output would change to that shown in Example 6-31.

Example 6-31. Document structure with custom converters for EmailAddress applied

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : "dave@dmband.com", … }
```

MongoTemplate

Now that we have both the general infrastructure in place and understand the way that object mapping works and can be configured, let's continue with the API we provide to interact with the store. As with all the other Spring Data modules, the core of the API is the `MongoOperations` interface, backed by a `MongoTemplate` implementation. Template implementations in Spring serve two primary purposes: resource management and exception translation. This means that `MongoTemplate` will take care of acquiring a connection through the configured `MongoDbFactory` and clean it up properly after the interaction with the store has ended or an exception has occurred. Exceptions being thrown by MongoDB will be transparently converted into Spring's `DataAccessException` hierarchy to prevent the clients from having to know about the persistence technology being used.

To illustrate the usage of the API, we will look at a repository implementation of the `CustomerRepository` interface (Example 6-32). It's called `MongoDbCustomerRepository` and located in the `com.oreilly.springdata.mongodb.core` package.

Example 6-32. MongoDbCustomerRepository implementation

```
import static org.springframework.data.mongodb.core.query.Criteria.*;
import static org.springframework.data.mongodb.core.query.Query.*;
…

@Repository
@Profile("mongodb")
class MongoDbCustomerRepository implements CustomerRepository {

  private final MongoOperations operations;

  @Autowired
  public MongoDbCustomerRepository(MongoOperations operations) {
    Assert.notNull(operations);
    this.operations = operations;
  }

  @Override
  public Customer findOne(Long id) {

    Query query = query(where("id").is(id));
    return operations.findOne(query, Customer.class);
  }

  @Override
  public Customer save(Customer customer) {

    operations.save(customer);
    return customer;
  }

  @Override
  public Customer findByEmailAddress(EmailAddress emailAddress) {
```

```
    Query query = query(where("emailAddress").is(emailAddress));
    return operations.findOne(query, Customer.class);
  }
}
```

As you can see, we have a standard Spring component annotated with `@Repository` to make the class discoverable by classpath scanning. We add the `@Profile` annotation to make sure it will be activated only if the configured Spring profile is activated. This will prevent the class from leaking into the default bean setup, which we will use later when introducing the Spring Data repositories for MongoDB.

The class's only dependency is `MongoOperations`; this is the interface of `MongoTemplate`, which we have configured in our application context (see *application-context.xml* or `ApplicationConfig` class [Example 6-12]). `MongoTemplate` provides two categories of methods to be used:

- General-purpose, high-level methods that enable you to execute commonly needed operations as one-line statements. This includes basic functions like `findOne(…)`, `findAll(…)`, `save(…)`, and `delete(…)`; more MongoDB-specific ones like `updateFirst(…)`, `updateMulti(…)`, and `upsert(…)`; and map-reduce and geospatial operations like `mapReduce(…)` and `geoNear(…)`. All these methods automatically apply the object-to-store mapping discussed in"The Mapping Subsystem" on page 83.
- Low-level, callback-driven methods that allow you to interact with the MongoDB driver API in the even that the high-level operations don't suffice for the functionality you need. These methods all start with `execute` and take either a `CollectionCallback` (providing access to a MongoDb `DbCollection`), `DbCallback` (providing access to a MongoDB `DB`), or a `DocumentCallbackHandler` (to process a `DBObject` directly).

The simplest example of the usage of the high-level API is the `save(…)` method of `MongoDbCustomerRepository`. We simply use the `save(…)` method of the `MongoOperations` interface to hand it the provided `Customer`. It will in turn convert the domain object into a MongoDB `DBObject` and save that using the MongoDB driver API.

The two other methods in the implementation—`findOne(…)` and `findByEmailAddress(…)`—use the query API provided by Spring Data MongoDB to ease creating queries to access MongoDB documents. The `query(…)` method is actually a static factory method of the `Query` class statically imported at the very top of the class declaration. The same applies to the `where(…)` method, except it's originating from `Criteria`. As you can see, defining a query is remarkably simple. Still, there are a few things to notice here.

In `findByEmailAddress(…)`, we reference the `emailAddress` property of the `Customer` class. Because it has been mapped to the `email` key by the `@Field` annotation, the property reference will be automatically translated into the correct field reference. Also, we hand the plain `EmailAddress` object to the criteria to build the equality predicate. It will also be transformed by the mapping subsystem before the query is actually applied. This

includes custom conversions registered for the given type as well. Thus, the `DBObject` representing the query will look something like Example 6-33.

Example 6-33. The translated query object for findByEmailAddress(…)

```
{ "email" : "dave@dmband.com" }
```

As you can see, the field key was correctly translated to `email` and the value object properly inlined due to the custom converter for the `EmailAddress` class we introduced in "Implementing custom converters" on page 92.

Mongo Repositories

As just described, the `MongoOperations` interface provides a decent API to implement a repository manually. However, we can simplify this process even further using the Spring Data repository abstraction, introduced in Chapter 2. We'll walk through the repository interface declarations of the sample project and see how invocations to the repository methods get handled.

Infrastructure Setup

We activate the repository mechanism by using either a JavaConfig annotation (Example 6-34) or an XML namespace element.

Example 6-34. Activating Spring Data MongoDB repositories in JavaConfig

```
@Configuration
@ComponentScan(basePackageClasses = ApplicationConfig.class)
@EnableMongoRepositories
public class ApplicationConfig extends AbstractMongoConfiguration {

  …
}
```

In this configuration sample, the `@EnableMongoRepositories` is the crucial part. It will set up the repository infrastructure to scan for repository interfaces in the package of the annotated configuration class by default. We can alter this by configuring either the `basePackage` or `basePackageClasses` attributes of the annotation. The XML equivalent looks very similar, except we have to configure the base package manually (Example 6-35).

Example 6-35. Activating Spring Data MongoDB repositories in XML

```
<mongo:repositories base-package="com.oreilly.springdata.mongodb" />
```

Repositories in Detail

For each of the repositories in the sample application, there is a corresponding integration test that we can run against a local MongoDB instance. These tests interact with the repository and invoke the methods exposed. With the log level set to `DEBUG`, you should be able to follow the actual discovery steps, query execution, etc.

Let's start with `CustomerRepository` since it's the most basic one. It essentially looks like Example 6-36.

Example 6-36. CustomerRepository interface

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

  Customer findOne(Long id);

  Customer save(Customer customer);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

The first two methods are essentially CRUD methods and will be routed into the generic `SimpleMongoRepository`, which provides the declared methods. The general mechanism for that is discussed in Chapter 2. So the really interesting method is `findByEmailAddress(…)`. Because we don't have a manual query defined, the query derivation mechanism will kick in, parse the method, and derive a query from it. Since we reference the `emailAddress` property, the logical query derived is essentially `emailAddress = ?0`. Thus, the infrastructure will create a `Query` instance using the Spring Data MongoDB query API. This will in turn translate the property reference into the appropriate field mapping so that we end up using the query `{ email : ?0 }`. On method invocation, the given parameters will be bound to the query and executed eventually.

The next repository is `PersonRepository`, shown in Example 6-37.

Example 6-37. PersonRepository interface

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescriptionContaining(String description, Pageable pageable);

  @Query("{ ?0 : ?1 }")
  List<Product> findByAttributes(String key, String value);
}
```

The first thing to notice is that `ProductRepository` extends `CrudRepository` instead of the plain `Repository` marker interface. This causes the CRUD methods to be pulled into our repository definition. Thus, we don't have to manually declare `findOne(…)` and `save(…)` manually. `findByDescriptionContaining(…)` once again uses the query derivation mechanism, just as we have seen in `CustomerRepository.findByEmailAddress(…)`. The difference from the former method is that this one additionally qualifies the

predicate with the `Containing` keyword. This will cause the description parameter handed into the method call to be massaged into a regular expression to match descriptions that contain the given `String` as a substring.

The second thing worth noting here is the use of the pagination API (introduced in "Pagination and Sorting" on page 18). Clients can hand in a `Pageable` to the method to restrict the results returned to a certain page with a given number and page size, as shown in Example 6-38. The returned `Page` then contains the results plus some meta-information, such as about how many pages there are in total. You can see a sample usage of the method in `PersonRepositoryIntegrationTests`: the `lookupProductsByDescription()` method.

Example 6-38. Using the findByDescriptionContaining(…) method

```
Pageable pageable = new PageRequest(0, 1, Direction.DESC, "name");
Page<Product> page = repository.findByDescriptionContaining("Apple", pageable);

assertThat(page.getContent(), hasSize(1));
assertThat(page, Matchers.<Product> hasItems(named("iPad")));
assertThat(page.isFirstPage(), is(true));
assertThat(page.isLastPage(), is(false));
assertThat(page.hasNextPage(), is(true));
```

First, we set up a `PageRequest` to request the first page with a page size of 1, requiring the results to be sorted in descending order by name. See how the returned page provides not only the results, but also information on where the returned page is located in the global set of pages.

The second method (refer back to Example 6-37) declared uses the `@Query` annotation to manually define a MongoDB query. This comes in handy if the query derivation mechanism does not provide the functionality you need for the query, or the query method's name is awkwardly long. We set up a general query `{ ?0 : ?1 }` to bind the first argument of the method to act as key and the second one to act as value. The client can now use this method the query for `Product`s that have a particular attribute (e.g., a dock connector plug), as shown in Example 6-39.

Example 6-39. Querying for Products with a dock connector plug

```
List<Product> products = repository.findByAttributes("attributes.connector", "plug");

assertThat(products, Matchers.<Product> hasItems(named("Dock")));
```

As expected, the iPod dock is returned from the method call. This way, the business logic could easily implement `Product` recommendations based on matching attribute pairs (connector plug and socket).

Last but not least, let's have a look at the `OrderRepository` (Example 6-40). Given that we already discussed two repository interfaces, the last one shouldn't come with too many surprises.

Example 6-40. OrderRepository interface

```
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {

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

The query method declared here is just a straightforward one using the query derivation mechanism. What has changed compared to the previous repository interfaces is the base interface we extend from. Inheriting from `PagingAndSortingRepository` not only exposes CRUD methods, but also methods like `findAll(Pageable pageable)` allow for paginating the entire set of `Order`s in a convenient way. For more information on the pagination API in general, see "Pagination and Sorting" on page 18.

Mongo Querydsl Integration

Now that we've seen how to add query methods to repository interfaces, let's have a look at how we can use Querydsl to dynamically create predicates for entities and execute them via the repository abstraction. Chapter 3 provides a general introduction to what Querydsl actually is and how it works. If you've read "Repository Querydsl Integration" on page 51, you'll see how remarkably similar the setup and usage of the API is, although we query a totally different store.

To generate the metamodel classes, we have configured the Querydsl Maven plug-in in our *pom.xml* file, as shown in Example 6-41.

Example 6-41. Setting up the Querydsl APT processor for MongoDB

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>maven-apt-plugin</artifactId>
  <version>1.0.5</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>process</goal>
      </goals>
      <configuration>
        <outputDirectory>target/generated-sources</outputDirectory>
        <processor>…data.mongodb.repository.support.MongoAnnotationProcessor</processor>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The only difference from the JPA approach is that we configure a `MongoAnnotationProcessor`. It will configure the APT processor to inspect the annotations provided by the Spring Data MongoDB mapping subsystem to generate the metamodel correctly. Beyond that, we provide integration to let Querydsl consider our mappings—and thus potentially registered custom converters—when creating the MongoDB queries.

To include the API to execute predicates built with the generated metamodel classes, we let the `ProductRepository` additionally extend `QueryDslPredicateExecutor` (Example 6-42).

Example 6-42. The ProductRepository interface extending QueryDslPredicateExecutor

```
public interface ProductRepository extends CrudRepository<Product, Long>,
                                           QueryDslPredicateExecutor<Product> { … }
```

The `QuerydslProductRepositoryIntegrationTest` now shows how to make use of the predicates. Again, the code is pretty much a 1:1 copy of the JPA code. We obtain a reference iPad by executing the `product.name.eq("iPad")` predicate and use that to verify the result of the execution of the predicate, looking up products by description, as shown in Example 6-43.

Example 6-43. Using Querydsl predicates to query for Products

```
QProduct product = QProduct.product;

Product iPad = repository.findOne(product.name.eq("iPad"));
Predicate tablets = product.description.contains("tablet");

Iterable<Product> result = repository.findAll(tablets);
assertThat(result, is(Matchers.<Product> iterableWithSize(1)));
assertThat(result, hasItem(iPad));
```

CHAPTER 7

Neo4j: A Graph Database

Graph Databases

This chapter introduces an interesting kind of NoSQL store: graph databases. Graph databases are clearly post-relational data stores, because they evolve several database concepts much further while keeping other attributes. They provide the means of storing semistructured but highly connected data efficiently and allow us to query and traverse the linked data at a very high speed.

Graph data consists of nodes connected with directed and labeled relationships. In property graphs, both nodes and relationships can hold arbitrary key/value pairs. Graphs form an intricate network of those elements and encourage us to model domain and real-world data close to the original structure. Unlike relational databases, which rely on fixed schemas to model data, graph databases are schema-free and put no constraints onto the data structure. Relationships can be added and changed easily, because they are not part of a schema but rather part of the actual data.

We can attribute the high performance of graph databases to the fact that moving the cost of relating entities (joins) to the insertion time—by materializing the relationships as first-level citizens of the data structure—allows for constant time traversal from one entity (node) to another. So, regardless of the dataset size, the time for a given traversal across the graph is always determined by the number of hops in that traversal, not the number of nodes and relationships in the graph as a whole. In other database models, the cost of finding connections between two (or more) entities occurs on each query instead.

Thanks to this, a single graph can store many different domains, creating interesting connections between entities from all of them. Secondary access or index structures can be integrated into the graph to allow special grouping or access paths to a number of nodes or subgraphs.

Due to the nature of graph databases, they don't rely on aggregate bounds to manage atomic operations but instead build on the well-established transactional guarantees of an ACID (atomicity, consistency, isolation, durability) data store.

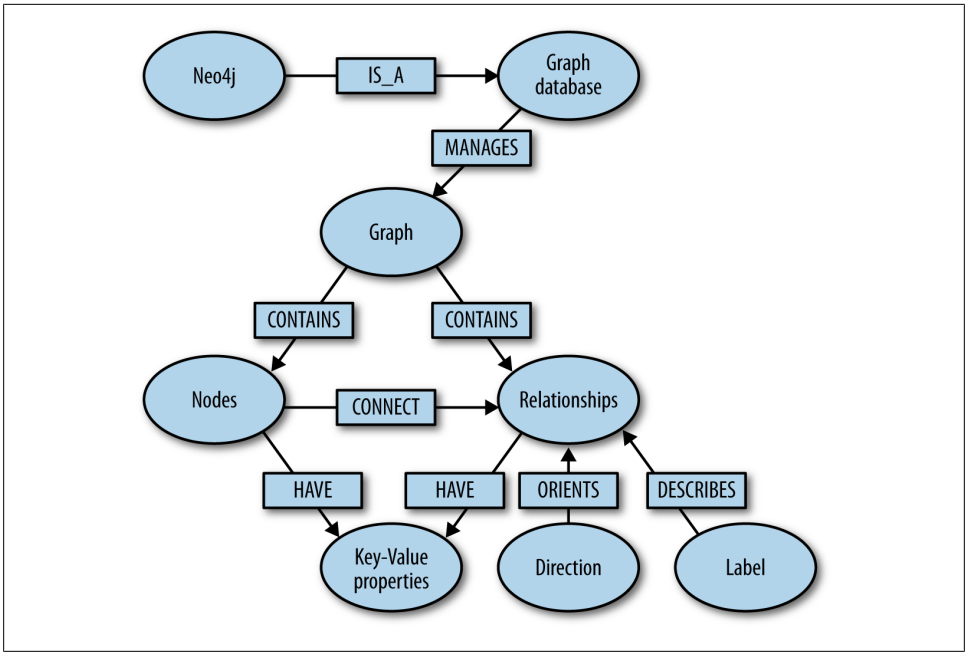

Figure 7-1. Graph database overview

Neo4j

Neo4j is the leading implementation of a property graph database. It is written predominantly in Java and leverages a custom storage format and the facilities of the Java Transaction Architecture (JTA) to provide XA transactions. The Java API offers an object-oriented way of working with the nodes and relationships of the graph (show in the example). Traversals are expressed with a fluent API. Being a graph database, Neo4j offers a number of graph algorithms like shortest path, Dijkstra, or A* out of the box.

Neo4j integrates a transactional, pluggable indexing subsystem that uses Lucene as the default. The index is used primarily to locate starting points for traversals. Its second use is to support unique entity creation. To start using Neo4j's embedded Java database, add the `org.neo4j:neo4j:<version>` dependency to your build setup, and you're ready to go. Example 7-1 lists the code for creating nodes and relationships with properties within transactional bounds. It shows how to access and read them later.

Example 7-1. Neo4j Core API Demonstration

```
GraphDatabaseService gdb = new EmbeddedGraphDatabase("path/to/database");

Transaction tx=gdb.beginTx();
try {
    Node dave = gdb.createNode();
    dave.setProperty("email","dave@dmband.com");
    gdb.index().forNodes("Customer").add
```

```
   (dave,"email",dave.getProperty("email");

    Node iPad = gdb.createNode();
    iPad.setProperty("name","Apple iPad");

    Relationship rel=dave.createRelationshipTo(iPad,Types.RATED);
    rel.setProperty("stars",5);

    tx.success();
} finally {
    tx.finish();
}

// to access the data

Node dave = gdb.index().forNodes("Customer").get("email","david@dmband.com").getSingle();
for (Relationship rating : dave.getRelationships(Direction.OUTGOING, Types.RATED)) {
    aggregate(rating.getEndNode(), rating.getProperty("stars"));
}
```

With the declarative Cypher query language, Neo4j makes it easier to get started for everyone who knows SQL from working with relational databases. Developers as well as operations and business users can run ad-hoc queries on the graph for a variety of use cases. Cypher draws its inspiration from a variety of sources: SQL, SparQL, ASCII-Art, and functional programming. The core concept is that the user describes the patterns to be matched in the graph and supplies starting points. The database engine then efficiently matches the given patterns across the graph, enabling users to define sophisticated queries like "find me all the customers who have friends who have recently bought similar products." Like other query languages, it supports filtering, grouping, and paging. Cypher allows easy creation, deletion, update, and graph construction.

The Cypher statement in Example 7-2 shows a typical use case. It starts by looking up a customer from an index and then following relationships via his orders to the products he ordered. Filtering out older orders, the query then calculates the top 20 largest volumes he purchased by product.

Example 7-2. Sample Cypher statement

```
START    customer=node:Customer(email = "dave@dmband.com")
MATCH    customer-[:ORDERED]->order-[item:LINEITEM]->product
WHERE    order.date > 20120101
RETURN   product.name, sum(item.amount) AS product
ORDER BY products DESC
LIMIT    20
```

Being written in Java, Neo4j is easily embeddable in any Java application which refers to single-instance deployments. However, many deployments of Neo4j use the standalone Neo4j server, which offers a convenient HTTP API for easy interaction as well as a comprehensive web interface for administration, exploration, visualization, and monitoring purposes. The Neo4j server is a simple download, and can be uncompressed and started directly.

It is possible to run the Neo4j server on top of an embedded database, which allows easy access to the web interface for inspection and monitoring (Figure 7-2).

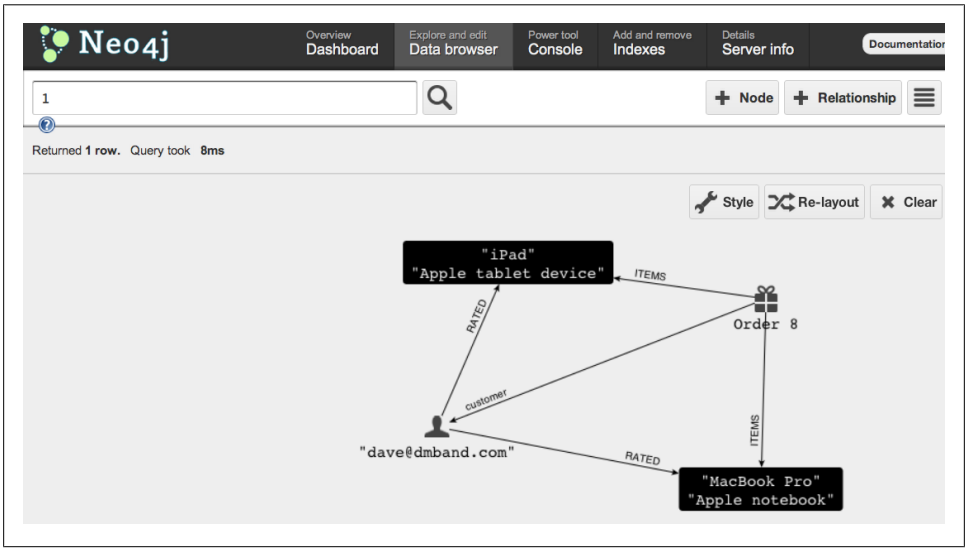

Figure 7-2. Neo4j server web interface

In the web interface, you can see statistics about your database. In the data browser, you can find nodes by ID, with index lookups, and with cypher queries (click the little blue question mark for syntax help), and switch to the graph visualizer with the right-hand button to explore your graph visually (as shown in Figure 7-2). The console allows you to enter Cypher statements directly or even issue HTTP requests. Server Info lists JMX beans, which, especially in the Enterprise edition, come with much more information.

As an open source product, Neo4j has a very rich and active ecosystem of contributors, community members, and users. Neo Technology, the company sponsoring the development of Neo4j, makes sure that the open source licensing (GPL) for the community edition, as well as the professional support for the enterprise editions, promote the continuous development of the product.

To access Neo4j, you have a variety of drivers available, most of them being maintained by the community. There are libraries for many programming languages for both the embedded and the server deployment mode. Some are maintained by the Neo4j team, Spring Data Neo4j being one of them.

Spring Data Neo4j Overview

Spring Data Neo4j was the original Spring Data project initiated by Rod Johnson and Emil Eifrem. It was developed in close collaboration with VMware and Neo Technology and offers Spring developers an easy and familiar way to interact with Neo4j. It intends to leverage the well-known annotation-based programming models with a tight integration in the Spring Framework ecosystem. As part of the Spring Data project, Spring Data Neo4j integrates both Spring Data Commons repositories (see Chapter 2) as well as other common infrastructures.

As in JPA, a few annotations on POJO (plain old Java object) entities and their fields provide the necessary metainformation for Spring Data Neo4j to map Java objects into graph elements. There are annotations for entities being backed by nodes (`@NodeEntity`) or relationships (`@RelationshipEntity`). Field annotations declare relationships to other entities (`@RelatedTo`), custom conversions, automatic indexing (`@Indexed`), or computed/derived values (`@Query`). Spring Data Neo4j allows us to store the type information (hierarchy) of the entities for performing advanced operations and type conversions. See Example 7-3.

Example 7-3. An annotated domain class

```
@NodeEntity
public class Customer {
  @GraphId Long id;

  String firstName, lastName;

  @Indexed(unique = true)
  String emailAddress;

  @RelatedTo(type = "ADDRESS")
  Set<Address> addresses = new HashSet<Address>();
}
```

The core infrastructure of Spring Data Neo4j is the `Neo4jTemplate`, which offers (similar to other template implementations) a variety of lower-level functionality that encapsulates the Neo4j API to support mapped domain objects. The Spring Data Neo4j infrastructure and the repository implementation uses the `Neo4jTemplate` for its operations. Like the other Spring Data projects, Spring Data Neo4j is configured via two XML namespace elements—for general setup and repository configuration.

To tailor Neo4j to individual use cases, Spring Data Neo4j supports both the embedded mode of Neo4j as well as the server deployment, where the latter is accessed via Neo4j's Java-REST binding. Two different mapping modes support the custom needs of developers. In the simple mapping mode, the graph data is copied into domain objects, being detached from the graph. The more advanced mapping mode leverages AspectJ to provide a live, connected representation of the graph elements bound to the domain objects.

Modeling the Domain as a Graph

The domain model described in Chapter 1 is already a good fit for a graph database like Neo4j (see Figure 7-3). To allow some more advanced graph operations, we're going to normalize it further and add some additional relationships to enrich the model.

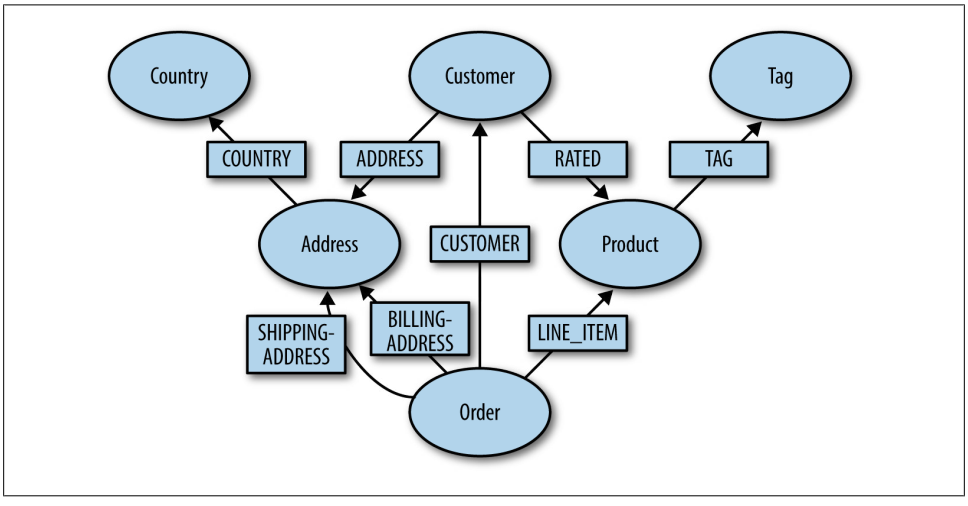

Figure 7-3. Domain model as a graph

The code samples listed here are not complete but contain the necessary information for understanding the mapping concepts. See the Neo4j project in the sample source-repository for a more complete picture.

In Example 7-4, the `AbstractEntity` as a superclass was kept with the same `id` field (which got a `@GraphId` annotation and `equals(…)` and `hashCode()` methods, as previously discussed). Annotating the `id` is required in the simple mapping mode, as it is the only way to keep the node or relationship `id` stored in the entity. Entities can be be loaded by their `id` with `Neo4jTemplate.findOne()`, and a similar method exists in the `GraphRepository`.

Example 7-4. Base domain class

```
public abstract class AbstractEntity {

  @GraphId
  private Long id;
}
```

The simplest mapped class is just marked with `@NodeEntity` to make it known to Spring Data Neo4j's mapping infrastructure. It can contain any number of primitive fields, which will be treated as node properties. Primitive types are mapped directly. Types

not supported by Neo4j can be converted to equivalent primitive representations by supplied Spring converters. Converters for `Enum` and `Date` fields come with the library.

In `Country`, both fields are just simple strings, as shown in Example 7-5. The `code` field represents a unique "business" key and is marked as `@Indexed(unique=true)` which causes the built-in facilities for unique indexes to be used; these are exposed via `Neo4jTemplate.getOrCreateNode()`. There are several methods in the `Neo4jTemplate` to access the Neo4j indexes; we can find entities by their indexed keys with `Neo4jTemplate.lookup()`.

Example 7-5. Country as a simple entity

```
@NodeEntity
public class Country extends AbstractEntity {

  @Indexed(unique=true)
  String code;
  String name;
}
```

Customers are stored as nodes; their unique key is the `emailAddress`. Here we meet the first references to other objects (in this case, `Address`), which are represented as relationships in the graph. So fields of single references or collections of references always cause relationships to be created when updated, or navigated when accessed.

As shown in Example 7-6, reference fields can be annotated with `@RelatedTo`, to document the fact that they are reference fields or set custom attributes like the relationship type (in this case, `"ADDRESS"`). If we do not provide the type, it defaults to the field name. The relationship points by default to the referred object (`Direction.OUTGOING`), the opposite direction can be specified in the annotation; this is especially important for bidirectional references, which should be mapped to just a single relationship.

Example 7-6. Customer has relationships to his addresses

```
@NodeEntity
public class Customer extends AbstractEntity {

  private String firstName, lastName;

  @Indexed(unique = true)
  private String emailAddress;

  @RelatedTo(type = "ADDRESS")
  private Set<Address> addresses = new HashSet<Address>();
}
```

The `Address` is pretty simple again. Example 7-7 shows how the `country` reference field doesn't have to be annotated—it just uses the field name as the relationship type for the outgoing relationship. The customers connected to this address are not represented in the mapping because they are not necessary for our use case.

Example 7-7. Address connected to country

```
@NodeEntity
public class Address extends AbstractEntity {

  private String street, city;
  private Country country;
}
```

The `Product` has a unique name and shows the use of a nonprimitive field; the price will be converted to a primitive representation by Springs' converter facilities. You can register your own converters for custom types (e.g., value objects) in your application context.

The description field will be indexed by an index that allows full-text search. We have to name the index explicitly, as it uses a different configuration than the default, exact index. You can then find the products by calling, for instance, `neo4jTemplate.lookup("search","description:Mac*")`, which takes a Lucene query string.

To enable interesting graph operations, we added a `Tag` entity and relate to it from the `Product`. These tags can be used to find similar products, provide recommendations, or analyze buying behavior.

To handle dynamic attributes of an entity (a map of arbitrary key/values), there is a special support class in Spring Data Neo4j. We decided against handling maps directly because they come with a lot of additional semantics that don't fit in the context. Currently, `DynamicProperties` are converted into properties of the node with prefixed names for separation. (See Example 7-8.)

Example 7-8. Tagged product with custom dynamic attributes

```
@NodeEntity
public class Product extends AbstractEntity {

  @Indexed(unique = true)
  private String name;
  @Indexed(indexType = IndexType.FULLTEXT, indexName = "search")
  private String description;
  private BigDecimal price;

  @RelatedTo
  private Set<Tag> tags = new HashSet<Tag> ();
  private DynamicProperties attributes = new PrefixedDynamicProperties("attributes");
}
```

The only unusual thing about the `Tag` is the `Object value` property. This property is converted according to the runtime value into a primitive value that can be stored by Neo4j. The `@GraphProperty` annotation, as shown in Example 7-9, allows some customization of the storage (e.g., the used property name or a specification of the primitive target type in the graph).

Example 7-9. A simple Tag

```
@NodeEntity
public class Tag extends AbstractEntity {

  @Indexed(unique = true)
  String name;

  @GraphProperty
  Object value;
}
```

The first `@RelationshipEntity` we encounter is something new that didn't exist in the original domain model but which is nonetheless well known from any website. To allow for some more interesting graph operations we add a `Rating` relationship between a `Customer` and a `Product`. This entity is annotated with `@RelationshipEntity` to mark it as such. Besides two simple fields holding the rating `stars` and a `comment`, we can see that it contains fields for the actual start and end of the relationship, which are annotated appropriately (Example 7-10).

Example 7-10. A Rating between Customer and Product

```
@RelationshipEntity(type = "RATED")
public class Rating extends AbstractEntity {
    @StartNode Customer customer;
    @EndNode Product product;
    int stars;
    String comment;
}
```

Relationship entities can be created as normal POJO classes, supplied with their start and endpoints, and saved via `Neo4jTemplate.save()`. In Example 7-11, we show with the `Order` how these entities can be retrieved as part of the mapping. In the more in-depth discussion of graph operations—see "Leverage Similar Interests (Collaborative Filtering)" on page 121—we'll see how to leverage those relationships in Cypher queries with `Neo4jTemplate.query` or repository finder methods.

The `Order` is the most connected entity so far; it sits in the middle of our domain. In Example 7-11, the relationship to the `Customer` shows the inverse `Direction.INCOMING` for a bidirectional reference that shares the same relationship.

The easiest way to model the different types of addresses (shipping and billing) is to use different relationship types—in this case, we just rely on the different field names. Please note that a single address object/node can be used in multiple places for example, as both the shipping and billing address of a single customer, or even across customers (e.g., for a family). In practice, a graph is often much more normalized than a relational database, and the removal of duplication actually offers multiple benefits both in terms of storage and the ability to run more interesting queries.

Example 7-11. Order, the centerpiece of the domain

```
@NodeEntity
public class Order extends AbstractEntity {

  @RelatedTo(type = "ORDERED", direction = Direction.INCOMING)
  private Customer customer;

  @RelatedTo
  private Address billingAddress;

  @RelatedTo
  private Address shippingAddress;

  @Fetch
  @RelatedToVia
  private Set<LineItem> lineItems = new HashSet<LineItem>();
}
```

The `LineItem`s are not modeled as nodes but rather as relationships between `Order` and `Product`. A `LineItem` has no identity of its own and just exists as long as both its endpoints exist, which it refers to via its `order` and `product` fields. In this model, `LineItem` only contains the `quantity` attribute, but in other use cases, it can also contain different attributes.

The interesting pieces in `Order` and `LineItem` are the `@RelatedToVia` annotation and `@Fetch`, which is discussed shortly. The annotation on the `lineItems` field is similar to `@RelatedTo` in that it applies only to references to relationship entities. It is possible to specify a custom relationship type or direction. The type would override the one provided in the `@RelationshipEntity` (see Example 7-12).

Example 7-12. A LineItem is just a relationship

```
@RelationshipEntity(type = "ITEMS")
public class LineItem extends AbstractEntity {

  @StartNode private Order order;

  @Fetch
  @EndNode
  private Product product;
  private int amount;
}
```

This takes us to one important aspect of object-graph mapping: fetch declarations. As we know from JPA, this can be tricky. For now we've kept things simple in Spring Data Neo4j by not fetching related entities by default.

Because the simple mapping mode needs to copy data out of the graph into objects, it must be careful about the fetch depth; otherwise you can easily end up with the whole graph pulled into memory, as graph structures are often cyclic. That's why the default strategy is to load related entities only in a shallow way. The `@Fetch` annotation is used

to declare fields to be loaded eagerly and fully. We can load them after the fact by `template.fetch(entity.field)`. This applies both to single relationships (one-to-one) and multi-relationship fields (one-to-many).

In the `Order`, the `LineItems` are fetched by default, becuse they are important in most cases when an order is loaded. For the `LineItem` itself, the `Product` is eagerly fetched so it is directly available. Depending on your use case, you would model it differently.

Now that we have created the domain classes, it's time to store their data in the graph.

Persisting Domain Objects with Spring Data Neo4j

Before we can start storing domain objects in the graph, we should set up the project. In addition to your usual Spring dependencies, you need either `org.springframework.data:spring-data-neo4j:2.1.0.RELEASE` (for simple mapping) or `org.springframework.data:spring-data-neo4j-aspects:2.1.0.RELEASE` (for advanced AspectJ-based mapping (see "Advanced Mapping Mode" on page 123) as a dependency. Neo4j is pulled in automatically (for simplicity, assuming the embedded Neo4j deployment).

The minimal Spring configuration is a single namespace config that also sets up the graph database (Example 7-13).

Example 7-13. Spring configuration setup

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/neo4j
        http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">


    <neo4j:config storeDirectory="target/graph.db" />
    <neo4j:repositories base-package="com.oreilly.springdata.neo4j" />

</beans>
```

As shown in Example 7-14, we can also pass a `graphDatabaseService` instance to `neo4j:config`, in order to configure the graph database in terms of caching, memory usage, or upgrade policies. This even allows you to use an in-memory `ImpermanentGraphDatabase` for testing.

Example 7-14. Passing a graphDatabaseService to the configuration

```
<neo4j:config graphDatabaseService="graphDatabaseService" />

<bean id="graphDatabaseService" class="org.neo4j.test.ImpermanentGraphDatabase" />

<!-- or -->
```

```
<bean id="graphDatabaseService" class="org.neo4j.kernel.EmbeddedGraphDatabase"
        destroy-method="shutdown">
    <constructor-arg value="target/graph.db" />
    <constructor-arg> <!-- passing configuration properties -->
        <map>
            <entry key="allow_store_upgrade" value="true" />
        </map>
    </constructor-arg>
</bean>
```

After defining the domain objects and the setup, we can pretty easily generate the sample dataset that will be used to illustrate some use cases (see Example 7-15 and Figure 7-4). Both the domain classes, as well as the dataset generation and integration tests documenting the use cases, can be found in the GitHub repository for the book (see "The Sample Code" on page 6 for details). To import the data, we can simply populate domain classes and use `template.save(entity)`, which either merges the entity with the existing element in the graph or creates a new one. That depends on mapped IDs and possibly unique field declarations, which would be used to identify existing entities in the graph with which we're merging.

Example 7-15. Populating the graph with the sample dataset

```
Customer dave = template.save(new Customer("Dave", "Matthews", "dave@dmband.com"));
template.save(new Customer("Carter","Beauford","carter@dmband.com"));
template.save(new Customer("Boyd","Tinsley","boyd@dmband.com"));

Country usa = template.save(new Country("US", "United States"));
template.save(new Address("27 Broadway","New York",usa));

Product iPad = template.save(new Product("iPad", "Apple tablet device").withPrice(499));
Product mbp = template.save(new Product("MacBook Pro", "Apple notebook").withPrice(1299));

template.save(new Order(dave).withItem(iPad,2).withItem(mbp,1));
```

The entities shown here use some convenience methods for construction to provide a more readable setup (Figure 7-4).

Neo4jTemplate

The Neo4jTemplate is like other Spring templates: a convenience API over a lower-level one, in this case the Neo4j API. It adds the usual benefits, like transaction handling and exception translation, but more importantly, automatic mapping from and to domain entities. The Neo4jTemplate is used in the other infrastructural parts of Spring Data Neo4j. Set it up by adding the `<neo4j:config/>` declaration to your application context or by creating a new instance, which is passed a Neo4j `GraphDatabaseService` (which is available as a Spring bean and can be injected into your code if you want to access the Neo4j API directly).

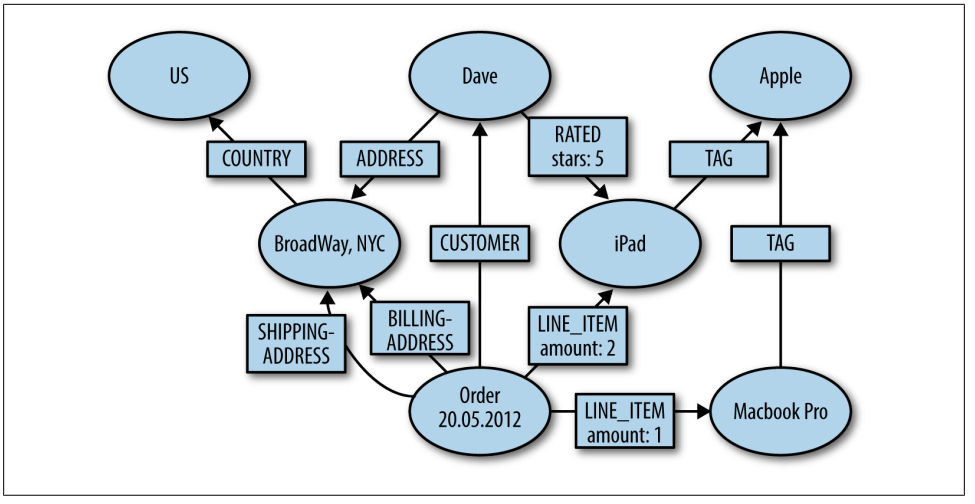

Figure 7-4. Graph of imported domain data

The operations for creating entities, nodes, and relationships and finding or removing them by `id` comprise the basics (`save()`, `getOrCreateNode()`, `findOne()`, `getNode()`, `getRelationshipsBetween()`, etc.). Most of the other mechanisms deal with more advanced ways to look up interesting things in the graph—by issuing index queries with `lookup`, executing Cypher statements with `query()`, or running a traversal with `traverse()`. The `Neo4jTemplate` offers methods to convert nodes into entities with `load()`, or one entity into a different type with `projectTo()` (see "Multiple Roles for a Single Node" on page 119). Lazily loaded entities can be loaded fully via `fetch()`.

You can achieve most of what you want to do with Spring Data Neo4j with the `Neo4jTemplate` alone, but the repository support adds a much more convenient way to perform many operations.

Combining Graph and Repository Power

With all that set up, we can now look into how repositories integrate with Spring Data Neo4j and how they are used in a "graphy" way.

Spring Data Commons repositories (see Chapter 2) make it easy to keep persistence access related code (or rather noncode) in one place and allow us to write as little of it as possible to satisfy the specific use cases. In Spring Data Neo4j, repositories are derived from the `GraphRepository<T>` base interface, which already combines some of the usually needed functionality: CRUD operations, and index and traversal functions. The basic setup for repositories is just another line of the namespace configuration, as shown in Example 7-16. Each domain class will be bound to an individual, concrete repository interface (see Example 7-17).

Example 7-16. Basic repository configuration setup

```
<neo4j:repositories base-package="com.oreilly.springdata.neo4j" />
```

Example 7-17. Basic repository interface declaration

```
import org.springframework.data.neo4j.repository.GraphRepository;

public interface CustomerRepository extends GraphRepository<Customer> {

  Customer findByEmailAddress(String emailAddress);
}
```

Spring Data Neo4j repositories provide support for `@Query`-annotated and derived finder methods, which are projected to Cypher statements. To understand how this mapping works, you need to be aware of the expressive syntax of Cypher, which is explained in the next sidebar, "Cypher Query Language".

Cypher Query Language

Neo4j comes with a clean, object-oriented Java API and enjoys many JVM (Java virtual machine) language bindings as well as a plethora of drivers for the Neo4j server. But often data operations are better expressed declaratively by asking "what" than by specifying "how."

That's why the Cypher query language was developed. It builds upon matching patterns in a graph that are bound to specified nodes and relationships and allows further filtering and paging of the results. Cypher has data manipulation features that allow us to modify the graph. Cypher query parts can be chained (piped) to enable more advanced and powerful graph operations.

Each Cypher query can consist of several parts:

START
: Defines identifiers, binding nodes, and relationships either by index or ID lookup.
`START user=node:customers(name="dave@...")`

MATCH
: Uses ASCII-ART descriptions for patterns to find in the graph. Patterns are bound to identifiers and define new identifiers. Each of the subgraphs found during the query execution spawns an individual result.

```
MATCH user-[rating:RATED]->product
```

WHERE
: Filters the result using boolean expressions, and uses dot notation for accessing properties, functions, collection predicates and functions, arithmetic operators, etc.

```
WHERE user.name = "Dave" AND ANY(color in product.colors : color = 'red')
OR rating.stars > 3
```

SKIP LIMIT
: Paginates the results with offsets and sizes.

```
SKIP 20 LIMIT 10
```

RETURN
: Declares what to return from the query. If aggregation functions are used, all non-aggregated values will be used as grouping values.

```
return user.name, AVG(rating.stars) AS WEIGHT, product
```

ORDER BY
: Orders by properties or any other expression. `ORDER BY user.name ASC, count(*) DESC`

UPDATES
: There is more to Cypher. With `CREATE [UNIQUE], SET, DELETE`, the graph can be modified on the fly. `WITH` and `FOREACH` allow for more advanced query structures.

PARAMETERS
: Cypher can be passed in a map of parameters which can be referenced by key (or position). `start n=node({nodeId}) where n.name=~{0} return n`

The results returned by Cypher are inherently tabular, much like `JDBC ResultSets`. The column names serve as row-value keys.

There is a Java DSL for Cypher that, instead of using semantic-free strings for queries, offers a type-safe API to build up Cypher queries. It allows us to optionally leverage Querydsl (see Chapter 3) to build expressions for filters and index queries out of generated domain object literals. With an existing JDBC driver, cypher queries can be easily integrated into existing Java (Spring) applications and other JDBC tools.

Basic Graph Repository Operations

The basic operations provided by the repositories mimic those offered by the `Neo4jTemplate`, only bound to the declared repository domain class. So `findOne(…)`, `save(…)`, `delete(…)`, `findAll(…)`, and so on, take and return instances of the domain class.

Spring Data Neo4j stores the type (hierarchy) information of the mapped entities in the graph. It uses one of several strategies for this purpose, defaulting to an index-based storage. This type information is used for all repository and template methods that operate on all instances of a type and for verification of requested types versus stored types.

The updating repository methods are transactional by default, so there is no need to declare a transaction around them. For domain use cases, however, it is sensible to do so anyway, as usually more than one database operation is encapsulated by a business transaction. (This uses the Neo4j supplied support for `JtaTransactionManager`)

For index operations, specific methods like `findAllByPropertyValue()`, `findAllByQuery()`, and `findAllByRange()` exist in the `IndexRepository` and are mapped directly

to the underlying index infrastructure of Neo4j, but take the repository domain class and existing index-related annotations into account. Similar methods are exposed in the `TraversalRepository` whose `findAllByTraversal()` method allows direct access to the powerful graph traversal mechanisms of Neo4j. Other provided repository interfaces offer methods for spatial queries or the Cypher-DSL integration.

Derived and Annotated Finder Methods

Besides the previously discussed basic operations, Spring Data Neo4j repositories support custom finder methods by leveraging the Cypher query language. For both annotated and derived finder methods, additional `Pageable` and `Sort` method parameters are taken into account during query execution. They are converted into appropriate `ORDER BY`, `SKIP`, and `LIMIT` declarations.

Annotated finder methods

Finders can use Cypher directly if we add a `@Query` annotation that contains the query string, as shown in Example 7-18. The method arguments are passed as parameters to the Cypher query, either via their parameter position or named according to their `@Parameter` annotation, so you can use `{index}` or `{name}` in the query string.

Example 7-18. An annotated cypher query on a repository query method

```
public interface OrderRepository extends GraphRepository<Order> {

  @Query(" START c=node({0}) " +
         " MATCH c-[:ORDERED]->order-[item:LINE_ITEM]->product " +
         "  WITH order, SUM (product.price * item.amount) AS value " +
         " WHERE value > {orderValue} " +
         "RETURN order")
  Collection<Order> findOrdersWithMinimumValue(Customer customer,
                                @Parameter("orderValue") int value);
}
```

Result handling

The return types of finder methods can be either an `Iterable<T>`, in which case the evaluation of the query happens lazily, or any of these interfaces: `Collection<T>`, `List<T>`, `Set<T>`, `Page<T>`. `T` is the result type of the query, which can be either a mapped domain entity (when returning nodes or relationships) or a primitive type. There is support for an interface-based simple mapping of query results. For mapping the results, we have to create an interface annotated with `@MapResult`. In the interface we declare methods for retrieving each column-value. We annotate the methods individually with `@ResultColumn("columnName")`. See Example 7-19.

Example 7-19. Defining a MapResult and using it in an interface method

```
@MapResult
interface RatedProduct {
```

```
  @ResultColumn("product")
  Product getProduct();

  @ResultColumn("rating")
  Float getRating();

  @ResultColumn("count")
  int getCount();
}

public interface ProductRepository extends GraphRepository<Product> {

  @Query(" START tag=node({0}) " +
         " MATCH tag-[:TAG]->product<-[rating:RATED]-() " +
          "RETURN product, avg(rating.stars) AS rating, count(*) as count " +
          " ORDER BY rating DESC")
  Page<RatedProduct> getTopRatedProductsForTag(Tag tag, Pageable page);
}
```

To avoid the proliferation of query methods for different granularities, result types, and container classes, Spring Data Neo4j provides a small fluent API for result handling. The API covers automatic and programmatic value conversion. The core of the result handling API centers on converting an iterable result into different types using a configured or given `ResultConverter`, deciding on the granularity of the result size and optionally on the type of the target container. See Example 7-20.

Example 7-20. Result handling API

```
public interface ProductRepository extends GraphRepository<Product> {

  Result<Map<String,Object>> findByName(String name);
}

Result<Map<String,Object>> result = repository.findByName("mac");

// return a single node (or null if nothing found)
Node n = result.to(Node.class).singleOrNull();
Page<Product> page = result.to(Product.class).as(Page.class);

Iterable<String> names = result.to(String.class,
  new ResultConverter<Map<String, Object>, String>() {
    public String convert(Map<String, Object> row) {
       return (String) ((Node) row.get("name")).getProperty("name");
    }
  });
```

Derived finder methods

As described in Chapter 2, the derived finder methods (see "Property expressions" on page 17) are a real differentiator. They leverage the existing mapping information about the targeted domain entity and an intelligent parsing of the finder method name to generate a query that fetches the information needed.

Derived finder methods—like `ProductRepository.findByNameAndColorAndTagName(name, color, tagName)`—start with `find(By)` or `get(By)` and then contain a succession of property expressions. Each of the property expressions either points to a property name of the current type or to another, related domain entity type and one of its properties. These properties must exist on the entity. If that is not the case, the repository creation fails early during `ApplicationContext` startup.

For all valid finder methods, the repository constructs an appropriate query by using the mapping information about domain entities. Many aspects—like in-graph type representation, indexing information, field types, relationship types, and directions—are taken into account during the query construction. This is also the point at which appropriate escaping takes place.

Thus, Example 7-20 would be converted to the query shown in Example 7-21.

Example 7-21. Derived query generation

```
@NodeEntity
class Product {

  @Indexed
  String name;
  int price;

  @RelatedTo(type = "TAG")
  Set<Tag> tags;
}

@NodeEntity
class Tag {

  @Indexed
  String name;
}

public interface ProductRepository extends GraphRepository<Product> {

  List<Product> findByNameAndPriceGreaterThanAndTagsName(String name, int price,
  String tagName);
}

// Generated query
 START product = node:Product(name = {0}), productTags = node:Tag(name = {3})
 MATCH product-[:TAG]->productTags
 WHERE product.price > {1}
RETURN product
```

This example demonstrates the use of index lookups for indexed attributes and the simple property comparison. If the method name refers to properties on other, related entities, then the query builder examines those entities for inclusion in the generated query. The builder also adds the direction and type of the relationship to that entity. If there are more properties further along the path, the same action is repeated.

Supported keywords for the property comparison are:

- Arithmetic comparisons like `GreaterThan`, `Equals`, or `NotEquals`.
- `IsNull` and `IsNotNull` check for `null` (or nonexistent) values.
- `Contains`, `StartsWith`, `EndsWith` and `Like` are used for string comparison.
- The `Not` prefix can be used to negate an expression.
- `Regexp` for matching regular expressions.

For many of the typical query use cases, it is easy enough to just code a derived finder declaration in the repository interface and use it. Only for more involved queries is an annotated query, traversal description, or manual traversing by following relationships necessary.

Advanced Graph Use Cases in the Example Domain

Besides the ease of mapping real-world, connected data into the graph, using the graph data model allows you to work with your data in interesting ways. By focusing on the value of relationships in your domain, you can find new insights and answers that are waiting to be revealed in the connections.

Multiple Roles for a Single Node

Due to the schema-free nature of Neo4j, a single node or relationship is not limited to be mapped to a single domain class. Sometimes it is sensible to structure your domain classes into smaller concepts/roles that are valid for a limited scope/context.

For example, an `Order` is used differently in different stages of its life cycle. Depending on the current state, it is either a shopping cart, a customer order, a dispatch note, or a return order. Each of those states is associated with different attributes, constraints, and operations. Usually, this would have been modeled either in different entities stored in separate tables or in a single `Order` class stored in a very large and sparse table row. With the schemaless nature of the graph database, the order will be stored in a node but only contains the state (and relationships) that are needed in the current state (and those still needed from past states). Usually, it gains attributes and relationships during its life, and gets simplified and locked down only when being retired.

Spring Data Neo4j allows us to model such entities with different classes, each of which covers one period of the life cycle. Those entities share a few attributes; each has some unique ones. All entities are mapped to the same node, and depending on the type provided at load time with `template.findOne(id,type)`, or at runtime with `template.projectTo(object, type)`, it can be used differently in different contexts. When the projected entity is stored, only its current attributes (and relationships) are updated; the other existing ones are left alone.

Product Categories and Tags as Examples for In-Graph Indexes

For handling larger product catalogs and ease of exploration, it is important to be able to put products into categories. A naive approach that uses a single category attribute with just one value per product falls short in terms of long-term usability. In a graph, multiple connections to category nodes per entity are quite natural. Adding a tree of categories, where each has relationships to its children and each product has relationships to the categories it belongs to, is really simple. Typical use cases are:

- Navigation of the category tree
- Listing of all products in a category subtree
- Listing similar products in the same category
- Finding implicit/non-obvious relationships between product categories (e.g., baby care products and lifestyle gadgets for young parents)

The same goes for tags, which are less restrictive than categories and often form a natural graph, with all the entities related to tags instead of a hierarchical tree like categories. In a graph database, both multiple categories as well as tags form implicit secondary indexing structures that allow navigational access to the stored entities in many different ways. There can be other secondary indexes (e.g., geoinformation, time-related indices, or other interesting dimensions). See Example 7-22.

Example 7-22. Product categories and tags

```
@NodeEntity
public class Category extends AbstractEntity {
  @Indexed(unique = true) String name;
  @Fetch // loads all children eagerly (cascading!)
  @RelatedTo(type="SUB_CAT")
  Set<Category> children = new HashSet<Category>();

  public void addChild(Category cat) {
    this.children.add(cat);
  }
}

@NodeEntity
public class Product extends AbstractEntity {
  @RelatedTo(type="CATEGORY")
  Set<Category> categories = new HashSet<Category>();

  public void addCategory(Category cat) {
    this.categories.add(cat);
  }
}

public interface ProductRepository extends GraphRepository<Product> {
  @Query("START cat=node:Category(name={0}) "+
         "MATCH cat-[SUB_CAT*0..5]-leaf<-[:CATEGORY]-product "+
         "RETURN distinct product")
```

```
  List<Product> findByCategory(String category);
}
```

The `Category` forms a nested composite structure with parent-child relationships. Each category has a unique name and a set of children. The category objects are used for creating the structure and relating products to categories. For leveraging the connectedness of the products, a custom (annotated) query navigates from a start (or root) category, via the next zero through five relationships, to the products connected to this subtree. All attached products are returned in a list.

Leverage Similar Interests (Collaborative Filtering)

`Collaborative filtering`, demonstrated in Example 7-23, relies on the assumption that we can find other "people" who are very similar/comparable to the current user in their interests or behavior. Which criteria are actually used for similarity—search/buying history, reviews, or others—is domain-specific. The more information the algorithm gets, the better the results.

In the next step, the products that those similar people also bought or liked are taken into consideration (measured by the number of their mentions and/or their rating scores) optionally excluding the items that the user has already bought, owns, or is not interested in.

Example 7-23. Collaborative filtering

```
public interface ProductRepository extends GraphRepository<Product> {
    @Query("START cust=node({0}) " +
           " MATCH cust-[r1:RATED]->product<-[r2:RATED]-people " +
           "           -[:ORDERED]->order-[:ITEMS]->suggestion " +
           " WHERE abs(r1.stars - r2.stars) <= 2 " +
           " RETURN suggestion, count(*) as score" +
           " ORDER BY score DESC")
    List<Suggestion> recommendItems(Customer customer);

    @MapResult
    interface Suggestion {
        @ResultColumn("suggestion") Product getProduct();
        @ResultColumn("score") Integer getScore();
    }
}
```

Recommendations

Generally in all domains, but particularly in the ecommerce domain, making recommendations of interesting products for customers is key to leveraging the collected information on product reviews and buying behavior. Obviously, we can derive recommendations from explicit customer reviews, especially if there is too little actual buying history or no connected user account. For the initial suggestion, a simple

ordering of listed products by number and review rating (or more advanced scoring mechanisms) is often sufficient.

For more advanced recommendations, we use algorithms that take multiple input data vectors into account (e.g., ratings, buying history, demographics, ad exposure, and geo-information).

The query in Example 7-24 looks up a product and all the ratings by any customer and returns a single page of top-rated products (depending on the average rating).

Example 7-24. Simple recommendation

```
public interface ProductRepository extends GraphRepository<Product> {
  @Query("START product=node:product_search({0}) "+
         "MATCH product<-[r:RATED]-customer "+
         "RETURN product ORDER BY avg(r.stars) DESC"
  Page<Product> listProductsRanked(String description, Pageable page);

}
```

Transactions, Entity Life Cycle, and Fetch Strategies

With Neo4j being a fully transactional database, Spring Data Neo4j participates in (declarative) Spring transaction management, and builds upon transaction managers provided by Neo4j that are compatible with the Spring `JtaTransactionManager`. The transaction-manager bean named `neo4jTransactionManager` (aliased to `transactionManager`) is created in the `<neo4j:config />` element. As transaction management is configured by default, `@Transactional` annotations are all that's needed to define transactional scopes. Transactions are needed for all write operations to the graph database, but reads don't need transactions. It is possible to nest transactions, but nested transactions will just participate in the running parent transaction (like `REQUIRED`).

Spring Data Neo4j, as well as Neo4j itself, can integrate with external XA transaction managers; the Neo4j manual describes the details.

For the simple mapping mode, the life cycle is straightforward: a new entity is just a POJO instance until it has been stored to the graph, in which case it will keep the internal `id` of the element (node or relationship) in the `@GraphId` annotated field for later reattachment or merging. Without the `id` set, it will be handled as a new entity and trigger the creation of a new graph element when saved.

Whenever entities are fetched in simple mapping mode from the graph, they are automatically detached. The data is copied out of the graph and stored in the domain object instances. An important aspect of using the simple mapping mode is the `fetch depth`. As a precaution, the transaction fetches only the direct properties of an entity and doesn't follow relationships by default when loading data.

To achieve a deeper fetch graph, we need to supply a `@Fetch` annotation on the fields that should be eagerly fetched. For entities and fields not already fetched, the `template.fetch(…)` method will load the data from the graph and update them in place.

Advanced Mapping Mode

Spring Data Neo4j also offers a more advanced mapping mode. Its main difference from the simple mapping mode is that it offers a live view of the graph projected into the domain objects. So each field access will be intercepted and routed to the appropriate properties or relationships (for `@RelatedTo[Via]` fields). This interception uses AspectJ under the hood to work its magic.

We can enable the advanced mapping mode by adding the `org.springframework.data:spring-data-neo4j-aspects` dependency and configuring either a AspectJ build plug-in or load-time-weaving activation (Example 7-25).

Example 7-25. Spring Data Neo4j advanced mapping setup

```
<properties>
  <aspectj.version>1.6.12</aspectj.version>
</properties>

<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j-aspects</artifactId>
  <version>${spring-data-neo4j.version}</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>${aspectj.version}</version>
</dependency>

....
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>aspectj-maven-plugin</artifactId>
  <version>1.2</version>
  <dependencies>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjrt</artifactId>
      <version>${aspectj.version}</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjtools</artifactId>
      <version>${aspectj.version}</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
```

```
      <goals>
        <goal>compile</goal>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <outxml>true</outxml>
    <aspectLibraries>
      <aspectLibrary>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
      </aspectLibrary>
      <aspectLibrary>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-neo4j-aspects</artifactId>
      </aspectLibrary>
    </aspectLibraries>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>
```

Fields are automatically read from the graph at any time, but for immediate write-through the operation must happen inside of a transaction. Because objects can be modified outside of a transaction, a life cycle of attached/detached objects has been established. Objects loaded from the graph or just `save`d inside a transaction are *attached*; if an object is modified outside of a transaction or newly created, it is `detached`. Changes to detached objects are stored in the object itself, and will only be reflected in the graph with the next `save` operation, causing the entity to become attached again.

This live view of the graph database allows for faster operation as well as "direct" manipulation of the graph. Changes will be immediately visible to other graph operations like traversals, Cypher queries, or Neo4j Core API methods. Because reads *always* happen against the live graph, all changes by other committed transactions are immediately visible. Due to the immediate live reads from the graph database, the advanced mapping mode has no need of fetch handling and the `@Fetch` annotation.

Working with Neo4j Server

We've already mentioned that Neo4j comes in two flavors. You can easily use the high-performance, embeddable Java database with any JVM language, preferably with that language's individual idiomatic APIs/drivers. Integrating the embedded database is as simple as adding the Neo4j libraries to your dependencies.

The other deployment option is Neo4j server. The Neo4j server module is a simple download or operating system package that is intended to be run as an independent service. Access to the server is provided via a web interface for monitoring, operations,

and visualizations (refer back to Example 7-1). A comprehensive REST API offers programmatic access to the database functionality. This REST API exposes a Cypher endpoint. Using the Neo4j-Java-Rest-Binding (which wraps the Neo4j Java API around the REST calls) to interact transparently with the server, Spring Data Neo4j can work easily with the server.

By depending on `org.springframework.data:spring-data-neo4j-rest` and changing the setup to point to the remote URL of the server, we can use Spring Data Neo4j with a server installation (Example 7-26). Please note that with the current implementation, not all calls are optimally transferred over the network API, so the server interaction for individual operations will be affected by network latency and bandwidth. It is recommended to use remotely executed queries as much as possible to reduce that impact.

Example 7-26. Server connection configuration setup

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/neo4j
        http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">


    <neo4j:config graphDatabaseService="graphDatabaseService" />
    <bean id="graphDatabaseService"
         class="org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
         <constructor-arg index="0" value="http://localhost:7474/db/data" />
    </bean>
</beans>
```

The `SpringRestGraphDatabase` connects via a `RestAPI` instance, which you can get to execute individual or batched REST operations more efficiently. For instance, creating entities with immediate property population, both for conventional or unique entities, is more efficient with the `RestAPI`.

Continuing From Here

This chapter presented some of the possibilities that graph databases—in particular Neo4j—offer and how Spring Data Neo4j gives you convenient access to them while keeping the doors open for raw, low-level graph processing.

The next thing you should do is consider the data you are working with (or want to work with) and see how connected the entities are. Look closely—you'll see they're often much more connected than you'd think at first glance. Taking one of these domains, and putting it first on a whiteboard and then into a graph database, is your first step toward realizing the power behind these concepts. For writing an application that uses the connected data, Spring Data Neo4j is an easy way to get started. It enables you

to easily create graph data and expose results of graph queries as your well-known POJOs, which eases the integration with other libraries and (UI) frameworks.

To learn how that process works for a complete web application, see [Hunger12] in the Bibliography, which is part of the reference documentation and the GitHub repository. The tutorial is a comprehensive walkthrough of creating the social movie database *cineasts.net*, and explains data modeling, integration with external services, and the web layer.

Feel free to reach out at any time to the Springsource Forums, Stackoverflow, or the Neo4j Google Group for answers to your questions. Enjoy!

CHAPTER 8

Redis: A Key/Value Store

In this chapter, we'll look at the support Spring Data offers for the key/value store Redis. We'll briefly look at how Redis manages data, show how to install and configure the server, and touch on how to interact with it from the command line. Then we'll look at how to connect to the server from Java and how the `RedisTemplate` organizes the multitude of operations we can perform on data stored in Redis. We'll look at ways to store POJOs using JSON, and we'll also briefly discuss how to use the fast and efficient pub/sub (publish/subscribe) capability to do basic event-based programming.

Redis in a Nutshell

Redis is an extremely high-performance, lightweight data store. It provides key/value data access to persistent byte arrays, lists, sets, and hash data structures. It supports atomic counters and also has an efficient topic-based pub/sub messaging functionality. Redis is simple to install and run and is, above all, very, very fast at data access. What it lacks in complex querying functionality (like that found in Riak or MongoDB), it makes up for in speed and efficiency. Redis servers can also be clustered together to provide for very flexible deployment. It's easy to interact with Redis from the command line using the *redis-cli* binary that comes with the installation.

Setting Up Redis

To start working with Redis, you'll want to have a local installation. Depending on your platform, the installation process ranges from easy to literally one command. The easiest installation process, shown in Example 8-1, is on Mac OS X using Homebrew. Other Unix systems are natively supported if you build the server from source. (Build instructions are on the Redis website, though they are identical to most other *NIX packages we've built—namely, unzip it, `cd` into that directory, and type **`make`**.) The download page for Redis also lists a couple of unofficial efforts to port Redis to the Win32/64 platform, though those are not considered production quality. For the purposes of this chapter, we'll stick to the *NIX version, where Redis is most at home.

Example 8-1. Installing Redis on Mac OS X using Homebrew

```
$ brew install redis
==> Downloading http://redis.googlecode.com/files/redis-2.4.15.tar.gz
######################################################################## 100.0%
==> make -C /private/tmp/homebrew-redis-2.4.15-WbS5/redis-2.4.15/src CC=/usr/bin/clang
==> Caveats
If this is your first install, automatically load on login with:
    mkdir -p ~/Library/LaunchAgents
    cp /usr/local/Cellar/redis/2.4.15/homebrew.mxcl.redis.plist ~/Library/LaunchAgents/
    launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist

If this is an upgrade and you already have the homebrew.mxcl.redis.plist loaded:
    launchctl unload -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist
    cp /usr/local/Cellar/redis/2.4.15/homebrew.mxcl.redis.plist ~/Library/LaunchAgents/
    launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist

  To start redis manually:
    redis-server /usr/local/etc/redis.conf

  To access the server:
    redis-cli
==> Summary
/usr/local/Cellar/redis/2.4.15: 9 files, 556K, built in 12 seconds
```

Just so we can get a server running quickly and see some results, let's run the server in a terminal, in the foreground. This is good for debugging because it logs directly to the console to let you know what the server is doing internally. Instructions on installing a boot script to get the server running when you restart your machine will, of course, vary by platform. Setting that up is an exercise left to the reader.

We're just going to use the default settings for the server, so starting it is simply a matter of executing `redis-server`, as in Example 8-2.

Example 8-2. Starting the server

```
$ redis-server
[91688] 25 Jul 09:37:36 # Warning: no config file specified, using the default config.
  In order to specify a config file use 'redis-server /path/to/redis.conf'
[91688] 25 Jul 09:37:36 * Server started, Redis version 2.4.15
[91688] 25 Jul 09:37:36 * The server is now ready to accept connections on port 6379
[91688] 25 Jul 09:37:36 - 0 clients connected (0 slaves), 922304 bytes in use
```

Using the Redis Shell

Redis comes with a very useful command-line shell that you can use interactively or from batch jobs. We'll just be using the interactive part of the shell so we can poke around inside the server, look at our data, and interact with it. The command shell has an extensive help system (Example 8-3) so once you're in there, hit the Tab key a couple of times to have the shell prompt you for help.

Example 8-3. Interacting with the Redis server

```
$ redis-cli
redis 127.0.0.1:6379> help
redis-cli 2.4.15
Type: "help @<group>" to get a list of commands in <group>
      "help <command>" for help on <command>
      "help <tab>" to get a list of possible help topics
      "quit" to exit
redis 127.0.0.1:6379> |
```

The Redis documentation is quite helpful here, as it gives a nice overview of all the commands available and shows you some example usage. Keep this page handy because you'll be referring back to it often.

It will pay dividends to spend some time familiarizing yourself with the basic SET and GET commands. Let's take a moment and play with inserting and retrieving data (Example 8-4).

Example 8-4. SET and GET data in Redis

```
$ redis-cli
redis 127.0.0.1:6379> keys *
(empty list or set)
redis 127.0.0.1:6379> set spring-data-book:redis:test-value 1
OK
redis 127.0.0.1:6379> keys *
1) "spring-data-book:redis:test-value"
redis 127.0.0.1:6379> get spring-data-book:redis:test-value
"1"
redis 127.0.0.1:6379> |
```

Notice that we didn't put quotes around the value `1` when we SET it. Redis doesn't have datatypes like other datastores, so it sees every value as a list of bytes. In the command shell, you'll see these printed as strings. When we GET the value back out, we see `"1"` in the command shell. We know by the quotes, then, that this is a string.

Connecting to Redis

Spring Data Redis supports connecting to Redis using either the Jedis, JRedis, RJC, or SRP driver libraries. Which you choose doesn't make any difference to your use of the Spring Data Redis library. The differences between the drivers have been abstracted out into a common set of APIs and template-style helpers. For the sake of simplicity, the example project uses the Jedis driver.

To connect to Redis using Jedis, you need to create an instance of `org.springframework.data.redis.connection.jedis.JedisConnectionFactory`. The other driver libraries have corresponding `ConnectionFactory` subclasses. A configuration using JavaConfig might look like Example 8-5.

Example 8-5. Connecting to Redis with Jedis

```
@Configuration
public class ApplicationConfig {

  @Bean
  public JedisConnectionFactory connectionFactory() {
    JedisConnectionFactory connectionFactory = new JedisConnectionFactory();
    connectionFactory.setHostName("localhost");
    connectionFactory.setPort(6379);
    return connectionFactory;
  }
}
```

The central abstraction you're likely to use when accessing Redis via Spring Data Redis is the `org.springframework.data.redis.core.RedisTemplate` class. Since the feature set of Redis is really too large to effectively encapsulate into a single class, the various operations on data are split up into separate `Operations` classes as follows (names are self-explanatory):

- `ValueOperations`
- `ListOperations`
- `SetOperations`
- `ZSetOperations`
- `HashOperations`
- `BoundValueOperations`
- `BoundListOperations`
- `BoundSetOperations`
- `BoundZSetOperations`
- `BoundHashOperations`

Object Conversion

Because Redis deals directly with byte arrays and doesn't natively perform `Object` to `byte[]` translation, the Spring Data Redis project provides some helper classes to make it easier to read and write data from Java code. By default, all keys and values are stored as serialized Java objects. If you're going to be dealing largely with `String`s, though, there is a template class—`StringRedisTemplate`, shown in Example 8-6—that installs the `String` serializer and has the added benefit of making your keys and values human-readable from the Redis command-line interface.

Example 8-6. Using the StringRedisTemplate

```
@Configuration
public class ApplicationConfig {
```

```
  @Bean
  public JedisConnectionFactory connectionFactory() { … }

  @Bean
  public StringRedisTemplate redisTemplate() {
    StringRedisTemplate redisTemplate = new StringRedisTemplate();
    redisTemplate.setConnectionFactory(connectionFactory());
    return redisTemplate;
  }
}
```

To influence how keys and values are serialized and deserialized, Spring Data Redis provides a `RedisSerializer` abstraction that is responsible for actually reading and writing the bytes stored in Redis. Set an instance of `org.springframework.data.redis.serializer.RedisSerializer` on either the `keySerializer` or `valueSerializer` property of the template. There is already a built-in `RedisSerializer` for `String`s, so to use `String`s for keys and `Long`s for values, you would create a simple serializer for `Long`s, as shown in Example 8-7.

Example 8-7. Creating reusable serializers

```
public enum LongSerializer implements RedisSerializer<Long> {

  INSTANCE;

  @Override
  public byte[] serialize(Long aLong) throws SerializationException {
    if (null != aLong) {
      return aLong.toString().getBytes();
    } else {
      return new byte[0];
    }
  }

  @Override
  public Long deserialize(byte[] bytes) throws SerializationException {
    if (bytes.length > 0) {
      return Long.parseLong(new String(bytes));
    } else {
      return null;
    }
  }
}
```

To use these serializers to make it easy to do type conversion when working with Redis, set the `keySerializer` and `valueSerializer` properties of the template like in the snippet of JavaConfig code shown in Example 8-8.

Example 8-8. Using serializers in a template instance

```
@Bean
public RedisTemplate<String, Long> longTemplate() {
```

```
  private static final StringRedisSerializer STRING_SERIALIZER =
    new StringRedisSerializer();

  RedisTemplate<String, Long> tmpl = new RedisTemplate<String, Long>();
  tmpl.setConnectionFactory(connFac);
  tmpl.setKeySerializer(STRING_SERIALIZER);
  tmpl.setValueSerializer(LongSerializer.INSTANCE);

  return tmpl;
}
```

You're now ready to start storing counts in Redis without worrying about `byte[]`-to-`Long` conversion. Since Redis supports such a large number of operations—which makes for a lot of methods on the helper classes—the methods for getting and setting values are defined in the various `RedisOperations` interfaces. You can access each of these interfaces by calling the appropriate `opsForX` method on the `RedisTemplate`. Since we're only storing discrete values in this example, we'll be using the `ValueOperations` template (Example 8-9).

Example 8-9. Automatic type conversion when setting and getting values

```
public class ProductCountTracker {

  @Autowired
  RedisTemplate<String, Long> redis;

  public void updateTotalProductCount(Product p) {
    // Use a namespaced Redis key
    String productCountKey = "product-counts:" + p.getId();

    // Get the helper for getting and setting values
    ValueOperations<String, Long> values = redis.opsForValue();

    // Initialize the count if not present
    values.setIfAbsent(productCountKey, 0L);

    // Increment the value by 1
    Long totalOfProductInAllCarts = values.increment(productCountKey, 1);
  }

}
```

After you call this method from your application and pass a `Product` with an `id` of `1`, you should be able to inspect the value from `redis-cli` and see the string `"1"` by issuing the Redis command `get product-counts:1`.

Object Mapping

It's great to be able to store simple values like counters and strings in Redis, but it's often necessary to store richer sets of related information. In some cases, these might be properties of an object. In other cases, they might be the keys and values of a hash.

Using the `RedisSerializer`, you can store an object into Redis as a single value. But doing so won't make the properties of that object very easy to inspect or retrieve individually. What you probably want in that case is to use a Redis hash. Storing your properties in a hash lets you access all of those properties together by pulling them all out as a `Map<String, String>`, or you can reference the individual properties in the hash without touching the others.

Since everything in Redis is a `byte[]`, for this hash example we're going to simpify by using `String`s for keys and values. The operations for hashes, like those for values, sets, and so on, are accessible from the `RedisTemplate opsForHash()` method. See Example 8-10.

Example 8-10. Using the HashOperations interface

```
private static final RedisSerializer<String> STRING_SERIALIZER =
  new StringRedisSerializer();

public void updateTotalProductCount(Product p) {

  RedisTemplate tmpl = new RedisTemplate();
  tmpl.setConnectionFactory(connectionFactory);
  // Use the standard String serializer for all keys and values
  tmpl.setKeySerializer(STRING_SERIALIZER);
  tmpl.setHashKeySerializer(STRING_SERIALIZER);
  tmpl.setHashValueSerializer(STRING_SERIALIZER);

  HashOperations<String, String, String> hashOps = tmpl.opsForHash();

  // Access the attributes for the Product
  String productAttrsKey = "products:attrs:" + p.getId();

  Map<String, String> attrs = new HashMap<String, String>();

  // Fill attributes
  attrs.put("name", "iPad");
  attrs.put("deviceType", "tablet");
  attrs.put("color", "black");
  attrs.put("price", "499.00");

  hashOps.putAll(productAttrsKey, attrs);

}
```

Assuming the `Product` has an `id` of `1`, from `redis-cli` you should be able to list all the keys of the hash by using the `HKEYS` command (Example 8-11).

Example 8-11. Listing hash keys

```
redis 127.0.0.1:6379> hkeys products:attrs:1
1) "price"
2) "color"
3) "deviceType"
4) "name"
```

```
redis 127.0.0.1:6379> hget products:attrs:1 name
"iPad"
```

Though this example just uses a `String` for the hash's value, you can use any `RedisSerializer` instance for the template's `hashValueSerializer`. If you wanted to store complex objects rather than `String`s, for instance, you might replace the `hashValueSerializer` in the template with an instance of `org.springframework.data.redis.serializer.JacksonJsonRedisSerializer` for serializing objects to JSON, or `org.springframework.data.redis.serializer.OxmSerializer` for marshalling and unmarshalling your object using Spring OXM.

Atomic Counters

Many people choose to use Redis because of the atomic counters that it supports. If multiple applications are all pointing at the same Redis instance, then those distributed applications can consistently and atomically increment a counter to ensure uniqueness. Java already contains `AtomicInteger` and `AtomicLong` classes for atomically incrementing counters across threads, but that won't help us if those counters are in other JVM processes or `ClassLoader`s. Spring Data Redis implements a couple of helper classes similar to `AtomicInteger` and `AtomicLong` and backs them by a Redis instance. Accessing distributed counters within your application is as easy as creating an instance of these helper classes and pointing them all to the same Redis server (Example 8-12).

Example 8-12. Using RedisAtomicLong

```
public class CountTracker {

  @Autowired
  RedisConnectionFactory connectionFactory;

  public void updateProductCount(Product p) {
    // Use a namespaced Redis key
    String productCountKey = "product-counts:" + p.getId();

    // Create a distributed counter.
    // Initialize it to zero if it doesn't yet exist
    RedisAtomicLong productCount =
      new RedisAtomicLong(productCountKey, connectionFactory, 0);

    // Increment the count
    Long newVal = productCount.incrementAndGet();
  }

}
```

Pub/Sub Functionality

Another important benefit of using Redis is the simple and fast publish/subscribe functionality. Although it doesn't have the advanced features of a full-blown message broker, Redis' pub/sub capability can be used to create a lightweight and flexible event bus. Spring Data Redis exposes a couple of helper classes that make working with this functionality extremely easy.

Listening and Responding to Messages

Following the pattern of the JMS `MessageListenerAdapter`, Spring Data Redis has a `MessageListenerAdapter` abstraction that works in basically the same way (Example 8-13). The JMS version, the `MessageListenerAdapter`, is flexible in what kind of listeners it accepts if you don't want to be tied to a particular interface. You can pass a POJO with a `handleMessage` method that takes as its first argument an `org.springframework.data.redis.connection.Message`, a `String`, a `byte[]`, or, if you use an appropriate `RedisSerializer`, an object of any convertible type. You can define an optional second parameter, which will be the channel or pattern that triggered this invocation. There is also a `MessageListener` interface to give your beans a solid contract to implement if you want to avoid the reflection-based invocation that's done when passing in a POJO.

Example 8-13. Adding a simple MessageListener using JavaConfig

```
@Bean public MessageListener dumpToConsoleListener() {
  return new MessageListener() {
    @Override
    public void onMessage(Message message, byte[] pattern) {
      System.out.println("FROM MESSAGE: " + new String(message.getBody()));
    }
  };
}
```

Spring Data Redis allows you to place POJOs on the `MessageListenerAdapter`, and the container will convert the incoming message into your custom type using a converter you provide. (See Example 8-14.)

Example 8-14. Setting up a MessageListenerContainer and simple message listener using a POJO

```
@Bean MessageListenerAdapter beanMessageListener() {
  MessageListenerAdapter listener = new MessageListenerAdapter( new BeanMessageListener()
);
  listener.setSerializer( new BeanMessageSerializer() );
  return listener;
}
```

`BeanMessageListener`, shown in Example 8-15, is simply a POJO with a method named `handleMessage` defined on it, with the first parameter being of type `BeanMessage` (an arbitrary class we've created for this example). It has a single property on it called `message`. Our `RedisSerializer` will store the contents of this `String` as bytes.

Example 8-15. Adding a POJO listener using JavaConfig

```
public class BeanMessageListener {
  public void handleMessage( BeanMessage msg ) {
    System.out.println( "msg: " + msg.message );
  }
}
```

The component responsible for actually invoking your listeners when the event is triggered is an `org.springframework.data.redis.listener.RedisMessageListenerContainer`. As demonstrated in Example 8-16, it needs to be configured with a `RedisConnectionFactory` and a set of listeners. The container has life cycle methods on it that will be called by the Spring container if you create it inside an `ApplicationContext`. If you create this container programmatically, you'll need to call the `afterPropertiesSet()` and `start()` methods manually. Remember to assign your listeners before you call the `start()` method, though, or your handlers will not be invoked since the wiring is done in the `start()` method.

Example 8-16. Configuring a RedisMessageListenerContainer

```
@Bean RedisMessageListenerContainer container() {
  RedisMessageListenerContainer container = new RedisMessageListenerContainer();
  container.setConnectionFactory(redisConnectionFactory());
  // Assign our BeanMessageListener to a specific channel
  container.addMessageListener(beanMessageListener(),
    new ChannelTopic("spring-data-book:pubsub-test:dump"));
  return container;
}
```

Using Spring's Cache Abstraction with Redis

Spring 3.1 introduced a common and reusable caching abstraction. This makes it easy to cache the results of method calls in your POJOs without having to explicitly manage the process of checking for the existence of a cache entry, loading new ones, and expiring old cache entries. Spring 3.1 gives you some helpers that work with a variety of cache backends to perform these functions for you.

Spring Data Redis supports this generic caching abstraction with the `org.springframework.data.redis.cache.RedisCacheManager`. To designate Redis as the backend for using the caching annotations in Spring, you just need to define a `RedisCacheManager` bean in your `ApplicationContext`. Then annotate your POJOs like you normally would, with `@Cacheable` on methods you want cached.

The `RedisCacheManager` needs a configured `RedisTemplate` in its constructor. In this example, we're letting the caching abstraction generate a unique integer for us to serve as the cache key. There are lots of options for how the cache manager stores your results. You can configure this behavior by placing the appropriate annotation on your `@Cacheable` methods. In Example 8-17, we're using an integer serializer for the key and the built-in `JdkSerializationRedisSerializer` for the value, since we really don't know

what we'll be storing. Using JDK serialization allows us to cache any `Serializable` Java object.

To enable the caching interceptor in your `ApplicationContext` using JavaConfig, you simply put the `@EnableCaching` annotation on your `@Configuration`.

Example 8-17. Configuring caching with RedisCacheManager

```
@Configuration
@EnableCaching
public class CachingConfig extends ApplicationConfig {

  @SuppressWarnings({"unchecked"})
  @Bean public RedisCacheManager redisCacheManager() {
    RedisTemplate tmpl = new RedisTemplate();
    tmpl.setConnectionFactory( redisConnectionFactory() );
    tmpl.setKeySerializer( IntSerializer.INSTANCE );
    tmpl.setValueSerializer( new JdkSerializationRedisSerializer() );
    RedisCacheManager cacheMgr = new RedisCacheManager( tmpl );
    return cacheMgr;
  }

  @Bean public CacheableTest cacheableTest() {
    return new CacheableTest();
  }

}
```