inspect classes in the *src/main/java* folder. To generate metamodel classes for classes in the test sources (*src/test/java*), add an execution of the `test-process` goal to the `generate-test-sources` phase.

Example 3-8. Setting up the Maven APT plug-in

```
<project …>
  <build>
    <plugins>
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <version>1.0.2</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <phase>generate-sources</phase>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor><!-- fully-qualified processor class name --></processor>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Supported Annotation Processors

Querydsl ships with a variety of APT processors to inspect different sets of annotations and generate the metamodel classes accordingly.

`QuerydslAnnotationProcessor`
: The very core annotation processor inspects Querydsl-specific annotations like `@QueryEntity` and `@QueryEmbeddable`.

`JPAAnnotationProcessor`
: Inspects `javax.persistence` annotations, such as `@Entity` and `@Embeddable`.

`HibernateAnnotationProcessor`
: Similar to the JPA processor but adds support for Hibernate-specific annotations.

`JDOAnnotationProcessor`
: Inspects JDO annotations, such as `@PersistenceCapable` and `@EmbeddedOnly`.

`MongoAnnotationProcessor`
: A Spring Data–specific processor inspecting the `@Document` annotation. Read more on this in "The Mapping Subsystem" on page 83.

Querying Stores Using Querydsl

Now that we have the query classes in place, let's have a look at how we can use them to actually build queries for a particular store. As already mentioned, Querydsl provides integration modules for a variety of stores that offer a nice and consistent API to create query objects, apply predicates defined via the generated query metamodel classes, and eventually execute the queries.

The JPA module, for example, provides a `JPAQuery` implementation class that takes an `EntityManager` and provides an API to apply predicates before execution; see Example 3-9.

Example 3-9. Using Querydsl JPA module to query a relational store

```
EntityManager entityManager = … // obtain EntityManager
JPAQuery query = new JPAQuery(entityManager);

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

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

If you remember Example 3-6, this code snippet doesn't look very different. In fact, the only difference here is that we use the `JPAQuery` as the base, whereas the former example used the collection wrapper. So you probably won't be too surprised to see that there's not much difference in implementing this scenario for a MongoDB store (Example 3-10).

Example 3-10. Using Querydsl MongoDB module with Spring Data MongoDB

```
MongoOperations operations = … // obtain MongoOperations
MongodbQuery query = new SpringDataMongodbQuery(operations, Product.class);

QProduct $ = QProduct.product;
List<Product> result = query.where($.description.contains("Apple").list();

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

Integration with Spring Data Repositories

As you just saw, the execution of queries with Querydsl generally consists of three major steps:

1. Setting up a store-specific query instance
2. Applying a set of filter predicates to it
3. Executing the query instance, potentially applying projections to it

Two of these steps can be considered boilerplate, as they will usually result in very similar code being written. On the other hand, the Spring Data repository tries to help users reduce the amount of unnecessary code; thus, it makes sense to integrate the repository extraction with Querydsl.

Executing Predicates

The core of the integration is the `QueryDslPredicateExecutor` interface ,which specifies the API that clients can use to execute Querydsl predicates in the flavor of the CRUD methods provided through `CrudRepository`. See Example 3-11.

Example 3-11. The QueryDslPredicateExecutor interface

```
public interface QueryDslPredicateExecutor<T> {

  T findOne(Predicate predicate);

  Iterable<T> findAll(Predicate predicate);
  Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);

  Page<T> findAll(Predicate predicate, Pageable pageable);
  long count(Predicate predicate);
}
```

Currently, Spring Data JPA and MongoDB support this API by providing implementation classes implementing the QueryDslPredicateExecutor interface shown in Example 3-11. To expose this API through your repository interfaces, let it extend `QueryDslPredicateExecutor` in addition to `Repository` or any of the other available base interfaces (see Example 3-12).

Example 3-12. The CustomerRepository interface extending QueryDslPredicateExecutor

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

Extending the interface will have two important results: the first—and probably most obvious—is that it pulls in the API and thus exposes it to clients of `CustomerRepository`. Second, the Spring Data repository infrastructure will inspect each repository interface found to determine whether it extends `QueryDslPredicateExecutor`. If it does and Querydsl is present on the classpath, Spring Data will select a special base class to back the repository proxy that generically implements the API methods by creating a store-specific query instance, bind the given predicates, potentially apply pagination, and eventually execute the query.

Manually Implementing Repositories

The approach we have just seen solves the problem of generically executing queries for the domain class managed by the repository. However, you cannot execute updates or deletes through this mechanism or manipulate the store-specific query instance. This is actually a scenario that plays nicely into the feature of *repository abstraction*, which allows you to selectively implement methods that need hand-crafted code (see "Manually Implementing Repository Methods" on page 21 for general details on that topic). To ease the implementation of a custom repository extension, we provide store-specific base classes. For details on that, check out the sections "Repository Querydsl Integration" on page 51 and "Mongo Querydsl Integration" on page 99.

PART II

Relational Databases

CHAPTER 4
JPA Repositories

The Java Persistence API (JPA) is the standard way of persisting Java objects into relational databases. The JPA consists of two parts: a mapping subsystem to map classes onto relational tables as well as an `EntityManager` API to access the objects, define and execute queries, and more. JPA abstracts a variety of implementations such as Hibernate, EclipseLink, OpenJpa, and others. The Spring Framework has always offered sophisticated support for JPA to ease repository implementations. The support consists of helper classes to set up an `EntityManagerFactory`, integrate with the Spring transaction abstraction, and translate JPA-specific exceptions into Spring's `DataAccessException` hierarchy.

The Spring Data JPA module implements the Spring Data Commons repository abstraction to ease the repository implementations even more, making a manual implementation of a repository obsolete in most cases. For a general introduction to the repository abstraction, see Chapter 2. This chapter will take you on a guided tour through the general setup and features of the module.

The Sample Project

Our sample project for this chapter consists of three packages: the *com.oreilly.springdata.jpa* base package plus a *core* and an *order* subpackage. The base package contains a Spring JavaConfig class to configure the Spring container using a plain Java class instead of XML. The two other packages contain our domain classes and repository interfaces. As the name suggests, the core package contains the very basic abstractions of the domain model: technical helper classes like `AbstractEntity`, but also domain concepts like an `EmailAddress`, an `Address`, a `Customer`, and a `Product`. Next, we have the *orders* package, which implements actual order concepts built on top of the foundational ones. So we'll find the `Order` and its `LineItem`s here. We will have a closer look at each of these classes in the following paragraphs, outlining their purpose and the way they are mapped onto the database using JPA mapping annotations.

The very core base class of all entities in our domain model is `AbstractEntity` (see Example 4-1). It's annotated with `@MappedSuperclass` to express that it is not a managed entity class on its own but rather will be extended by entity classes. We declare an `id` of type `Long` here and instruct the persistence provider to automatically select the most appropriate strategy for autogeneration of primary keys. Beyond that, we implement `equals(…)` and `hashCode()` by inspecting the `id` property so that entity classes of the same type with the same `id` are considered equal. This class contains the main technical artifacts to persist an entity so that we can concentrate on the actual domain properties in the concrete entity classes.

Example 4-1. The AbstractEntity class

```
@MappedSuperclass
public class AbstractEntity {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  @Override
  public boolean equals(Object obj) { … }

  @Override
  public int hashCode() { … }
}
```

Let's proceed with the very simple `Address` domain class. As Example 5-2 shows, it is a plain `@Entity` annotated class and simply consists of three `String` properties. Because they're all basic ones, no additional annotations are needed, and the persistence provider will automatically map them into table columns. If there were demand to customize the names of the columns to which the properties would be persisted, you could use the `@Column` annotation.

Example 4-2. The Address domain class

```
@Entity
public class Address extends AbstractEntity {

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

The `Address`es are referred to by the `Customer` entity. `Customer` contains quite a few other properties (e.g., the primitive ones `firstname` and `lastname`). They are mapped just like the properties of `Address` that we have just seen. Every `Customer` also has an email address represented through the `EmailAddress` class (see Example 4-3).

Example 4-3. The EmailAddress domain class

```
@Embeddable
public class EmailAddress {
```

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

  @Column(name = "email")
  private String emailAddress;

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

  protected EmailAddress() { }

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

This class is a value object, as defined in Eric Evans's book *Domain Driven Design* [Evans03]. Value objects are usually used to express domain concepts that you would naively implement as a primitive type (a `string` in this case) but that allow implementing domain constraints inside the value object. Email addresses have to adhere to a specific format; otherwise, they are not valid email addresses. So we actually implement the format check through some regular expression and thus prevent an `EmailAddress` instance from being instantiated if it's invalid.

This means that we can be sure to have a valid email address if we deal with an instance of that type, and we don't have to have some component validate it for us. In terms of persistence mapping, the `EmailAddress` class is an `@Embeddable`, which will cause the persistence provider to flatten out all properties of it into the table of the surrounding class. In our case, it's just a single column for which we define a custom name: `email`.

As you can see, we need to provide an empty constructor for the JPA persistence provider to be able to instantiate `EmailAddress` objects via reflection (Example 5-4). This is a significant shortcoming because you effectively cannot make the `emailAddress` a final one or assert make sure it is not `null`. The Spring Data mapping subsystem used for the NoSQL store implementations does not impose that need onto the developer. Have a look at "The Mapping Subsystem" on page 83 to see how a stricter implementation of the value object can be modeled in MongoDB, for example.

Example 4-4. The Customer domain class

```
@Entity
public class Customer extends AbstractEntity{

  private String firstname, lastname;

  @Column(unique = true)
  private EmailAddress emailAddress;
```

```
  @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
  @JoinColumn(name = "customer_id")
  private Set<Address> addresses;
}
```

We use the `@Column` annotation on the email address to make sure a single email address cannot be used by multiple customers so that we are able to look up customers uniquely by their email address. Finally we declare the `Customer` having a set of `Address`es. This property deserves deeper attention, as there are quite a few things we define here.

First, and in general, we use the `@OneToMany` annotation to specify that one `Customer` can have multiple `Address`es. Inside this annotation, we set the cascade type to `CascadeType.ALL` and also activate orphan removal for the addresses. This has a few consequences. For example, whenever we initially persist, update, or delete a customer, the `Address`es will be persisted, updated, or deleted as well. Thus, we don't have to persist an `Address` instance up front or take care of removing all `Address`es whenever we delete a `Customer`; the persistence provider will take care of that. Note that this is not a database-level cascade but rather a cascade managed by your JPA persistence provider. Beyond that, setting the orphan removal flag to `true` will take care of deleting `Address`es from that database if they are removed from the collection.

All this results in the `Address` life cycle being controlled by the `Customer`, which makes the relationship a classical composition. Plus, in domain-driven design (DDD) terminology, the `Customer` qualifies as aggregate root because it controls persistence operations and constraints for itself as well as other entities. Finally, we use `@JoinColumn` with the `addresses` property, which causes the persistence provider to add another column to the table backing the `Address` object. This additional column will then be used to refer to the `Customer` to allow joining the tables. If we had left out the additional annotation, the persistence provider would have created a dedicated join table.

The final piece of our core package is the `Product` (Example 4-5). Just as with the other classes discussed, it contains a variety of basic properties, so we don't need to add annotations to get them mapped by the persistence provider. We add only the `@Column` annotation to define the name and price as mandatory properties. Beyond that, we add a `Map` to store additional attributes that might differ from `Product` to `Product`.

Example 4-5. The Product domain class

```
@Entity
public class Product extends AbstractEntity {

  @Column(nullable = false)
  private String name;
  private String description;

  @Column(nullable = false)
  private BigDecimal price;
```

```
  @ElementCollection
  private Map<String, String> attributes = new HashMap<String, String>();
}
```

Now we have everything in place to build a basic customer relation management (CRM) or inventory system. Next, we're going to add abstractions that allow us to implement orders for Products held in the system. First, we introduce a LineItem that captures a reference to a Product alongside the amount of products as well as the price at which the product was bought. We map the Product property using a @ManyToOne annotation that will actually be turned into a product_id column in the LineItem table pointing to the Product (see Example 4-6).

Example 4-6. The LineItem domain class

```
@Entity
public class LineItem extends AbstractEntity {

  @ManyToOne
  private Product product;

  @Column(nullable = false)
  private BigDecimal price;
  private int amount;
}
```

The final piece to complete the jigsaw puzzle is the Order entity, which is basically a pointer to a Customer, a shipping Address, a billing Address, and the LineItems actually ordered (Example 4-7). The mapping of the line items is the very same as we already saw with Customer and Address. The Order will automatically cascade persistence operations to the LineItem instances. Thus, we don't have to manage the persistence life cycle of the LineItems separately. All other properties are many-to-one relationships to concepts already introduced. Note that we define a custom table name to be used for Orders because Order itself is a reserved keyword in most databases; thus, the generated SQL to create the table as well as all SQL generated for queries and data manipulation would cause exceptions when executing.

Example 4-7. The Order domain class

```
@Entity
@Table(name = "Orders")
public class Order extends AbstractEntity {

  @ManyToOne(optional = false)
  private Customer customer;
  @ManyToOne
  private Address billingAddress;

  @ManyToOne(optional = false, cascade = CascadeType.ALL)
  private Address shippingAddress;
```

```
  @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
  @JoinColumn(name = "order_id")
  private Set<LineItem>;

  …

  public Order(Customer customer, Address shippingAddress,
    Address billingAddress) {

    Assert.notNull(customer);
    Assert.notNull(shippingAddress);

    this.customer = customer;
    this.shippingAddress = shippingAddress.getCopy();
    this.billingAddress = billingAddress == null ? null :
      billingAddress.getCopy();
  }
}
```

A final aspect worth noting is that the constructor of the `Order` class does a defensive copy of the shipping and billing address. This is to ensure that changes to the `Address` instance handed into the method do not propagate into already existing orders. If we didn't create the copy, a customer changing her `Address` data later on would also change the `Address` on all of her `Order`s made to that `Address` as well.

The Traditional Approach

Before we start, let's look at how Spring Data helps us implement the data access layer for our domain model, and discuss how we'd implement the data access layer the traditional way. You'll find the sample implementation and client in the sample project annotated with additional annotations like `@Profile` (for the implementation) as well as `@ActiveProfile` (in the test case). This is because the Spring Data repositories approach will create an instance for the `CustomerRepository`, and we'll have one created for our manual implementation as well. Thus, we use the Spring profiles mechanism to bootstrap the traditional implementation for only the single test case. We don't show these annotations in the sample code because they would not have actually been used if you implemented the entire data access layer the traditional way.

To persist the previously shown entities using plain JPA, we now create an interface and implementation for our repositories, as shown in Example 4-8.

Example 4-8. Repository interface definition for Customers

```
public interface CustomerRepository {

  Customer save(Customer account);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

So we declare a method `save(…)` to be able to store accounts, and a query method to find all accounts that are assigned to a given customer by his email address. Let's see what an implementation of this repository would look like if we implemented it on top of plain JPA (Example 4-9).

Example 4-9. Traditional repository implementation for Customers

```
@Repository
@Transactional(readOnly = true)
class JpaCustomerRepository implements CustomerRepository {

  @PersistenceContext
  private EntityManager em;

  @Override
  @Transactional
  public Customer save(Customer customer) {

    if (customer.getId() == null) {
      em.persist(customer);
      return customer;
    } else {
      return em.merge(customer);
    }
  }

  @Override
  public Customer findByEmailAddress(EmailAddress emailAddress) {

    TypedQuery<Customer> query = em.createQuery(
      "select c from Customer c where c.emailAddress = :emailAddress", Customer.class);
    query.setParameter("emailAddress", emailAddress);

    return query.getSingleResult();
  }
}
```

The implementation class uses a JPA `EntityManager`, which will get injected by the Spring container due to the JPA `@PersistenceContext` annotation. The class is annotated with `@Repository` to enable exception translation from JPA exceptions to Spring's `DataAccessException` hierarchy. Beyond that, we use `@Transactional` to make sure the `save(…)` operation is running in a transaction and to allow setting the `readOnly` flag (at the class level) for `findByEmailAddress(…)`. This helps optimize performance inside the persistence provider as well as on the database level.

Because we want to free the clients from the decision of whether to call `merge(…)` or `persist(…)` on the `EntityManager`, we use the `id` field of the `Customer` to specify whether we consider a `Customer` object as new or not. This logic could, of course, be extracted into a common repository superclass, as we probably don't want to repeat this code for every domain object–specific repository implementation. The query method is quite straightforward as well: we create a query, bind a parameter, and execute the query to get a result. It's almost *so* straightforward that you could regard the implementation code as boilerplate. With a little bit of imagination, we can derive an implementation from the method signature: we expect a single customer, the query is quite close to the method name, and we simply bind the method parameter to it. So, as you can see, there's room for improvement.

Bootstrapping the Sample Code

We now have our application components in place, so let's get them up and running inside a Spring container. To do so, we have to do two things: first, we need to configure the general JPA infrastructure (i.e., a `DataSource` connecting to a database as well as a JPA `EntityManagerFactory`). For the former we will use HSQL, a database that supports being run in-memory. For the latter we will choose Hibernate as the persistence provider. You can find the dependency setup in the *pom.xml* file of the sample project. Second, we need to set up the Spring container to pick up our repository implementation and create a bean instance for it. In Example 4-10, you see a Spring JavaConfig configuration class that will achieve the steps just described.

Example 4-10. Spring JavaConfig configuration

```
@Configuration
@ComponentScan
@EnableTransactionManagement
class ApplicationConfig {

  @Bean
  public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.HSQL).build();
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setDatabase(Database.HSQL);
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory =
      new LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan(getClass().getPackage().getName());
    factory.setDataSource(dataSource());

    return factory;
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory());
```

```
    return txManager;
  }
}
```

The `@Configuration` annotation declares the class as a Spring JavaConfig configuration class. The `@ComponentScan` instructs Spring to scan the package of the `ApplicationConfig` class and all of its subpackages for Spring components (classes annotated with `@Service`, `@Repository`, etc.). `@EnableTransactionManagement` activates Spring-managed transactions at methods annotated with `@Transactional`.

The methods annotated with `@Bean` now declare the following infrastructure components: `dataSource()` sets up an embedded data source using Spring's embedded database support. This allows you to easily set up various in-memory databases for testing purposes with almost no configuration effort. We choose HSQL here (other options are H2 and Derby). On top of that, we configure an `EntityManagerFactory`. We use a new Spring 3.1 feature that allows us to completely abstain from creating a *persistence.xml* file to declare the entity classes. Instead, we use Spring's classpath scanning feature through the `packagesToScan` property of the `LocalContainerEntityManagerFactoryBean`. This will trigger Spring to scan for classes annotated with `@Entity` and `@MappedSuperclass` and automatically add those to the JPA `PersistenceUnit`.

The same configuration defined in XML looks something like Example 4-11.

Example 4-11. XML-based Spring configuration

```xml
<?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:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

  <context:componen-scan base-package="com.oreilly.springdata.jpa" />

  <tx:annotation-driven />

  <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>

  <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="com.oreilly.springdata.jpa" />
```

```
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="HSQL" />
        <property name="generateDdl" value="true" />
      </bean>
    </property>
  </bean>

  <jdbc:embedded-database id="dataSource" type="HSQL" />

</beans>
```

The `<jdbc:embedded-database />` at the very bottom of this example creates the in-memory `Datasource` using HSQL. The declaration of the `LocalContainerEntityManagerFactoryBean` is analogous to the declaration in code we've just seen in the JavaConfig case (Example 4-10). On top of that, we declare the `JpaTransactionManager` and finally activate annotation-based transaction configuration and component scanning for our base package. Note that the XML configuration in Example 4-11 is slightly different from the one you'll find in the *META-INF/spring/application-context.xml* file of the sample project. This is because the sample code is targeting the Spring Data JPA-based data access layer implementation, which renders some of the configuration just shown obsolete.

The sample application class creates an instance of an `AnnotationConfigApplicationContext`, which takes a Spring JavaConfig configuration class to bootstrap application components (Example 4-12). This will cause the infrastructure components declared in our `ApplicationConfig` configuration class and our annotated repository implementation to be discovered and instantiated. Thus, we can access a Spring bean of type `CustomerRepository`, create a customer, store it, and look it up by its email address.

Example 4-12. Bootstrapping the sample code

```
@RunWith(SpringJunit4ClassRunner.class)
@ContextConfiguration(classes = ApplicationConfig.class)
class CustomerRepositoryIntegrationTests {

  @Autowired
  CustomerRepository customerRepository;

  @Test
  public void savesAndFindsCustomerByEmailAddress {

    Customer dave = new Customer("Dave", "Matthews");
    dave.setEmailAddress("dave@dmband.com");

    Customer result = repository.save(dave);
    Assert.assertThat(result.getId(), is(nonNullValue()));

    result = repository.findByEmailAddress("dave@dmband.com");
    Assert.assertThat(result, is(dave));
  }
}
```

Using Spring Data Repositories

To enable the Spring data repositories, we must make the repository interfaces discoverable by the Spring Data repository infrastructure. We do so by letting our `CustomerRepository` extend the Spring Data Repository marker interface. Beyond that, we keep the declared persistence methods we already have. See Example 4-13.

Example 4-13. Spring Data CustomerRepository interface

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

  Customer save(Account account);

  Customer findByEmailAddress(String emailAddress);
}
```

The `save(…)` method will be backed by the generic `SimpleJpaRepository` class that implements all CRUD methods. The query method we declared will be backed by the generic query derivation mechanism, as described in "Query Derivation" on page 17. The only thing we now have to add to the Spring configuration is a way to activate the Spring Data repository infrastructure, which we can do in either XML or JavaConfig. For the JavaConfig way of configuration, all you need to do is add the `@EnableJpaRepositories` annotation to your configuration class. We remove the `@ComponentScan` annotation be removed for our sample because we don't need to look up the manual implementation anymore. The same applies to `@EnableTransactionManagement`. The Spring Data repository infrastructure will automatically take care of the method calls to repositories taking part in transactions. For more details on transaction configuration, see "Transactionality" on page 50. We'd probably still keep these annotations around were we building a more complete application. We remove them for now to prevent giving the impression that they are necessary for the sole data access setup. Finally, the header of the `ApplicationConfig` class looks something like Example 4-14.

Example 4-14. Enabling Spring Data repositories using JavaConfig

```
@Configuration
@EnableJpaRepositories
class ApplicationConfig {

  // … as seen before
}
```

If you're using XML configuration, add the `repositories` XML namespace element of the JPA namespace, as shown in Example 4-15.

Example 4-15. Activating JPA repositories through the XML namespace

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

To see this working, have a look at `CustomerRepositoryIntegrationTest`. It basically uses the Spring configuration set up in `AbstractIntegrationTest`, gets the `CustomerRepository` wired into the test case, and runs the very same tests we find in `JpaCustomerRepositoryIntegrationTest`, only without us having to provide any implementation class for the repository interface whatsoever. Let's look at the individual methods declared in the repository and recap what Spring Data JPA is actually doing for each one of them. See Example 4-16.

Example 4-16. Repository interface definition for Customers

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

  Customer findOne(Long);

  Customer save(Customer account);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

The `findOne(…)` and `save(…)` methods are actually backed by `SimpleJpaRepository`, which is the class of the instance that actually backs the proxy created by the Spring Data infrastructure. So, solely by matching the method signatures, the calls to these two methods get routed to the implementation class. If we wanted to expose a more complete set of CRUD methods, we might simply extend `CrudRepository` instead of `Repository`, as it contains these methods already. Note how we actually prevent `Customer` instances from being deleted by not exposing the `delete(…)` methods that would have been exposed if we had extended `CrudRepository`. Find out more about of the tuning options in "Fine-Tuning Repository Interfaces" on page 20.

The last method to discuss is `findByEmailAddress(…)`, which obviously is not a CRUD one but rather intended to be executed as a query. As we haven't manually declared any, the bootstrapping purpose of Spring Data JPA will inspect the method and try to derive a query from it. The derivation mechanism (details on that in "Query Derivation" on page 17) will discover that `EmailAddress` is a valid property reference for `Customer` and eventually create a JPA Criteria API query whose JPQL equivalent is `select c from Customer c where c.emailAddress = ?1`. Because the method returns a single `Customer`, the query execution expects the query to return at most one resulting entity. If no `Customer` is found, we'll get `null`; if there's more than one found, we'll see a `IncorrectResultSizeDataAccessException`.

Let's continue with the `ProductRepository` interface (Example 4-17). The first thing you note is that compared to `CustomerRepository`, we're extending `CrudRepository` first because we'd like to have the full set of CRUD methods available. The method `findByDescriptionContaining(…)` is clearly a query method. There are several things to note here. First, we not only reference the description property of the product, but also qualify the predicate with the `Containing` keyword. That will eventually lead to the given description parameter being surrounded by `%` characters, and the resulting `String` being bound via the LIKE operator. Thus, the query is as follows: select p from Product p where p.description like ?1 with a given description of Apple bound as %Apple%. The second interesting thing is that we're using the pagination abstraction to retrieve only a subset of the products matching the criteria. The lookupProductsByDescription() test case in ProductRepositoryIntegrationTest shows how that method can be used (Example 4-18).

Example 4-17. Repository interface definition for Products

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

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

  @Query("select p from Product p where p.attributes[?1] = ?2")
  List<Product> findByAttributeAndValue(String attribute, String value);
}
```

Example 4-18. Test case for ProductRepository findByDescriptionContaining(…)

```
@Test
public void lookupProductsByDescription() {

  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.getTotalElements(), is(2L));
  assertThat(page.isFirstPage(), is(true));
  assertThat(page.isLastPage(), is(false));
  assertThat(page.hasNextPage(), is(true));
}
```

We create a new PageRequest instance to ask for the very first page by specifying a page size of one with a descending order by the name of the Product. We then simply hand that Pageable into the method and make sure we've got the iPad back, that we're the first page, and that there are further pages available. As you can see, the execution of the paging method retrieves the necessary metadata to find out how many items the query would have returned if we hadn't applied pagination. Without Spring Data, reading that metadata would require manually coding the extra query execution, which does a count projection based on the actual query. For more detailed information on pagination with repository methods, see "Pagination and Sorting" on page 18.

The second method in ProductRepository is findByAttributeAndValue(…). We'd essentially like to look up all Products that have a custom attribute with a given value. Because the attributes are mapped as @ElementCollection (see Example 4-5 for reference), we unfortunately cannot use the query derivation mechanism to get the query created for us. To manually define the query to be executed, we use the @Query annotation. This also comes in handy if the queries get more complex in general. Even if they were derivable, they'd result in awfully verbose method names.

Finally, let's have a look at the `OrderRepository` (Example 4-19), which should already look remarkably familiar. The query method `findByCustomer(…)` will trigger query derivation (as shown before) and result in `select o from Order o where o.customer = ?1`. The only crucial difference from the other repositories is that we extend `PagingAndSortingRepository`, which in turn extends `CrudRepository`. `PagingAndSortingRepository` adds `findAll(…)` methods that take a `Sort` and `Pageable` parameter on top of what `CrudRepository` already provides. The main use case here is that we'd like to access all `Order`s page by page to avoid loading them all at once.

Example 4-19. Repository interface definition for Orders

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

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

Transactionality

Some of the CRUD operations that will be executed against the JPA `EntityManager` require a transaction to be active. To make using Spring Data Repositories for JPA as convenient as possible, the implementation class backing `CrudRepository` and `PagingAndSortingRepository` is equipped with `@Transactional` annotations with a default configuration to let it take part in Spring transactions automatically, or even trigger new ones in case none is already active. For a general introduction into Spring transactions, please consult the Spring reference documentation.

In case the repository implementation actually triggers the transaction, it will create a default one (store-default isolation level, no timeout configured, rollback for runtime exceptions only) for the `save(…)` and `delete(…)` operations and read-only ones for all find methods including the paged ones. Enabling read-only transactions for reading methods results in a few optimizations: first, the flag is handed to the underlying JDBC driver which will—depending on your database vendor—result in optimizations or the driver even preventing you from accidentally executing modifying queries. Beyond that, the Spring transaction infrastructure integrates with the life cycle of the `EntityManager` and can set the `FlushMode` for it to `MANUAL`, preventing it from checking each entity in the persistence context for changes (so-called *dirty checking*). Especially with a large set of objects loaded into the persistence context, this can lead to a significant improvement in performance.

If you'd like to fine-tune the transaction configuration for some of the CRUD methods (e.g., to configure a particular timeout), you can do so by redeclaring the desired CRUD method and adding `@Transactional` with your setup of choice to the method declaration. This will then take precedence over the default configuration declared in `SimpleJpaRepository`. See Example 4-20.

Example 4-20. Reconfiguring transactionality in CustomerRepository interface

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

  @Transactional(timeout = 60)
  Customer save(Customer account);
}
```

This, of course, also works if you use custom repository base interfaces; see "Fine-Tuning Repository Interfaces" on page 20.

Repository Querydsl Integration

Now that we've seen how to add query methods to repository interfaces, let's 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.

To generate the metamodel classes, we have configured the Querydsl Maven plug-in in our *pom.xml* file, as shown in Example 4-21.

Example 4-21. Setting up the Querydsl APT processor for JPA

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>maven-apt-plugin</artifactId>
  <version>1.0.4</version>
  <configuration>
    <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>
  </configuration>
  <executions>
    <execution>
      <id>sources</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>process</goal>
      </goals>
      <configuration>
        <outputDirectory>target/generated-sources</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The only JPA-specific thing to note here is the usage of the `JPAAnnotationProcessor`. It will cause the plug-in to consider JPA mapping annotations to discover entities, relationships to other entities, embeddables, etc. The generation will be run during the normal build process and classes generated into a folder under *target*. Thus, they will be cleaned up with each clean build, and don't get checked into the source control system.

If you're using Eclipse and add the plug-in to your project setup, you will have to trigger a Maven project update (right-click on the project and choose Maven→Update Project…). This will add the configured output directory as an additional source folder so that the code using the generated classes compiles cleanly.

Once this is in place, you should find the generated query classes `QCustomer`, `QProduct`, and so on. Let's explore the capabilities of the generated classes in the context of the `ProductRepository`. To be able to execute Querydsl predicates on the repository, we add the `QueryDslPredicateExecutor` interface to the list of extended types, as shown in Example 4-22.

Example 4-22. The ProductRepository interface extending QueryDslPredicateExecutor

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

This will pull methods like `findAll(Predicate predicate)` and `findOne(Predicate predicate)` into the API. We now have everything in place, so we can actually start using the generated classes. Let's have a look at the `QuerydslProductRepositoryIntegrationTest` (Example 4-23).

Example 4-23. 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));
```

First, we obtain a reference to the `QProduct` metamodel class and keep that inside the product property. We can now use this to navigate the generated path expressions to create predicates. We use a `product.name.eq("iPad")` to query for the `Product` named iPad and keep that as a reference. The second predicate we build specifies that we'd like to look up all products with a description containing `tablet`. We then go on executing the `Predicate` instance against the repository and assert that we found exactly the iPad we looked up for reference before.

You see that the definition of the predicates is remarkably readable and concise. The built predicates can be recombined to construct higher-level predicates and thus allow for querying flexibility without adding complexity.

CHAPTER 5

Type-Safe JDBC Programming with Querydsl SQL

Using JDBC is a popular choice for working with a relational database. Most of Spring's JDBC support is provided in the *spring-jdbc* module of the Spring Framework itself. A good guide for this JDBC support is *Just Spring Data Access* by Madhusudhan Konda [Konda12]. The Spring Data JDBC Extensions subproject of the Spring Data project does, however, provide some additional features that can be quite useful. That's what we will cover in this chapter. We will look at some recent developments around type-safe querying using Querydsl.

In addition to the Querydsl support, the Spring Data JDBC Extensions subproject contains some database-specific support like connection failover, message queuing, and improved stored procedure support for the Oracle database. These features are limited to the Oracle database and are not of general interest, so we won't be covering them in this book. The Spring Data JDBC Extensions subproject does come with a detailed reference guide that covers these features if you are interested in exploring them further.

The Sample Project and Setup

We have been using strings to define database queries in our Java programs for a long time, and as mentioned earlier this can be quite error-prone. Column or table names can change. We might add a column or change the type of an existing one. We are used to doing similar refactoring for our Java classes in our Java IDEs, and the IDE will guide us so we can find any references that need changing, including in comments and configuration files. No such support is available for strings containing complex SQL query expressions. To avoid this problem, we provide support for a type-safe query alternative in Querydsl. Many data access technologies integrate well with Querydsl, and Chapter 3 provided some background on it. In this section we will focus on the Querydsl SQL module and how it integrates with Spring's `JdbcTemplate` usage, which should be familiar to every Spring developer.

Before we look at the new JDBC support, however, we need to discuss some general concerns like database configuration and project build system setup.

The HyperSQL Database

We are using the HyperSQL database version 2.2.8 for our Querydsl examples in this chapter. One nice feature of HyperSQL is that we can run the database in both server mode and in-memory. The in-memory option is great for integration tests since starting and stopping the database can be controlled by the application configuration using Spring's `EmbeddedDatabaseBuilder`, or the `<jdbc:embedded-database>` tag when using the spring-jdbc XML namespace. The build scripts download the dependency and start the in-memory database automatically. To use the database in standalone server mode, we need to download the distribution and unzip it to a directory on our system. Once that is done, we can change to the *hsqldb* directory of the unzipped distribution and start the database using this command:

```
java -classpath lib/hsqldb.jar org.hsqldb.server.Server \
  --database.0 file:data/test --dbname.0 test
```

Running this command starts up the server, which generates some log output and a message that the server has started. We are also told we can use Ctrl-C to stop the server. We can now open another command window, and from the same *hsqldb* directory we can start up a database client so we can interact with the database (creating tables and running queries, etc.). For Windows, we need to execute only the *runManagerSwing.bat* batch file located in the *bin* directory. For OS X or Linux, we can run the following command:

```
java -classpath lib/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing
```

This should bring up the login dialog shown in Figure 5-1. We need to change the Type to HSQL Database Engine Server and add "test" as the name of the database to the URL so it reads `jdbc:hsqldb:hsql://localhost/test`. The default user is "sa" with a blank password. Once connected, we have an active GUI database client.

The SQL Module of Querydsl

The SQL module of Querydsl provides a type-safe option for the Java developer to work with relational databases. Instead of writing SQL queries and embedding them in strings in your Java program, Querydsl generates query types based on metadata from your database tables. You use these generated types to write your queries and perform CRUD operations against the database without having to resort to providing column or table names using strings.

The way you generate the query types is a bit different in the SQL module compared to other Querydsl modules. Instead of relying on annotations, the SQL module relies on the actual database tables and available JDBC metadata for generating the query

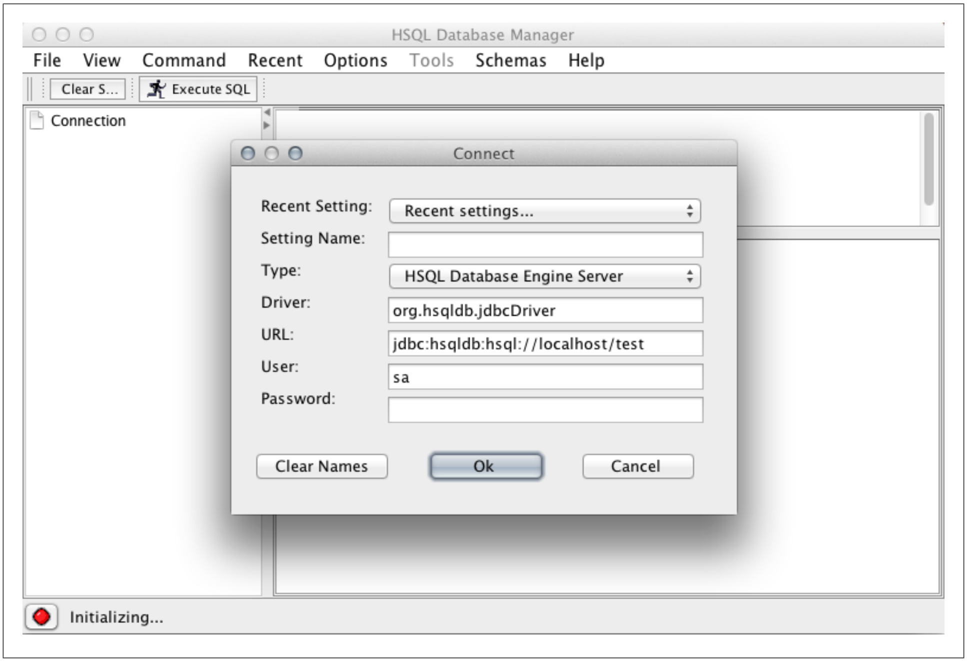

Figure 5-1. HSQLDB client login dialog

types. This means that you need to have the tables created and access to a live database before you run the query class generation. For this reason, we recommend running this as a separate step of the build and saving the generated classes as part of the project in the source control system. We need to rerun this step only when we have made some changes to our table structures and before we check in our code. We expect the continuous integration system to run this code generation step as well, so any mismatch between the Java types and the database tables would be detected at build time.

We'll take a look at what we need to generate the query types later, but first we need to understand what they contain and how we use them. They contain information that Querydsl can use to generate queries, but they also contain information you can use to compose queries; perform updates, inserts, and deletes; and map data to domain objects. Let's take a quick look at an example of a table to hold address information. The `address` table has three `VARCHAR` columns: `street`, `city`, and `country`. Example 5-1 shows the SQL statement to create this table.

Example 5-1. Creating the address table

```
CREATE TABLE address (
  id BIGINT IDENTITY PRIMARY KEY,
  customer_id BIGINT CONSTRAINT address_customer_ref
    FOREIGN KEY REFERENCES customer (id),
  street VARCHAR(255),
  city VARCHAR(255),
  country VARCHAR(255));
```

Example 5-2 demonstrates the generated query type based on this `address` table. It has some constructors, Querydsl path expressions for the columns, methods to create primary and foreign key types, and a static field that provides an instance of the `QAddress` class.

Example 5-2. A generated query type—QAddress

```
package com.oreilly.springdata.jdbc.domain;

import static com.mysema.query.types.PathMetadataFactory.*;

import com.mysema.query.types.*;
import com.mysema.query.types.path.*;

import javax.annotation.Generated;


/**
 * QAddress is a Querydsl query type for QAddress
 */
@Generated("com.mysema.query.sql.codegen.MetaDataSerializer")
public class QAddress extends com.mysema.query.sql.RelationalPathBase<QAddress> {

  private static final long serialVersionUID = 207732776;

  public static final QAddress address = new QAddress("ADDRESS");

  public final StringPath city = createString("CITY");
  public final StringPath country = createString("COUNTRY");
  public final NumberPath<Long> customerId = createNumber("CUSTOMER_ID", Long.class);
  public final NumberPath<Long> id = createNumber("ID", Long.class);
  public final StringPath street = createString("STREET");
  public final com.mysema.query.sql.PrimaryKey<QAddress> sysPk10055 = createPrimaryKey(id);
  public final com.mysema.query.sql.ForeignKey<QCustomer> addressCustomerRef =
      createForeignKey(customerId, "ID");

  public QAddress(String variable) {
    super(QAddress.class, forVariable(variable), "PUBLIC", "ADDRESS");
  }

  public QAddress(Path<? extends QAddress> entity) {
    super(entity.getType(), entity.getMetadata(), "PUBLIC", "ADDRESS");
  }

  public QAddress(PathMetadata<?> metadata) {
    super(QAddress.class, metadata, "PUBLIC", "ADDRESS");
  }
}
```

By creating a reference like this:

```
QAddress qAddress = QAddress.address;
```

in our Java code, we can reference the table and the columns more easily using `qAddress` instead of resorting to using string literals.

In Example 5-3, we query for the street, city, and country for any address that has London as the city.

Example 5-3. Using the generated query class

```
QAddress qAddress = QAddress.address;
SQLTemplates dialect = new HSQLDBTemplates();
SQLQuery query = new SQLQueryImpl(connection, dialect)
    .from(qAddress)
    .where(qAddress.city.eq("London"));
List<Address> results = query.list(
    new QBean<Address>(Address.class, qAddress.street,
      qAddress.city, qAddress.country));
```

First, we create a reference to the query type and an instance of the correct `SQLTemplates` for the database we are using, which in our case is `HSQLDBTemplates`. The `SQLTemplates` encapsulate the differences between databases and are similar to Hibernate's `Dialect`. Next, we create an `SQLQuery` with the JDBC `javax.sql.Connection` and the `SQLTemplates` as the parameters. We specify the table we are querying using the `from` method, passing in the query type. Next, we provide the where clause or predicate via the `where` method, using the `qAddress` reference to specify the criteria that `city` should equal `London`.

Executing the `SQLQuery`, we use the `list` method, which will return a `List` of results. We also provide a mapping implementation using a `QBean`, parameterized with the domain type and a projection consisting of the columns `street`, `city`, and `country`.

The result we get back is a `List` of `Address`es, populated by the `QBean`. The `QBean` is similar to Spring's `BeanPropertyRowMapper`, and it requires that the domain type follows the `JavaBean` style. Alternatively, you can use a `MappingProjection`, which is similar to Spring's familiar `RowMapper` in that you have more control over how the results are mapped to the domain object.

Based on this brief example, let's summarize the components of Querydsl that we used for our SQL query:

- The `SQLQueryImpl` class , which will hold the target table or tables along with the predicate or where clause and possibly a join expression if we are querying multiple tables
- The `Predicate`, usually in the form of a `BooleanExpression` that lets us specify filters on the results
- The mapping or results extractor, usually in the form of a `QBean` or `MappingProjection` parameterized with one or more `Expression`s as the projection

So far, we haven't integrated with any Spring features, but the rest of the chapter covers this integration. This first example is just intended to introduce the basics of the Querydsl SQL module.

Build System Integration

The code for the Querydsl part of this chapter is located in the *jdbc* module of the sample GitHub project.

Before we can really start using Querydsl in our project, we need to configure our build system so that we can generate the query types. Querydsl provides both Maven and Ant integration, documented in the "Querying SQL" chapter of the Querydsl reference documentation.

In our Maven *pom.xml* file, we add the plug-in configuration shown in Example 5-4.

Example 5-4. Setting up code generation Maven plug-in

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${querydsl.version}</version>
  <configuration>
    <jdbcDriver>org.hsqldb.jdbc.JDBCDriver</jdbcDriver>
    <jdbcUrl>jdbc:hsqldb:hsql://localhost:9001/test</jdbcUrl>
    <jdbcUser>sa</jdbcUser>
    <schemaPattern>PUBLIC</schemaPattern>
    <packageName>com.oreilly.springdata.jdbc.domain</packageName>
    <targetFolder>${project.basedir}/src/generated/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>2.2.8</version>
    </dependency>
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

We will have to execute this plug-in explicitly using the following Maven command:

```
mvn com.mysema.querydsl:querydsl-maven-plugin:export
```

You can set the plug-in to execute as part of the `generate-sources` life cycle phase by specifying an execution goal. We actually do this in the example project, and we also use a predefined HSQL database just to avoid forcing you to start up a live database when you build the example project. For real work, though, you do need to have a database where you can modify the schema and rerun the Querydsl code generation.

The Database Schema

Now that we have the build configured, we can generate the query classes, but let's first review the database schema that we will be using for this section. We already saw the `address` table, and we are now adding a `customer` table that has a one-to-many relationship with the `address` table. We define the schema for our HSQLDB database as shown in Example 5-5.

Example 5-5. schema.sql

```
CREATE TABLE customer (
  id BIGINT IDENTITY PRIMARY KEY,
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  email_address VARCHAR(255));

CREATE UNIQUE INDEX ix_customer_email ON CUSTOMER (email_address ASC);

CREATE TABLE address (
  id BIGINT IDENTITY PRIMARY KEY,
  customer_id BIGINT CONSTRAINT address_customer_ref FOREIGN KEY REFERENCES customer (id),
  street VARCHAR(255),
  city VARCHAR(255),
  country VARCHAR(255));
```

The two tables, `customer` and `address`, are linked by a foreign key reference from `address` to `customer`. We also define a unique index on the `email_address` column of the `address` table.

This gives us the domain model implementation shown in Figure 5-2.

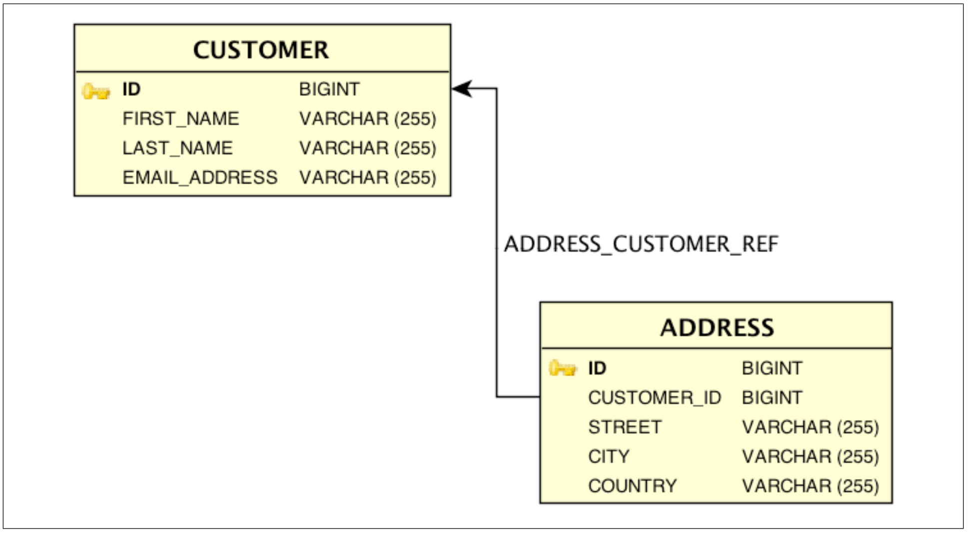

Figure 5-2. Domain model implementation used with Querydsl SQL

The Domain Implementation of the Sample Project

We have already seen the schema for the database, and now we will take a look at the corresponding Java domain classes we will be using for our examples. We need a `Customer` class plus an `Address` class to hold the data from our database tables. Both of these classes extend an `AbstractEntity` class that, in addition to `equals(…)` and `hashCode()`, has setters and getters for the `id`, which is a `Long`:

```
public class AbstractEntity {

  private Long id;

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  @Override
  public boolean equals(Object obj) { … }

  @Override
  public int hashCode() { … }
}
```

The `Customer` class has name and email information along with a set of addresses. This implementation is a traditional JavaBean with getters and setters for all properties:

```
public class Customer extends AbstractEntity {

  private String firstName;
  private String lastName;
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public EmailAddress getEmailAddress() {
    return emailAddress;
```

```
  }

  public void setEmailAddress(EmailAddress emailAddress) {
    this.emailAddress = emailAddress;
  }

  public Set<Address> getAddresses() {
    return Collections.unmodifiableSet(addresses);
  }

  public void addAddress(Address address) {
    this.addresses.add(address);
  }

  public void clearAddresses() {
    this.addresses.clear();
  }
}
```

The email address is stored as a VARCHAR column in the database, but in the Java class we use an EmailAddress value object type that also provides validation of the email address using a regular expression. This is the same class that we have seen in the other chapters:

```
public class EmailAddress {

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

  private String value;

  protected EmailAddress() {
  }

  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 last domain class is the Address class, again a traditional JavaBean with setters and getters for the address properties. In addition to the no-argument constructor, we have a constructor that takes all address properties:

```
public class Address extends AbstractEntity {
  private String street, city, country;

  public Address() {
  }
```

```
    public Address(String street, String city, String country) {
      this.street = street;
      this.city = city;
      this.country = country;
    }

    public String getCountry() {
      return country;
    }

    public void setCountry(String country) {
      this.country = country;
    }

    public String getStreet() {
      return street;
    }

    public void setStreet(String street) {
      this.street = street;
    }

    public String getCity() {
      return city;
    }

    public void setCity(String city) {
      this.city = city;
    }
  }
```

The preceding three classes make up our domain model and reside in the `com.oreilly.springdata.jdbc.domain` package of the JDBC example project. Now we are ready to look at the interface definition of our `CustomerRepository`:

```
public interface CustomerRepository {

  Customer findById(Long id);

  List<Customer> findAll();

  void save(Customer customer);

  void delete(Customer customer);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

We have a couple of finder methods and save and delete methods. We don't have any repository methods to save and delete the `Address` objects since they are always owned by the `Customer` instances. We will have to persist any addresses provided when the `Customer` instance is saved.

The QueryDslJdbcTemplate

The central class in the Spring Data integration with Querydsl is the QueryDslJdbcTemplate. It is a wrapper around a standard Spring JdbcTemplate that has methods for managing SQLQuery instances and executing queries as well as methods for executing inserts, updates, and deletes using command-specific callbacks. We'll cover all of these in this section, but let's start by creating a QueryDslJdbcTemplate.

To configure the QueryDslJdbcTemplate, you simply pass in either a DataSource:

```
QueryDslJdbcTemplate qdslTemplate = new QueryDslJdbcTemplate(dataSource);
```

or an already configured JdbcTemplate in the constructor:

```
jdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
QueryDslJdbcTemplate qdslTemplate = new QueryDslJdbcTemplate(jdbcTemplate);
```

Now we have a fully configured QueryDslJdbcTemplate to use. We saw earlier that usually you need to provide a Connection and an SQLTemplates matching your database when you create an SQLQuery object. However, when you use the QueryDslJdbcTemplate, there is no need to do this. In usual Spring fashion, the JDBC layer will manage any database resources like connections and result sets. It will also take care of providing the SQLTemplates instance based on database metadata from the managed connection. To obtain a managed instance of an SQLQuery, you use the newSqlQuery static factory method of the QueryDslJdbcTemplate:

```
SQLQuery sqlQuery = qdslTemplate.newSqlQuery();
```

The SQLQuery instance obtained does not yet have a live connection, so you need to use the query methods of the QueryDslJdbcTemplate to allow connection management to take place:

```
SQLQuery addressQuery = qdslTemplate.newSqlQuery()
  .from(qAddress)
  .where(qAddress.city.eq("London"));

List<Address> results = qdslTemplate.query(
  addressQuery,
  BeanPropertyRowMapper.newInstance(Address.class),
  qAddress.street, qAddress.city, qAddress.country);
```

There are two query methods: query returning a List and queryForObject returning a single result. Both of these have three overloaded versions, each taking the following parameters:

- SQLQuery object obtained via the newSqlQuery factory method
- One of the following combinations of a mapper and projection implementation:
  - —RowMapper, plus a projection expressed as one or more Expressions
  - —ResultSetExtractor, plus a projection expressed as one or more Expressions
  - —ExpressionBase, usually expressed as a QBean or MappingProjection

The first two mappers, RowMapper and ResultSetExtractor, are standard Spring interfaces often used with the regular JdbcTemplate. They are responsible for extracting the data from the results returned by a query. The ResultSetExtractor extracts data for all rows returned, while the RowMapper handles only one row at the time and will be called repeatedly, once for each row. QBean and MappingProjection are Querydsl classes that also map one row at the time. Which ones you use is entirely up to you; they all work equally well. For most of our examples, we will be using the Spring types—this book is called *Spring Data*, after all.

Executing Queries

Now we will look at how we can use the QueryDslJdbcTemplate to execute queries by examining how we should implement the query methods of our CustomerRepository.

The Beginning of the Repository Implementation

The implementation will be autowired with a DataSource; in that setter, we will create a QueryDslJdbcTemplate and a projection for the table columns used by all queries when retrieving data needed for the Customer instances. (See Example 5-6.)

Example 5-6. Setting up the QueryDslCustomerRepository instance

```
@Repository
@Transactional
public class QueryDslCustomerRepository implements CustomerRepository {

  private final QCustomer qCustomer = QCustomer.customer;
  private final QAddress qAddress = QAddress.address;

  private final QueryDslJdbcTemplate template;
  private final Path<?>[] customerAddressProjection;

  @Autowired
  public QueryDslCustomerRepository(DataSource dataSource) {

    Assert.notNull(dataSource);

    this.template = new QueryDslJdbcTemplate(dataSource);
    this.customerAddressProjection = new Path<?>[] { qCustomer.id, qCustomer.firstName,
      qCustomer.lastName, qCustomer.emailAddress, qAddress.id, qAddress.customerId,
      qAddress.street, qAddress.city, qAddress.country };
  }

  @Override
  @Transactional(readOnly = true)
  public Customer findById(Long id) { … }

  @Override
  @Transactional(readOnly = true)
  public Customer findByEmailAddress(EmailAddress emailAddress) { … }
```

```
  @Override
  public void save(final Customer customer) { … }

  @Override
  public void delete(final Customer customer) { … }
}
```

We are writing a repository, so we start off with an `@Repository` annotation. This is a standard Spring stereotype annotation, and it will make your component discoverable during classpath scanning. In addition, for repositories that use ORM-style data access technologies, it will also make your repository a candidate for exception translation between the ORM-specific exceptions and Spring's `DataAccessException` hierarchy. In our case, we are using a template-based approach, and the template itself will provide this exception translation.

Next is the `@Transactional` annotation. This is also a standard Spring annotation that indicates that any call to a method in this class should be wrapped in a database transaction. As long as we provide a transaction manager implementation as part of our configuration, we don't need to worry about starting and completing these transactions in our repository code.

We also define two references to the two query types that we have generated, `QCustomer` and `QAddress`. The array `customerAddressProjection` will hold the Querydsl `Path` entries for our queries, one `Path` for each column we are retrieving.

The constructor is annotated with `@Autowired`, which means that when the repository implementation is configured, the Spring container will inject the `DataSource` that has been defined in the application context. The rest of the class comprises the methods from the `CustomerRepository` that we need to provide implementations for, so let's get started.

Querying for a Single Object

First, we will implement the `findById` method (Example 5-7). The ID we are looking for is passed in as the only argument. Since this is a read-only method, we can add a `@Transactional(readOnly = true)` annotation to provide a hint that some JDBC drivers will use to improve transaction handling. It never hurts to provide this optional attribute for read-only methods even if some JDBC drivers won't make use of it.

Example 5-7. Query for single object

```
@Transactional(readOnly = true)
public Customer findById(Long id) {

  SQLQuery findByIdQuery = template.newSqlQuery()
    .from(qCustomer)
    .leftJoin(qCustomer._addressCustomerRef, qAddress)
    .where(qCustomer.id.eq(id));
```

```
  return template.queryForObject(
    findByIdQuery,
    new CustomerExtractor(),
    customerAddressProjection);
}
```

We start by creating an `SQLQuery` instance. We have already mentioned that when we are using the `QueryDslJdbcTemplate`, we need to let the template manage the `SQLQuery` instances. That's why we use the factory method `newSqlQuery()` to obtain an instance. The `SQLQuery` class provides a fluent interface where the methods return the instance of the `SQLQuery`. This makes it possible to string a number of methods together, which in turn makes it easier to read the code. We specify the main table we are querying (the `customer` table) with the `from` method. Then we add a left outer join against the `address` table using the `leftJoin(…)` method. This will include any address rows that match the foreign key reference between address and customer. If there are none, the address columns will be `null` in the returned results. If there is more than one address, we will get multiple rows for each customer, one for each address row. This is something we will have to handle in our mapping to the Java objects later on. The last part of the `SQLQuery` is specifying the predicate using the `where` method and providing the criteria that the `id` column should equal the `id` parameter.

After we have the `SQLQuery` created, we execute our query by calling the `queryForObject` method of the `QueryDslJdbcTemplate`, passing in the `SQLQuery` and a combination of a mapper and a projection. In our case, that is a `ResultSetExtractor` and the `customerAddressProjection` that we created earlier. Remember that we mentioned earlier that since our query contained a `leftJoin`, we needed to handle potential multiple rows per `Customer`.

Example 5-8 is the implementation of this `CustomerExtractor`.

Example 5-8. CustomerExtractor for single object

```
private static class CustomerExtractor implements ResultSetExtractor<Customer> {

  CustomerListExtractor customerListExtractor =
    new CustomerListExtractor(OneToManyResultSetExtractor.ExpectedResults.ONE_OR_NONE);

  @Override
  public Customer extractData(ResultSet rs) throws SQLException, DataAccessException {

    List<Customer> list = customerListExtractor.extractData(rs);
    return list.size() > 0 ? list.get(0) : null;
  }
}
```

As you can see, we use a `CustomerListExtractor` that extracts a `List` of `Customer` objects, and we return the first object in the `List` if there is one, or `null` if the `List` is empty. We know that there could not be more than one result since we set the parameter `expectedResults` to `OneToManyResultSetExtractor.ExpectedResults.ONE_OR_NONE` in the constructor of the `CustomerListExtractor`.

The OneToManyResultSetExtractor Abstract Class

Before we look at the `CustomerListExtractor`, let's look at the base class, which is a special implementation named `OneToManyResultSetExtractor` that is provided by the Spring Data JDBC Extension project. Example 5-9 gives an outline of what the `OneToManyResultSetExtractor` provides.

Example 5-9. Outline of OneToManyResultSetExtractor for extracting List of objects

```
public abstract class OneToManyResultSetExtractor<R, C, K>
    implements ResultSetExtractor<List<R>> {

  public enum ExpectedResults {
    ANY,
    ONE_AND_ONLY_ONE,
    ONE_OR_NONE,
    AT_LEAST_ONE
  }

  protected final ExpectedResults expectedResults;
  protected final RowMapper<R> rootMapper;
  protected final RowMapper<C> childMapper;

  protected List<R> results;

  public OneToManyResultSetExtractor(RowMapper<R> rootMapper, RowMapper<C> childMapper) {
    this(rootMapper, childMapper, null);
  }

  public OneToManyResultSetExtractor(RowMapper<R> rootMapper, RowMapper<C> childMapper,
      ExpectedResults expectedResults) {

    Assert.notNull(rootMapper);
    Assert.notNull(childMapper);

    this.rootMapper = rootMapper;
    this.childMapper = childMapper;
    this.expectedResults = expectedResults == null ? ExpectedResults.ANY : expectedResults;
  }

  public List<R> extractData(ResultSet rs) throws SQLException, DataAccessException { … }

  /**
   * Map the primary key value to the required type.
   * This method must be implemented by subclasses.
   * This method should not call {@link ResultSet#next()}
   * It is only supposed to map values of the current row.
   *
   * @param rs the ResultSet
   * @return the primary key value
   * @throws SQLException
   */
  protected abstract K mapPrimaryKey(ResultSet rs) throws SQLException;
```

```
  /**
   * Map the foreign key value to the required type.
   * This method must be implemented by subclasses.
   * This method should not call {@link ResultSet#next()}.
   * It is only supposed to map values of the current row.
   *
   * @param rs the ResultSet
   * @return the foreign key value
   * @throws SQLException
   */
  protected abstract K mapForeignKey(ResultSet rs) throws SQLException;

  /**
   * Add the child object to the root object
   * This method must be implemented by subclasses.
   *
   * @param root the Root object
   * @param child the Child object
   */
  protected abstract void addChild(R root, C child);
}
```

This `OneToManyResultSetExtractor` extends the `ResultSetExtractor`, parameterized with `List<T>` as the return type. The method `extractData` is responsible for iterating over the `ResultSet` and extracting row data. The `OneToManyResultSetExtractor` has three abstract methods that must be implemented by subclasses `mapPrimaryKey`, `mapForeignKey`, and `addChild`. These methods are used when iterating over the result set to identify both the primary key and the foreign key so we can determine when there is a new root, and to help add the mapped child objects to the root object.

The `OneToManyResultSetExtractor` class also needs `RowMapper` implementations to provide the mapping required for the root and child objects.

The CustomerListExtractor Implementation

Now, let's move on and look at the actual implementation of the `CustomerListExtractor` responsible for extracting the results of our customer and address results. See Example 5-10.

Example 5-10. CustomerListExtractor implementation for extracting List of objects

```
private static class CustomerListExtractor
  extends OneToManyResultSetExtractor<Customer, Address, Integer> {

  private static final QCustomer qCustomer = QCustomer.customer;

  private final QAddress qAddress = QAddress.address;

  public CustomerListExtractor() {
    super(new CustomerMapper(), new AddressMapper());
  }
```

```
  public CustomerListExtractor(ExpectedResults expectedResults) {
    super(new CustomerMapper(), new AddressMapper(), expectedResults);
  }

  @Override
  protected Integer mapPrimaryKey(ResultSet rs) throws SQLException {
    return rs.getInt(qCustomer.id.toString());
  }

  @Override
  protected Integer mapForeignKey(ResultSet rs) throws SQLException {
    String columnName = qAddress.addressCustomerRef.getLocalColumns().get(0).toString();
    if (rs.getObject(columnName) != null) {
      return rs.getInt(columnName);
    } else {
      return null;
    }
  }

  @Override
  protected void addChild(Customer root, Address child) {
    root.addAddress(child);
  }
}
```

The `CustomerListExtractor` extends this `OneToManyResultSetExtractor`, calling the superconstructor passing in the needed mappers for the `Customer` class, `CustomerMapper` (which is the root of the one-to-many relationship), and the mapper for the `Address` class, `AddressMapper` (which is the child of the same one-to-many relationship).

In addition to these two mappers, we need to provide implementations for the `mapPrimaryKey`, `mapForeignKey`, and `addChild` methods of the abstract `OneToManyResultSetExtractor` class.

Next, we will take a look at the `RowMapper` implementations we are using.

The Implementations for the RowMappers

The `RowMapper` implementations we are using are just what you would use with the regular `JdbcTemplate`. They implement a method named `mapRow` with a `ResultSet` and the row number as parameters. The only difference with using a `QueryDslJdbcTemplate` is that instead of accessing the columns with string literals, you use the query types to reference the column labels. In the `CustomerRepository`, we provide a static method for extracting this label via the `toString` method of the `Path`:

```
private static String columnLabel(Path<?> path) {
  return path.toString();
}
```

Since we implement the `RowMapper`s as static inner classes, they have access to this private static method.

First, let's look at the mapper for the `Customer` object. As you can see in Example 5-11, we reference columns specified in the `qCustomer` reference to the `QCustomer` query type.

Example 5-11. Root RowMapper implementation for Customer

```
private static class CustomerMapper implements RowMapper<Customer> {

  private static final QCustomer qCustomer = QCustomer.customer;

  @Override
  public Customer mapRow(ResultSet rs, int rowNum) throws SQLException {

    Customer c = new Customer();

    c.setId(rs.getLong(columnLabel(qCustomer.id)));
    c.setFirstName(rs.getString(columnLabel(qCustomer.firstName)));
    c.setLastName(rs.getString(columnLabel(qCustomer.lastName)));

    if (rs.getString(columnLabel(qCustomer.emailAddress)) != null) {
      c.setEmailAddress(
          new EmailAddress(rs.getString(columnLabel(qCustomer.emailAddress))));
    }

    return c;
  }
}
```

Next, we look at the mapper for the `Address` objects, using a `qAddress` reference to the `QAddress` query type (Example 5-12).

Example 5-12. Child RowMapper implementation for Address

```
private static class AddressMapper implements RowMapper<Address> {

  private final QAddress qAddress = QAddress.address;

  @Override
  public Address mapRow(ResultSet rs, int rowNum) throws SQLException {

    String street = rs.getString(columnLabel(qAddress.street));
    String city = rs.getString(columnLabel(qAddress.city));
    String country = rs.getString(columnLabel(qAddress.country));

    Address a = new Address(street, city, country);
    a.setId(rs.getLong(columnLabel(qAddress.id)));

    return a;
  }
}
```

Since the `Address` class has setters for all properties, we could have used a standard Spring `BeanPropertyRowMapper` instead of providing a custom implementation.

Querying for a List of Objects

When it comes to querying for a list of objects, the process is exactly the same as for querying for a single object except that you now can use the CustomerListExtractor directly without having to wrap it and get the first object of the List. See Example 5-13.

Example 5-13. Query for list of objects

```
@Transactional(readOnly = true)
public List<Customer> findAll() {

  SQLQuery allCustomersQuery = template.newSqlQuery()
      .from(qCustomer)
      .leftJoin(qCustomer._addressCustomerRef, qAddress);

  return template.query(
      allCustomersQuery,
      new CustomerListExtractor(),
      customerAddressProjection);
}
```

We create an SQLQuery using the from(…) and leftJoin(…) methods, but this time we don't provide a predicate since we want all customers returned. When we execute this query, we use the CustomerListExtractor directly and the same customerAddressProjection that we used earlier.

Insert, Update, and Delete Operations

We will finish the CustomerRepository implementation by adding some insert, update, and delete capabilities in addition to the query features we just discussed. With Querydsl, data is manipulated via operation-specific clauses like SQLInsertClause, SQLUpdateClause, and SQLDeleteClause. We will cover how to use them with the QueryDslJdbcTemplate in this section.

Inserting with the SQLInsertClause

When you want to insert some data into the database, Querydsl provides the SQLInsertClause class. Depending on whether your tables autogenerate the key or you provide the key explicitly, there are two different execute(…) methods. For the case where the keys are autogenerated, you would use the executeWithKey(…) method. This method will return the generated key so you can set that on your domain object. When you provide the key, you instead use the execute method, which returns the number of affected rows. The QueryDslJdbcTemplate has two corresponding methods: insertWithKey(…) and insert(…).

We are using autogenerated keys, so we will be using the insertWithKey(…) method for our inserts, as shown in Example 5-14. The insertWithKey(…) method takes a reference to the query type and a callback of type SqlInsertWithKeyCallback parameterized with the type of the generated key. The `SqlInsertWithKeyCallback` callback interface has a single method named `doInSqlInsertWithKeyClause(…)`. This method has the `SQLInsertClause` as its parameter. We need to set the values using this `SQLInsertClause` and then call `executeWithKey(…)`. The key that gets returned from this call is the return value of the `doInSqlInsertWithKeyClause`.

Example 5-14. Inserting an object

```
Long generatedKey = qdslTemplate.insertWithKey(qCustomer,
  new SqlInsertWithKeyCallback<Long>() {

    @Override
    public Long doInSqlInsertWithKeyClause(SQLInsertClause insert) throws SQLException {

      EmailAddress emailAddress = customer.getEmailAddress();
      String emailAddressString = emailAddress == null ? null : emailAddress.toString();

      return insert.columns(
              qCustomer.firstName, qCustomer.lastName, qCustomer.emailAddress)
          .values(customer.getFirstName(), customer.getLastName(), emailAddress);
          .executeWithKey(qCustomer.id);
    }
  });

customer.setId(generatedKey);
```

Updating with the SQLUpdateClause

Performing an update operation is very similar to the insert except that we don't have to worry about generated keys. The method on the `QueryDslJdbcTemplate` is called `update`, and it expects a reference to the query type and a callback of type `SqlUpdateCallback`. The `SqlUpdateCallback` has the single method `doInSqlUpdateClause(…)` with the `SQLUpdateClause` as the only parameter. After setting the values for the update and specifying the where clause, we call `execute` on the `SQLUpdateClause`, which returns an update count. This update count is also the value we need to return from this callback. See Example 5-15.

Example 5-15. Updating an object

```
qdslTemplate.update(qCustomer, new SqlUpdateCallback() {

  @Override
  public long doInSqlUpdateClause(SQLUpdateClause update) {

    EmailAddress emailAddress = customer.getEmailAddress();
    String emailAddressString = emailAddress == null ? null : emailAddress.toString();

    return update.where(qCustomer.id.eq(customer.getId()))
        .set(qCustomer.firstName, customer.getFirstName())
        .set(qCustomer.lastName, customer.getLastName())
        .set(qCustomer.emailAddress, emailAddressString)
        .execute();
```

```
  }
});
```

Deleting Rows with the SQLDeleteClause

Deleting is even simpler than updating. The `QueryDslJdbcTemplate` method you use is called `delete`, and it expects a reference to the query type and a callback of type `SqlDeleteCallback`. The `SqlDeleteCallback` has the single method `doInSqlDeleteClause` with the `SQLDeleteClause` as the only parameter. There's no need to set any values here—just provide the where clause and call execute. See Example 5-16.

Example 5-16. Deleting an object

```
qdslTemplate.delete(qCustomer, new SqlDeleteCallback() {

  @Override
  public long doInSqlDeleteClause(SQLDeleteClause delete) {
    return delete.where(qCustomer.id.eq(customer.getId())).execute();
  }
});
```

PART III

NoSQL

CHAPTER 6

MongoDB: A Document Store

This chapter will introduce you to the Spring Data MongoDB project. We will take a brief look at MongoDB as a document store and explain you how to set it up and configure it to be usable with our sample project. A general overview of MongoDB concepts and the native Java driver API will round off the introduction. After that, we'll discuss the Spring Data MongoDB module's features, the Spring namespace, how we model the domain and map it to the store, and how to read and write data using the `MongoTemplate`, the core store interaction API. The chapter will conclude by discussing the implementation of a data access layer for our domain using the Spring Data repository abstraction.

MongoDB in a Nutshell

MongoDB is a document data store. Documents are structured data—basically maps—that can have primitive values, collection values, or even nested documents as values for a given key. MongoDB stores these documents in BSON, a binary derivative of JSON. Thus, a sample document would look something like Example 6-1.

Example 6-1. A sample MongoDB document

```
{ firstname : "Dave",
  lastname : "Matthews",
  addresses : [ { city : "New York", street : "Broadway" } ] }
```

As you can see, we have primitive `String` values for `firstname` and `lastname`. The `addresses` field has an array value that in turn contains a nested address document. Documents are organized in *collections*, which are arbitrary containers for a set of documents. Usually, you will keep documents of the same type inside a single collection, where *type* essentially means "similarly structured." From a Java point of view, this usually reads as a collection per type (one for `Customers`, one for `Products`) or type hierarchy (a single collection to hold `Contacts`, which can either be `Persons` or `Companies`).

Setting Up MongoDB

To start working with MongoDB, you need to download it from the project's website. It provides binaries for Windows, OS X, Linux, and Solaris, as well as the sources. The easiest way to get started is to just grab the binaries and unzip them to a reasonable folder on your hard disk, as shown in Example 6-2.

Example 6-2. Downloading and unzipping MongoDB distribution

```
$ cd ~/dev
$ curl http://fastdl.mongodb.org/osx/mongodb-osx-x86_64-2.0.6.tgz > mongo.tgz

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 41.1M  100 41.1M    0     0   704k      0  0:00:59  0:00:59 --:--:--  667k

$ tar -zxvf mongo.tgz

x mongodb-osx-x86_64-2.0.6/
x mongodb-osx-x86_64-2.0.6/bin/
x mongodb-osx-x86_64-2.0.6/bin/bsondump
x mongodb-osx-x86_64-2.0.6/bin/mongo
x mongodb-osx-x86_64-2.0.6/bin/mongod
x mongodb-osx-x86_64-2.0.6/bin/mongodump
x mongodb-osx-x86_64-2.0.6/bin/mongoexport
x mongodb-osx-x86_64-2.0.6/bin/mongofiles
x mongodb-osx-x86_64-2.0.6/bin/mongoimport
x mongodb-osx-x86_64-2.0.6/bin/mongorestore
x mongodb-osx-x86_64-2.0.6/bin/mongos
x mongodb-osx-x86_64-2.0.6/bin/mongosniff
x mongodb-osx-x86_64-2.0.6/bin/mongostat
x mongodb-osx-x86_64-2.0.6/bin/mongotop
x mongodb-osx-x86_64-2.0.6/GNU-AGPL-3.0
x mongodb-osx-x86_64-2.0.6/README
x mongodb-osx-x86_64-2.0.6/THIRD-PARTY-NOTICES
```

To bootstrap MongoDB, you need to create a folder to contain the data and then start the *mongod* binary, pointing it to the just-created directory (see Example 6-3).

Example 6-3. Preparing and starting MongoDB

```
$ cd mongodb-osx-x86_64-2.0.6
$ mkdir data
$ ./bin/mongod --dbpath=data

Mon Jun 18 12:35:00 [initandlisten] MongoDB starting : pid=15216 port=27017 dbpath=data
  64-bit …
Mon Jun 18 12:35:00 [initandlisten] db version v2.0.6, pdfile version 4.5
Mon Jun 18 12:35:00 [initandlisten] git version: e1c0cbc25863f6356aa4e31375add7bb49fb05bc
Mon Jun 18 12:35:00 [initandlisten] build info: Darwin erh2.10gen.cc 9.8.0 Darwin Kernel
  Version 9.8.0: …
Mon Jun 18 12:35:00 [initandlisten] options: { dbpath: "data" }
Mon Jun 18 12:35:00 [initandlisten] journal dir=data/journal
Mon Jun 18 12:35:00 [initandlisten] recover : no journal files present, no recovery needed
```

```
Mon Jun 18 12:35:00 [websvr] admin web console waiting for connections on port 28017
Mon Jun 18 12:35:00 [initandlisten] waiting for connections on port 27017
```

As you can see, MongoDB starts up, uses the given path to store the data, and is now waiting for connections.

Using the MongoDB Shell

Let's explore the very basic operations of MongoDB using its shell. Switch to the directory in which you've just unzipped MongoDB and run the shell using the *mongo* binary, as shown in Example 6-4.

Example 6-4. Starting the MongoDB shell

```
$ cd ~/dev/mongodb-osx-x86_64-2.0.6
$ ./bin/mongo

MongoDB shell version: 2.0.6
connecting to: test
>
```

The shell will connect to the locally running MongoDB instance. You can now use the `show dbs` command to inspect all database, currently available on this instance. In Example 6-5, we select the local database and issue a `show collections` command, which will not reveal anything at this point because our database is still empty.

Example 6-5. Selecting a database and inspecting collections

```
> show dbs
local (empty)
> use local
switched to db local
> show collections
>
```

Now let's add some data to the database. We do so by using the `save(…)` command of a collection of our choice and piping the relevant data in JSON format to the function. In Example 6-6, we add two customers, Dave and Carter.

Example 6-6. Inserting data into MongoDB

```
> db.customers.save({ firstname : 'Dave', lastname : 'Matthews',
                      emailAddress : 'dave@dmband.com' })
> db.customers.save({ firstname : 'Carter', lastname : 'Beauford' })
> db.customers.find()
{ "_id" : ObjectId("4fdf07c29c62ca91dcdfd71c"), "firstname" : "Dave",
  "lastname" : "Matthews", "emailAddress" : "dave@dmband.com" }
{ "_id" : ObjectId("4fdf07da9c62ca91dcdfd71d"), "firstname" : "Carter",
  "lastname" : "Beauford" }
```

The `customers` part of the command identifies the collection into which the data will go. Collections will get created on the fly if they do not yet exist. Note that we've added

Carter without an email address, which shows that the documents can contain different sets of attributes. MongoDB will not enforce a schema onto you by default. The `find(…)` command actually can take a JSON document as input to create queries. To look up a customer with the email address of *dave@dmband.com*, the shell interaction would look something like Example 6-7.

Example 6-7. Looking up data in MongoDB

```
> db.customers.find({ emailAddress : 'dave@dmband.com' })
{ "_id" : ObjectId("4fdf07c29c62ca91dcdfd71c"), "firstname" : "Dave",
  "lastname" : "Matthews", "emailAddress" : "dave@dmband.com" }
```

You can find out more about working with the MongoDB shell at the MongoDB home page. Beyond that, [ChoDir10] is a great resource to dive deeper into the store's internals and how to work with it in general.

The MongoDB Java Driver

To access MongoDB from a Java program, you can use the Java driver provided and maintained by 10gen, the company behind MongoDB. The core abstractions to interact with a store instance are `Mongo`, `Database`, and `DBCollection`. The `Mongo` class abstracts the connection to a MongoDB instance. Its default constructor will reach out to a locally running instance on subsequent operations. As you can see in Example 6-8, the general API is pretty straightforward.

Example 6-8. Accessing a MongoDB instance through the Java driver

```
Mongo mongo = new Mongo();
DB database = mongo.getDb("database");
DBCollection customers = db.getCollection("customers");
```

This appears to be classical infrastructure code that you'll probably want to have managed by Spring to some degree, just like you use a `DataSource` abstraction when accessing a relational database. Beyond that, instantiating the `Mongo` object or working with the `DBCollection` subsequently could throw exceptions, but they are MongoDB-specific and shouldn't leak into client code. Spring Data MongoDB will provide this basic integration into Spring through some infrastructure abstractions and a Spring namespace to ease the setup even more. Read up on this in "Setting Up the Infrastructure Using the Spring Namespace" on page 81.

The core data abstraction of the driver is the `DBObject` interface alongside the `BasicDBObject` implementation class. It can basically be used like a plain Java `Map`, as you can see in Example 6-9.

Example 6-9. Creating a MongoDB document using the Java driver

```
DBObject address = new BasicDBObject("city", "New York");
address.put("street", "Broadway");
```

```
DBObject addresses = new BasicDBList();
addresses.add(address);

DBObject customer = new BasicDBObject("firstname", "Dave");
customer.put("lastname", "Matthews");
customer.put("addresses", addresses);
```

First, we set up what will end up as the embedded address document. We wrap it into a list, set up the basic customer document, and finally set the complex address property on it. As you can see, this is very low-level interaction with the store's data structure. If we wanted to persist Java domain objects, we'd have to map them in and out of `BasicDBObject`s manually—for each and every class. We will see how Spring Data MongoDB helps to improve that situation in a bit. The just-created document can now be handed to the `DBCollection` object to be stored, as shown in Example 6-10.

Example 6-10. Persisting the document using the MongoDB Java driver

```
DBCollection customers = db.getCollection("customers");
customers.insert(customer);
```

Setting Up the Infrastructure Using the Spring Namespace

The first thing Spring Data MongoDB helps us do is set up the necessary infrastructure to interact with a MongoDB instance as Spring beans. Using JavaConfig, we can simply extend the `AbstractMongoConfiguration` class, which contains a lot of the basic configuration for us but we can tweak to our needs by overriding methods. Our configuration class looks like Example 6-11.

Example 6-11. Setting up MongoDB infrastructure with JavaConfig

```
@Configuration
@EnableMongoRepositories
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
  }
}
```

We have to implement two methods to set up the basic MongoDB infrastructure. We provide a database name and a Mongo instance, which encapsulates the information about how to connect to the data store. We use the default constructor, which will assume we have a MongoDB instance running on our local machine listening to the default port, 27017. Right after that, we set the `WriteConcern` to be used to `SAFE`. The `WriteConcern` defines how long the driver waits for the MongoDB server when doing write operations. The default setting does not wait at all and doesn't complain about network issues or data we're attempting to write being illegal. Setting the value to `SAFE` will cause exceptions to be thrown for network issues and makes the driver wait for the server to okay the written data. It will also generate complaints about index constraints being violated, which will come in handy later.

These two configuration options will be combined in a bean definition of a `SimpleMongoDbFactory` (see the `mongoDbFactory()` method of `AbstractMongoConfiguration`). The `MongoDbFactory` is in turn used by a `MongoTemplate` instance, which is also configured by the base class. It is the central API to interact with the MongoDB instance, and persist and retrieve objects from the store. Note that the configuration class you find in the sample project already contains extended configuration, which will be explained later.

The XML version of the previous configuration looks as follows like Example 6-12.

Example 6-12. Setting up MongoDB infrastructure using XML

```
<?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:mongo="http://www.springframework.org/schema/data/mongo"
       xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
                           http://www.springframework.org/schema/data/mongo/
                               spring-mongo.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <mongo:db-factory id="mongoDbFactory" dbname="e-store" />

  <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongoDbFactory" />
    <property name="writeConcern" value="SAFE" />
  </bean>

</beans>
```

The `<db-factory />` element sets up the `SimpleMongoDbFactory` in a similar way as we saw in the JavaConfig example. The only difference here is that it also defaults the Mongo instance to be used to the one we had to configure manually in JavaConfig. We can customize that setup by manually defining a `<mongo:mongo />` element and configuring its attributes to the values needed. As we'd like to avoid that here, we set the `WriteConcern` to be used on the `MongoTemplate` directly. This will cause all write operations invoked through the template to be executed with the configured concern.

The Mapping Subsystem

To ease persisting objects, Spring Data MongoDB provides a mapping subsystem that can inspect domain classes for persistence metadata and automatically convert these objects into MongoDB `DBObject`s. Let's have a look at the way our domain model could be modeled and what metadata is necessary to tweak the object-document mapping to our needs.

The Domain Model

First, we introduce a base class for all of our top-level documents, as shown in Example 6-13. It consists of an `id` property only and thus removes the need to repeat that property all over the classes that will end up as documents. The `@Id` annotation is optional. By default we consider properties named `id` or `_id` the ID field of the document. Thus, the annotation comes in handy in case you'd like to use a different name for the property or simply to express a special purpose for it.

Example 6-13. The AbstractDocument class

```
public class AbstractDocument {

  @Id
  private BigInteger id;

  …
}
```

Our `id` property is of type `BigInteger`. While we generally support any type to be used as `id`, there are a few types that allow special features to be applied to the document. Generally, the recommended `id` type to end up in the persistent document is `ObjectID`. `ObjectID`s are value objects that allow for generating consistently growing `id`s even in a cluster environment. Beyond that, they can be autogenerated by MongoDB. Translating these recommendations into the Java driver world also implies it's best to have an `id` of type `ObjectID`. Unfortunately, this would create a dependency to the Mongo Java driver inside your domain objects, which might be something you'd like to avoid. Because `ObjectID`s are 12-byte binary values essentially, they can easily be converted into either `String` or `BigInteger` values. So, if you're using `String`, `BigInteger`, or `ObjectID` as `id` types, you can leverage MongoDB's `id` autogeneration feature, which will automatically convert the `id` values into `ObjectID`s before persisting and back when reading. If you manually assign `String` values to your `id` fields that cannot be converted into an `ObjectID`, we will store them as is. All other types used as `id` will also be stored this way.

Addresses and email addresses

The `Address` domain class, shown in Example 6-14, couldn't be simpler. It's a plain wrapper around three `final` primitive `String` values. The mapping subsystem will transform objects of this type into a `DBObject` by using the property names as field keys and setting the values appropriately, as you can see in Example 6-15.

Example 6-14. The Address domain class

```
public class Address {

  private final String street, city, country;

  public Address(String street, String city, String country) {

    Assert.hasText(street, "Street must not be null or empty!");
    Assert.hasText(city, "City must not be null or empty!");
    Assert.hasText(country, "Country must not be null or empty!");

    this.street = street;
    this.city = city;
    this.country = country;
  }

  // … additional getters
}
```

Example 6-15. An Address object's JSON representation

```
{ street : "Broadway",
  city : "New York",
  country : "United States" }
```

As you might have noticed, the `Address` class uses a complex constructor to prevent an object from being able to be set up in an invalid state. In combination with the `final` fields, this makes up a classic example of a value object that is immutable. An `Address` will never be changed, as changing a property forces a new `Address` instance to be created. The class does not provide a no-argument constructor, which raises the question of how the object is being instantiated when the `DBObject` is read from the database and has to be turned into an `Address` instance. Spring Data uses the concept of a so-called *persistence constructor*, the constructor being used to instantiate persisted objects. Your class providing a no-argument constructor (either implicit or explicit) is the easy scenario. The mapping subsystem will simply use that to instantiate the entity via reflection. If you have a constructor taking arguments, it will try to match the parameter names against property names and eagerly pull values from the store representation—the `DBObject` in the case of MongoDB.

Another example of a domain concept embodied through a value object is an `EmailAddress` (Example 6-16). Value objects are an extremely powerful way to encapsulate business rules in code and make the code more expressive, readable, testable, and maintainable. For a more in-depth discussion, refer to Dan Bergh-Johnsson's talk on this topic, available on InfoQ. If you carried an email address around in a plain `String`, you could never be sure whether it had been validated and actually represents a valid email address. Thus, the plain wrapper class checks the given source value