NTNU

Norwegian University of
Science and Technology

Construction of Object-Oriented Queries Towards Relational Data

In View of Industrial Practices

Stein Magnus Jodal

Master of Science in Computer Science

Submission date: February 2009

Supervisor: Svein Erik Bratsberg, IDI
Co-supervisor: Anders Haugeto, Iterate AS
Morten Berg, Iterate AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Today it is increasingly common to facilitate object-relational mapping (ORM) to access data stored in relational databases. As the data are represented as real objects we also need an object-oriented way to retrieve the objects. This becomes challenging when dealing with complex queries where the underlying relational data model places limitations on the queries. In the current solutions the source code of these complex, object-oriented queries are harder to comprehend than the corresponding SQL query on the relational data. The assignment is to identify the limitations of the current APIs for object-oriented queries and propose possible improvements on these in context of JPQL and Hibernate's Criteria API. Further, we want to compare this approach with the queries in the web development framework Django.

Assignment given: 22. September 2008
Supervisor: Svein Erik Bratsberg, IDI

Abstract

The focus of this work is querying relational data through an object-relational mapper (ORM). In Java projects, it is common to use the Hibernate ORM and write the queries using HQL and Criteria. These approaches have limitations in regard to readability and static analysis. The limitations are identified and explained in this thesis. Several possible solutions are discussed. One of the solutions is looked at in depth and implemented in a real world project. The described solution eases the construction of queries and provides a way to fully utilize the development support tools.

Preface

This Master's Thesis is the final part of a Master of Science degree from the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor at NTNU, Professor Svein Erik Bratsberg, and my co-supervisors at Iterate AS, Anders Haugeto and Morten Berg, for their support during my work on this thesis.

Also, I am thankful to Nina Heitmann for proofreading my work and for believing in me.

Stein Magnus Jodal (sign.)

Oslo, February 15, 2009

Contents

Listings

Acronyms

API Application Programming Interface

DAO Data Access Object

DBMS Database Management System

DOM Domain Object Model

DSL Domain Specific Language

DTO Data Transfer Object

HQL Hibernate Query Language

IDE Integrated Development Environment

JDBC Java Database Connectivity

JPA Java Persistence API

JPQL Java Persistence Query Language

JSR Java Specification Request

JVM Java Virtual Machine

LINQ Language Integrated Query

ORM Object-Relational Mapping

POJO Plain Old Java Object

QBC Query by Criteria

QBE Query by Example

SQL Structured Query Language

XML Extensible Markup Language

Chapter 1

Introduction

In software development it is usually a need for data to exist after the application has been shut down. Commonly, the data is stored in a relational database. The way data are stored and used in a database is quite different from how the data is used in an object-oriented software application. Retrieving data from a relational database in a way that conforms with object-oriented and development practices is a challenge. Thus, several software solutions for this purpose exist, so-called Object-Relational Mapping (ORM) layers, that provides an abstraction layer for the relation data.

Iterate is a software development consultancy which uses the Java platform for many of their projects. In most of these projects, they are using the popular ORM implementation Hibernate for managing the persistence of objects to relational databases. To retrieve previously persisted objects from the database they are using two different approaches to querying, which both have limitations.

The first is an API, called Criteria, which uses objects to build the query, but not all queries are possible to express using this approach, and complex queries expressed with the API are also difficult for the developers to read.

The second approach is based on query strings, similar to SQL, called HQL. The strings are comparably easy to understand, even for complex queries, but errors in the string queries are not detected before they fail at run-time, and the query strings are also easily broken by small changes to the domain object models.

We want a better solution which takes the best from each approach; an object-oriented API for querying where even complex queries are both possible to express and easy to understand.

A better solution could help developers create fewer defects, as an API that is easy to understand is also easier to use correctly. Instead of failing first at run-time, a good API will detect many invalid queries already at compile-time, or when an IDE is used, already when they are written.

A better query mechanism will make developers more effective. With the readability of e.g. SQL and full support for code completion in the IDE the queries will be easy both to comprehend and to formulate. Finally, a type safe and static API would make the code more flexible as the developer could change and reorganize the code safely in an automated fashion with the help of the IDE.

In this thesis we will study what possibilities exists for improving the task of query construction in context of Java and Hibernate, with the goal of proposing improvements to, or a replacement of, the currently used approaches.

1.1 Outline

The thesis is organized as follows. In Chapter 2 we look at ORM, Hibernate and the current approaches to querying. In Chapter 3 the problems with the current state are described, and we decide on the requirements for a solution. The requirements are used in Chapter 4 to evaluate five existing approaches to querying, and in Chapter 5 the one the query APIs is tried out in both a test project and in a real-world project. Chapter 6 looks at queries in Django, and the upcoming JPA2 standard. Finally, in Chapter 7 we discuss our findings, and Chapter 8 concludes our work.

Chapter 2

Background

This chapter first explains what ORM is, why we need it, and how queries relate to persisting of objects through an ORM. Then we look at a popular implementation of ORM for applications developed in the Java programming language, called Hibernate, and its two approaches to constructing queries, HQL and Criteria. Lastly, we take a look at what an Integrated Development Environment (IDE) is, and how it can help the developer with code completion and refactoring.

2.1 Object-Relational Mapping

Today, ORM software is an important tool in most software development where objects are persisted, or, in other words, the data outlives a single execution of the application. Figure 2.1 shows stack of software where the upper layers are closer to the user interface, and the lower are closer to the hardware. An application built using object-oriented techniques are at the top of this stack, and it produces some valuable data that needs to be persisted to a database, or it needs to access some previously persisted data to do its job.

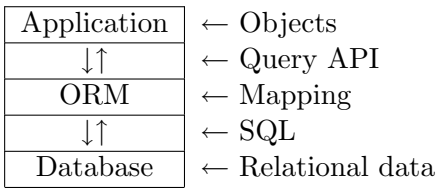

Figure 2.1: The surroundings of an ORM

The figure illustrates the ORM software's placement between the application and the database, where it maps data between the different models used by the two layers above and below. Between the application and the ORM, either some string based query language like HQL is used,

or we find some form of query API, e.g. Hibernate's Criteria API, where the query is constructed in a programmatic way using objects and methods. The ORM software translates the query string, or the calls to the query API, to SQL, which is the query language most common databases understands. Typically, the application, query API and the ORM software are written in the same programming language, while the SQL query is plain text, and at the bottom is any database which understands SQL or a similar query language.

In other words, the role of the ORM software is to bridge the gap between the object-oriented models in the application at the top of the stack, and the relational data in the database at the bottom, solving what is known as the impedance mismatch problem.

2.1.1 The Impedance Mismatch Problem

The impedance mismatch problem [29, Ch. 8] has its origin in the difference between the data model of e.g. SQL at the database level, and conventional programming languages, like C and Java, at the application level.

The conventional programming languages has a series of data types, like integers, floats, and chars, pointers, and various data structures, like arrays. The data has no inherent meaning, but are gradually modified through the series of simple statements which any program consists of. This is known as the imperative programming paradigm [35].

On the other hand, relational databases and SQL uses a relational data model, consisting of tables—with data tuples as rows, and attributes as columns—and the relations between the tables. The database queries are not imperative like the programming languages, but declarative. That is, the queries states *what the result should be*, and not *how to achieve the result*, which would include how to find, read, parse, compute, and store the data, and preferably be efficient in doing so. The *how* is taken care of by the Database Management System (DBMS), which makes it possible for users without knowledge of a database's inner workings to get useful information from the database with comparatively little effort.

Obviously, the differences between imperative programming languages and declarative database queries are rather large, and the different paradigms are well suited for their particular problem domains. As it would be a lot harder to express the task efficiently using the opposite paradigm, it is both impractical and inefficient to do everything in SQL or to do everything in a generic programming language. Thus, if database operations are a part of the application, a combination of SQL and generic programming languages are needed, and we need to easily pass data between them.

Further, the differences in the data models makes the needed marshalling and unmarshalling of data passed between SQL and a generic programming

language both non-trivial and repetitive. This again makes the developer's task of mapping prone to human error, or difficult due to lack of knowledge, often resulting in e.g. SQL injection possibilities [26, p. 74], one of the more common security problems in web applications.

These are important problems that affect a large portion of today's applications, both web applications and custom business applications, and this is what ORM software provides a solution for.

2.1.2 ORM and Querying

When developing an application that persist objects through an ORM, there are essentially two ways to get a reference to a persisted object: querying and navigation. Both are functionality provided by the ORM.

When we need an object that is unrelated to the objects we are already working on—or if we do not have references to any persisted objects at all—we need to use a query. Search is a typical case. If we are interested in a customer whose last name is "Smith", a query could easily provide a list of all matching customers.

After we have received some initial objects by querying, we can use navigation to get to objects which are related. Say we pick the customer we were searching for from the list returned by the query, and call this object *aliceSmith*. Then we want to list all invoices sent to this customer. One solution is to issue a new query for all invoices with a relation to the customer. The other solution is to traverse the references from the customer object to its child objects. To retrieve a list of all the invoice objects related to the customer, we could simply call *aliceSmith.getInvoices()*. This is called navigation, and makes the ORM issue the needed queries for us in the background, while we are using the objects as if they were all in memory and not persisted to a database.

In this thesis, we are focusing solely on querying, and not on navigation.

2.2 Example Data Model

The simple data model in Figure 2.2 is used throughout most of our work. The rectangles are entities, i.e. tables in a database, or classes in an object-oriented programming language. The ellipses are fields, i.e. columns in a database, or attributes of a class. An instance of an entity, like "Alice Smith" of the customer entity, becomes a row in the database table, or an instantiated object of the given class.

Explaining the entities, their fields and relations briefly: All entities have an unique ID. A customer has a name and zero or more invoices. Each invoice has zero or more invoice lines, and must have one and only one customer. Each invoice line has an amount, and must have one and only one invoice.

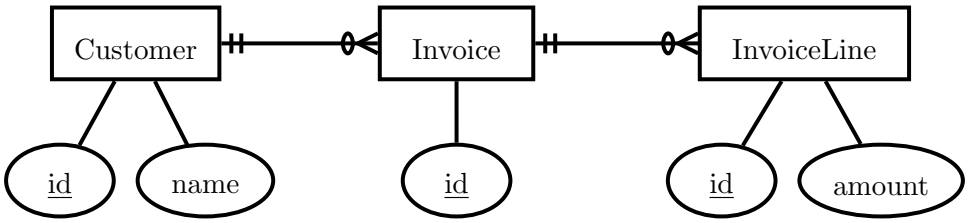

Figure 2.2: ER diagram of the data model used in examples

2.3 Hibernate

Hibernate [6] is one of the most widely used ORM implementations for Java. It was originally developed by a team of Java developers lead by Gavin King. JBoss, Inc.—a part of Red Hat, Inc. since June 2006—later hired King to work on Hibernate and has since been leading the development. Hibernate is freely available under the terms of the open-source license GNU Lesser General Public License, also known as the LGPL.

2.3.1 Java Persistence

Java Persistence API (JPA) is a part of Java Specification Request (JSR) 220, which is titled *Enterprise JavaBeans 3.0* [23]. The specification was published in its final form in May 2006 by the EJB 3.0 software expert group. The stated goal of the work was to improve *the EJB architecture by reducing its complexity from the developer's point of view.*

Java Persistence provides persistence for plain old Java objects, POJOs, using object-relational mapping. Java Persistence can be divided in three parts: JPA, i.e. the Java package *javax.persistence*, the query language, Java Persistence Query Language (JPQL), and metadata describing the mapping between objects and relational data [12].

2.3.2 Application Programming Interface (API)

Hibernate may either be used by programming directly against Hibernate's own API, or one may use Hibernate as an Java Persistence Provider, as defined by JPA. In practice, this means that the code uses the interfaces of the *javax.persistence* package and Hibernate as an implementation of these interfaces. If we at some later time wants to replace Hibernate by another Java Persistence Provider, only small parts of the code, if anything, needs to be modified. If needing to use features supported by Hibernate which is not included by JPA, one can access the Hibernate API directly where needed and keep to JPA everywhere else. This minimizes the changes needed to the code if one ever were to replace Hibernate.

2.3.3 Mapping Objects to the Database

There are two common ways in Java Persistence and Hibernate to describe the mapping between objects and the database. The older way is to use definitions in Extensible Markup Language (XML) which lives beside the source code, as shown in Listing 2.1. NHibernate, the Microsoft .NET port of Hibernate, still only supports this variant.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
        >

<hibernate-mapping>
    <class name="example.domain.Customer" table="customer">
        <id name="id" column="id">
            <generator class="native" />
        </id>
        <property name="name" column="name" />
        <set name="invoices" inverse="true">
            <key column="customer" />
            <one-to-many class="example.domain.Invoice" />
        </set>
    </class>
</hibernate-mapping>
```

Listing 2.1: Mapping with XML

This XML file maps the class *example.domain.Customer* to the database table *customer*. The class attribute *id* is mapped to the database table column *id*, and the DBMS is given the responsibility of generating new unique IDs. The *name* attribute is mapped to the *name* column. Finally, the class attribute *invoices* are mapped to a set of all *Invoice* objects which have a reference to the current customer in the table column *customer*. The last mapping is just a convenience making navigation between related objects possible both in the direction defined in the database, where the invoice references its customer, and in the opposite direction like we see in this example. This is the meaning of the XML attribute *inverse="true"*.

The newer approach to mapping objects to database tables, as of Java 5, is to use the language feature *annotations* to describe the mapping directly in the source code. The annotations are of the form *@NameOfAnnotation(name1 = value1, name2 = value2)* and apply to the class, field or method directly following it. Multiple annotations can be applied to one element, and annotations can also be nested by using an annotation in the place of e.g. *value1*. An example of mapping a class with annotations, achieving approximately the same as the XML example, is shown in Listing 2.2.

```
1  @Entity
2  @Table(name = "customer")
3  public class Customer {
4      @Id
5      @GeneratedValue(strategy = GenerationType.AUTO)
6      private Long id;
7
8      private String name;
9
10     @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
11     private Set<Invoice> invoices;
12 }
```

Listing 2.2: Mapping with Java annotations

As we can see, the annotations encode the same information as the XML file, but the information is attached to the source code instead of being shipped as an independent file. There are a couple of differences. First, there is no annotation on the field *name*. Annotations can be assigned either to the class fields or to getters accessing the class fields. If, as in this case, one field is annotated, Hibernate treats all other fields as if they were annotated using *@Column(name = "fieldName")*. For the field *name* the behaviour is as if it had an *@Column(name = "name")* annotation. If a field should not be mapped, it should be marked with the annotation *@Transient*. Second, the *invoice* field is a bit simpler. We simply state that the field is mapped by the database column *customer* on the other side of the relation, while the relevant class on the other side is inferred from the type of the set, i.e. from *Invoice* in *Set<Invoice>*, and the relevant database table is taken from the other class' mapping.

2.3.4 Hibernate Query Language

Moving on to queries, which is what we will focus on throughout this work. Hibernate Query Language (HQL) is Hibernate's primary means for constructing queries. HQL is a superset of JPQL, which means that all JPQL queries are valid HQL queries, but HQL supports additional features, some of which predates the Java Persistence standardization.

A HQL query is a text string consisting mostly of HQL keywords, class names, and property names. It may look very similar to a SQL query, as it uses many of the same keywords, like *select*, *from*, and *where*, but HQL works with classes, objects and attributes instead tables, rows and columns. HQL also supports object-oriented concepts such as inheritance, polymorphism and association [33, Ch. 14].

Listing 2.3 shows the short Java method *searchForCustomer()*. It takes a string *name* as its only argument, uses a HQL query at line 2–3 to request all

```
1 public List<Customer> searchForCustomer(String name) {
2     String query = "from Customer as customer "
3         + "where customer.name = :name";
4     return session.createQuery(query)
5         .setString("name", name)
6         .list();
7 }
```

Listing 2.3: Example of HQL usage

persisted *Customer* objects with names matching *name*, executes the query, and returns the result as a list. Note that instead of using the string *name* directly in the query, a placeholder *:name* is used, and then replaced with the content of *name* at line 5. This avoids SQL injection vulnerabilities, as mentioned in Section 2.1.1, completely.

In SQL the corresponding query would be approximately *SELECT * FROM customer WHERE name = ?*. Very similar to the HQL query, but with important differences. The HQL query works on a Java class (*Customer*), its instances (*customer*), and the instances' attribute (*name*). The results returned by the query are real Java objects. We know that the objects are persisted in a database somewhere, but no hints to this or any information about the database leaks through the small bits of Hibernate API used or the query string itself.

Accordingly, the SQL query works on a database table (*customer*), selecting all its data fields (*), and returning a list of tuples of primitive data types. The same data are available, but in as primitives in a data structure without any further meaning or functionally associated with them. Back in the result of *searchForCustomer()*, the data are stored as fields on a series of object instances—the data are the instances' state—which also contain methods applying behaviour on these fields. The data and the related business logic are encoded in a single location, called a business object or domain object.

2.3.5 The Criteria API

In addition to HQL, Hibernate exposes an API called Criteria for query construction [33, Ch. 15]. While HQL constructs queries using strings of text, Criteria build the query using objects. Text strings are only for e.g. the names of class attributes.

The method *searchForCustomer()* in Listing 2.4 implements the same functionality as the one in Listing 2.3, only using the Criteria API instead of HQL. Using a Hibernate session (*session*), it creates a Criteria for the class *Customer*. If we now just added *.list()*, we would get a list of all *Customer* instances. As we want to restrict the result list to instances where the *name* attribute matches the *name* argument of *searchForCustomer()*, we

```
1 public List<Customer> searchForCustomer(String name) {
2     return session.createCriteria(Customer.class)
3         .add(Restrictions.eq("name", name))
4         .list();
5 }
```

Listing 2.4: Example of Criteria (QBC) usage

use *.add()* to add a *Restriction* instance. The *Restriction* instance requires that *Customer.name* equals the string *name*.

The example in Listing 2.4 uses what in Hibernate terms are called Query by Criteria (QBC). The Criteria API also supports another way to query objects called Query by Example (QBE). When using QBE we create a new object of the same type as we want to receive as a result of the query. One this new object—the "example" object—we set the attributes which we want to restrict the result set on. E.g. if we want all *Customer* objects whose *name* is Alice, we would create a new *Customer* object and call *setName("Alice")* on the object. A complete example is shown in Listing 2.5.

```
1 public List<Customer> searchForCustomer(String name) {
2     Customer exampleCustomer = new Customer();
3     exampleCustomer.setName(name);
4     return session.createCriteria(Customer.class)
5         .add(Example.create(exampleCustomer))
6         .list();
7 }
```

Listing 2.5: Example of Criteria (QBE) usage

The queries in Listings 2.4 and 2.5 are almost identical. When using QBC we add an *org.hibernate.criterion.Restrictions* instance stating that the *name* attribute of the returned objects should match the local variable *name*. When using QBE we add an *org.hibernate.criterion.Example* instance holding an example *Customer* object where all attributes is *NULL*, except the *name* attribute, which is set to the value of the local variable *name*.

We achieve exactly the same result, but in two different ways. The two approaches may be more or less intuitive and fitting, depending on the situation and the complexity of the query.

2.4 Integrated Development Environment

Some of the limitations we identify in the next chapter are related to how the developer cannot fully utilize the development support tools with the currently used approaches to querying, thus we will explain the importance

of an IDE in modern software development, and how it can help the developer with code completion and refactoring.

Java applications are usually developed in an Integrated Development Environment (IDE). An IDE combines multiple development tools into an integrated suite. Common components in an IDE are a source code editor, a compiler, a debugger, build automation tools, a unit test runner, and integration with one or more version control systems. For Java, three well known IDEs are Eclipse [5], IntelliJ IDEA [9], and NetBeans [13].

2.4.1 Code Completion

Code completion in source code editors is basically the same as auto completion in other types of application, like interactive command line shells or the address field of a web browser. The common factor between these applications is that they have a limited range of valid user input.

In source code editors, code completion is commonly implemented using a pop-up select list displaying the possible continuations of the current prefix. The list pops up after the developer has stopped typing for a given length of time, e.g. 0.5 seconds, or she presses a key combination, typically *Ctrl+Space*.

In programming languages, the valid input is restricted to the keywords for the programming language and the names and variables of the current namespace. In object-oriented programming languages, code completion is especially useful, as it can be used to explore the fields and methods available on an object and to navigate from object to object. Code completion both serves as easily available documentation, and encourages using good and descriptive names, even though they may be long, since even long names can be written using only a few key strokes.

2.4.2 Refactoring

Refactoring is to perform changes to a source code without changing its functionality or behaviour. The purpose of refactoring is to make the source code more readable and structured, removing duplication and shortcuts taken in previous development.

Most IDE source code editors can help the developer with common refactoring tasks, like moving or renaming classes, fields and methods, extracting interfaces from classes, and extracting new methods from blocks of code. When using the IDE for this, the IDE makes sure that all references to the changed elements are updated correctly. E.g. when renaming a variable, the developer only changes one usage of the variable, and the IDE handles the renaming of all the other uses of the same variable.

However, refactoring support relies on the IDE understanding the source code as completely as possible, usually through static analysis. This means

the IDE cannot reliably know if it should change a string matching the name it is currently changing all occurrences of, or if the string content just coincidentally happens to match the name.

Chapter 3

Limitations and Requirements

In this chapter, we will look at the limitations of queries in Hibernate when using HQL and the Criteria API, then we will define requirements for a better solution, which is used when looking at various query APIs in Chapter 4.

3.1 Limitations of Queries in Hibernate

The following sections explain the limitations, with regard to query construction in Hibernate, experienced by consultants at Iterate who use Hibernate in their daily work. Primarily, the focus is usability and ease of use, and not technical corner cases.

3.1.1 Strings in Queries

The main limitation of using HQL is that the queries are formed as strings. The use of strings to encapsulate query logic in the application means that the IDE cannot help with code completion, and it cannot safely change the query when e.g. refactoring the domain object model. For the developer, the use of strings also means that he gets no validation of whether the query is syntactically valid before executing it, hopefully in a unit test. This leads to a slower write-test iteration time, as the developer does not get instant feedback on the validity of his code. The alternative is to use an API and an IDE which supports code completion and makes many invalid queries fail already at compile-time, which in practice means that the IDE detects the error immediately.

For the Criteria API, strings are still a problem, but to a lesser extent. In the Query by Criteria example of Listing 2.4 strings are only used for the name of fields on the object queried, while in the Query by Example

example of Listing 2.5, the use of strings is avoided completely by calling the fields' *set* method on the example object instead. Unfortunately, QBE is by its nature not applicable for other queries than *selections* on single tables. These are also very simple queries, while in a bit more complex QBC queries, like in Listing 3.1, strings are also used for aliases, corresponding to the *AS* keyword in SQL and the *rename* operation in relational algebra.

3.1.2 SQL Experience of Little Use

The next problem with Criteria is related to experience with SQL. SQL is the greatest common factor between database-driven applications implemented in different languages and frameworks. This means that many, if not most, developers got some experience formulating queries using SQL. Experience which is of little help when using the Criteria API as its resemblances with SQL is limited. For example, the terminology is mostly different, *projections* is used in place of *SELECT*, and *restrictions* in place of *WHERE*.

3.1.3 Low Signal-to-Noise Ratio

A more subjective consideration, but still a valid one, is the signal-to-noise ratio of queries using the Criteria API. A low signal-to-noise ratio means that there is much unwanted background noise interfering with the signal.

```
1 public List<Object[]> retrieveInvoicesAndTotalAmounts() {
2     return getSession().createCriteria(Invoice.class, "invoice")
3         .createAlias("invoiceLines", "il")
4         .setProjection(Projections.projectionList()
5             .add(Projections.sum("il.amount"), "totalAmount")
6             .add(Projections.groupProperty("il.invoice")))
7         .list();
8 }
```

Listing 3.1: A more complex Criteria query

Consider the example in Listing 3.1. Most operations are performed by *add()*-ing some object to the query. The objects added are created by using static factory methods on classes such as *Projections* for *projections*, in relational algebra terms, and *Restrictions* for *selections*. Also, when adding more than one object of a type, one cannot simply list them as a series of arguments, but must create e.g. a *projectionList()*, again calling a static method on a class with a rather long name. In total, this query repeats the word *projection* four times. And that for a query where only two values—an invoice and the total amount of its invoice lines—is returned in each result tuple. The net result is a query where most of the text are due to how one

is forced to use the API, and not due to the size or complexity of the actual query.

```
1 SELECT invoice, SUM(il.amount)
2 FROM invoice JOIN invoiceLine il ON invoice.id = il.invoice
3 GROUP BY il.invoice
```

Listing 3.2: Same query in SQL

When comparing with the same query expressed in SQL in Listing 3.2, it becomes abvious that the Criteria queries contains too much noise and too little useful signal, and this severely reduces the queries' readability.

3.1.4 Missing Features

Finally, Criteria lack a bit in feature completeness when compared to HQL and SQL. As a concrete example of this, Criteria supports aggregations and the functionality of *GROUP BY* in SQL, but not the counterpart *HAVING*[1] [7]. This may force developers to use e.g. HQL to perform the query, even though one highly prefers to keep to APIs.

We do not address this problem, as the problem has little to do with usability and refactoring, but it is still a relevant problem for users of Criteria.

3.2 Requirements for a Solution

Given the identified limitations, the following requirements are proposed for a solution.

3.2.1 Usage

First, a query API must be easy to use for the developers and enable them to produce working, clean and readable code with little effort and few errors. To achieve this, the API must support as much code completion as possible and make use of the experience many developers got from SQL. To reduce run-time errors, the API should try to avoid the construction of invalid queries by fetching all syntactic and many semantic errors at compile-time.

[1] When performing aggregation operations, *HAVING* is used to select which data rows to include in groups. This is similar to how *WHERE* is used to select of which data rows should be included in *SELECT*, *UPDATE* and *DELETE* operations. In relational algebra, both *HAVING* and *WHERE* are known as *selections*.

3.2.2 Refactoring

Second, the API must support refactoring, as refactoring together with unit tests is maybe the most efficient way to keep code clean and flexible through time, keeping the amount of *technical debt* in the project to a minimum [34, 31, 30].

[34] defines technical debt as "the total amount of less-than-perfect design and implementation decisions in your project. This includes quick and dirty hacks intended just to get something working *right now!* and design decisions that may no longer apply due to business changes. (...) Unchecked technical debt makes the software more expensive to modify than to reimplement." This is maybe the largest challenge in software development projects today.

To support refactoring, the API must minimize the use of strings in the application as they cannot reliably be changed by an IDE, as we discussed in Section 2.4.2. Using strings in the API calls makes the code more fragile and prone to errors. When keeping to queries without strings, the code adopts easier to changes in the domain object models.

3.2.3 Integration

Finally, to make the query API applicable for more projects, the API must integrate easily with existing Hibernate projects.

To integrate easily, the API must build on Hibernate and JPA, and within reasonable boundaries keep to the same object types as Hibernate for e.g. query results. It should be possible to use the API as a drop-in replacement for Criteria or HQL without changing any other code than the Data Access Objects.

3.2.4 Summary

To summary, the requirements a solution needs to fulfill are as follows:

Usage
: Queries must resemble SQL, support code completion, and reject many invalid queries at compile-time.

Refactoring
: The API must minimize the use of strings.

Integration
: The API must build on Hibernate and JPA, and queries should primarily have the same return types etc. as HQL and Criteria.

Chapter 4

Existing Query APIs

In the Microsoft .NET framework they have solved the problems with queries by making them a native of the .NET programming languages, through what is known as Language Integrated Query (LINQ). In this chapter we will first take a brief look at LINQ and how their approach to queries could apply to Java.

Through initial research, we discovered five query APIs for Java which at the surface seemed to solve some of our problems. We study and evaluate each of them using the requirements from Chapter 3. Finally, we round of the chapter with a summary of the query APIs.

4.1 .NET Language Integrated Query

Microsoft .NET Framework introduced LINQ in version 3.5 released in November 2007 [28]. LINQ integrates queries into all the .NET programming languages by letting the queries keywords become keywords of the programming language too. LINQ is not specific to database access, but it does require data to be mapped to objects before it can be queried through LINQ. The mapping is done by LINQ providers, which also includes NHibernate, the .NET port of Hibernate.

```
1 public List SearchForCustomer(string Name) {
2     List<Customer> AllCustomers = GetCustomerList();
3     var Customers =
4         from c in AllCustomers
5         where c.name == Name
6         select c;
7     return Customers;
8 }
```

Listing 4.1: Example of LINQ usage in C#

Listing 4.1 shows the *SearchForCustomer* method implemented in C#.

LINQ are used for the query in lines 4–6. Note that the query is not delimited by quotation marks. The keywords of the query, like *from*, *in*, *where*, and *select*, are keywords of the programming language, just like *public* and *return*.

4.1.1 Language Integration in Java vs .NET

Several Java APIs mimic .NET LINQ, except they are not actually integrated into the language, but are regular APIs. These APIs can be considered as Domain Specific Languages implemented in Java. A Domain Specific Language (DSL) is a language for a specific problem, typically implemented in a generic programming language to avoid all the work which are needed to implement a full independent language.

The advantage of integration into the programming language over the use of an API is mostly the removal of *syntactic sugar* like parentheses around the method arguments, and the dots combining the return value of one method call with the next method call. E.g. *from(c).in(allCustomers)* has more syntactic sugar than *from c in allCustomers*.

On the other side, the advantages of APIs over a query language integrated into the programming language are that no special support in tools like IDEs are needed. The queries are just normal method calls and arguments, while the programming language itself is unchanged. In addition, not having the query mechanism integrated into the programming language opens up for a multitude of competing APIs and implementations.

When comparing the Microsoft-controlled .NET environment with the mostly open source Java environment, it is obvious that integrating a query mechanism into the programming languages of the .NET platform is considerably easier, considering that Microsoft provides most of the supporting tools, like Visual Studio. In the Java environment, language changes are driven through the open JSR process and tools are developed by numerous loosely cooperating companies and groups. After large changes to the Java language, it can take years before both the tool support and a large installation base of JVMs supporting the new features are in place.

In the next sections, we will give an overview of some query APIs for Java. All of them are inspired by LINQ or SQL when defining their public API.

4.2 Quaere

The Quaere project was started by the Norwegian developer Anders Norås as an example of a DSL implemented in Java. The example was meant for a presentation at the yearly JavaZone conference in Oslo in September 2007. The project became a bit more than an example, and was later released under the open source Apache Software Foundation License 2.0 [15].

4.2.1 Usage

Quaere resembles SQL in that it uses the same vocabulary of keywords. The main difference is that the *select* clause is at the end and not at the start, which seems very common among the Java APIs that resembles SQL, as we will soon see.

```
1  public class GettingStartedWithQuaere {
2      public static void main() {
3          City[] cities=City.ALL_CITIES;
4
5          Iterable<String> largePopulations =
6              from("city")
7              .in(cities)
8              .where(gt("city.getPopulation()", 10000000))
9              .select("city.getName()");
10
11         for (String cityName: largePopulations) {
12             System.out.println(cityName);
13         }
14     }
15 }
```

Listing 4.2: Query example with Quaere (from [16])

Listing 4.2 shows an example query, expressed with the Quaere API, which returns the name of all cities with a population greater than ten million. *from("city").in(cities)* in line 6–7 creates *city* as an alias to an element of *cities*, analogous to a Java 5 *for-each loop* such as *for (City city : cities)*.

The arguments to the *where()* call is more similar to Hibernate's Criteria API than to SQL. Just as Criteria, the *gt(a, b)* form is used instead of SQL's greater-than expression $a > b$, though this should be of little hindrance to the query's readability.

The query ends with a call to *select("city.getName()")*. For each *City* instance *city* remaining after the filtering done by *where()*, the values returned by *city.getName()* are returned as the result of the entire query.

Even before we get to the refactoring part where strings are banned, strings are causing problems for the usage. As *city.getPopulation()* and *city.getName()* both are expressed within strings, the developer gets no code completion when programming these parts of the query. Additionally, using *city.population* and *city.name* could have been supported through automatic expanding of *population* and *name* to the matching accessor methods. Hibernate does this by prefixing the field name with *is* for boolean fields and *get* for all other fields to get the read accessor method, and prefixing with *set* to get the write accessor method, as is the Java convention.

4.2.2 Refactoring

With regard to refactoring, Quaere has the same limitations as Criteria. Strings are used for both instance names, field names and method names. IDEs can of course do changes by best effort, but unless they have specific support for the exact query API used changes can never be made automatically with the same confidence as static queries would yield.

4.2.3 Integration

The current status of Quaere is that Quaere for Java collections are rather feature complete, but that Quaere for JPA is incomplete and has not seen any development since the start of 2008. Thus, Quaere is currently not usable for existing projects based on Hibernate or JPA, but will maybe be usable for such projects in the future.

4.2.4 Conclusion

At the time of writing, the project still labels itself as in pre-beta state and has seen no development in the last year, querying through JPA are not supported and the API itself is not any improvement over Criteria with regards to refactoring. The plus with Quaere is that the queries are readable without any previous experience with the specific API.

4.3 JaQu

JaQu is a query API for building SQL statements created by Thomas Mueller, also the original creator of HSQLDB and a contributor to Quaere before starting on JaQu [10]. Listing 4.3 shows an example query expressed in JaQu. The query returns a list of products which has no units in stock, ordered by the product ID.

```
1 public List<Product> retrieveSoldOutProducts() {
2     Product p = new Product();
3     List<Product> soldOutProducts =
4         db.from(p).
5         where(p.unitsInStock).is(0).
6         orderBy(p.productId).select();
7     return soldOutProducts;
8 }
```

Listing 4.3: Query example with JaQu (from [10])

4.3.1 Usage

The example shows that JaQu is very similar to SQL, which would be something like *SELECT * FROM Product WHERE unitsInStock = 0 ORDER BY productId*. That is, almost exactly the same as the JaQu query. Again we observe that the select part is moved from the start to the end of the query.

With regard to code completion, JaQu is clearly better than Quaere as the use of strings for e.g. the *where* clause is replaced with a static reference to the actual field on the model, which allows for both code completion when developing, and automatic renaming of e.g. field names when refactoring.

Further, JaQu seems to prefer single-argument—or monadic—methods, which also helps on the readability of the code [31, Ch. 3]. Where Quaere and Criteria uses the static method *gt()* with two arguments for a greater-than comparison, JaQu passes the first argument to *where()*, then on the returned object calls *is()* with the second argument to do an equals comparison, which is analogous to a greater-than comparision. Where Quaere only guides the developer and the IDE's code completion with the defined type of *where()*'s argument, JaQu can define exactly what methods are possible to call on the object returned by *where()*. This leads to the construction of fewer invalid queries that passes compile-time checks, but fails at run-time, and is thus speeding up the development cycle.

4.3.2 Refactoring

As mentioned, JaQu's use of static object and field references, instead of strings, improve the refactoring support substantially. For example, if one where to rename a field name on a domain class to make its meaning and use more obvious, the IDE's rename function could automatically and safely change the query. If strings where used, the situation would be very different. The IDE would have to guess, and probably ask the developer, if a string—or ten—which happens to contain the field name, also should be changed.

4.3.3 Integration

As for integration, JaQu currently works as a replacement for an independent ORM, like Hibernate. It is though stated in the list of ideas for future enhancements at [10] to "internally use a JPA implementation (for example Hibernate) instead of SQL directly." If and when this happens, JaQu may be of more interest for existing projects searching for new query mechanisms on top of Hibernate.

4.3.4 Conclusion

To sum up, JaQu seems interesting with regards to the readable queries it creates and the good support for automatic refactoring. Though, due to lack of support for use on top Hibernate or JPA it does not integrate easily into existing Hibernate projects.

4.4 Squill

Squill is a query API with focus at replacing SQL queries in code with an almost identical DSL [22]. The API works on Data Transfer Objects generated from metadata about the layout of the database. Objects which are free of business logic, in contrast to traditional domain objects that Hibernate works with.

```
1  ComplaintTable c = new ComplaintTable();
2  List<Tuple2<String, Integer>> rows =
3      squill
4          .from(c, c.customer)
5          .where(
6              gt(c.customer.isActive, 0)
7              notNull(c.percentSolved),
8              notNull(c.refoundSum))
9          .orderBy(desc(c.customer.id))
10         .selectList(
11             c.customer.lastName,
12             c.percentSolved);
13
14 for (Tuple2<String, Integer> tuple2 : rows) {
15     System.out.println(
16         "Customer " + tuple2.v1 + " has a complaint solved "
17         + tuple2.v2 + "%");
18 }
```

Listing 4.4: Query example with Squill (from [22])

4.4.1 Usage

Listing 4.4 shows a *from-where-orderby-select* query using Squill. The query returns a list of customer's last names and how solved their complaint is. Only active complaints with a set percent and refund sum are included. The results are ordered by the customer ID in descending order.

Again, the select clause has moved from the beginning to the end of the query, when comparing to SQL. Except for this minor detail, the query is constructed in the same way as a SQL query, and [22] states that "it is designed to allow everything SQL allows you to do, exactly the way SQL is meant to do it."

The *from* clause uses two database tables: *c*, the complaint table, and *c.customer*, the customer table. By accessing the customer table through *c*, the tables are joined. No aliases for the tables are created, but *c* and *c.customer* are used consistently throughout the query. In the *where* clause, three restrictions are applied by a single call to *where()* with one argument for each restriction. The first restriction, *gt(c.customer.isActive, 0)* uses a static method *gt()* just as Criteria and Quaere does. Though, there is one major difference. The Squill uses no strings at all, which enables the use of code completion in the IDE. The *order by* orders the result by the customer's ID in descending order by calling *orderBy(desc(c.customer.id))*, while the corresponding SQL would be *ORDER BY customer.id DESC*. Finally, a call to *selectList* with two arguments returns a list of *Tuple2* objects, each with a String and an Integer value, which are explicitly defined using Java generics. Squill has *TupleN* classes for N in 1–10. These are used as data structures for returning multiple values in a typesafe manner, using Java generics for all the individual types in the tuple. To access the values, the *TupleN* objects got N fields named *v1* to *vN*, as seen in the for-loop in the listing.

4.4.2 Refactoring

As JaQu, Squill avoids the use of strings for instance, field and method names, and instead work on its own table objects which directly represent database tables. As Squill also uses generics, it is "almost complete typesafe" and further guides the IDE in understanding the code and performing correct refactorings.

4.4.3 Integration

As Squill is a direct replacement for SQL in the source code, it does not work on top of an ORM like Hibernate but executes the generated SQL queries directly through a Java Database Connectivity (JDBC) data source. Also, it uses simple DTOs directly representing database tables and columns, instead of the more common Domain Object Model (DOM) objects which may abstract parts of the database representation and usually are loaded with business logic in addition to representing persistable data.

4.4.4 Conclusion

Since Squill maps very closely to SQL and aims to make everything possible with SQL possible through Squill in the same way, it may be interesting for use in performance-critical parts of an application, where the developer wants full control of what queries are passed to the database. Normally, developers will be forced to drop from a query API to raw SQL through the JDBC API, but with Squill they can get the same amount of control as

raw SQL and at the same time achieve increased productivty from the code completion and type safety.

4.5 JaQue

JaQue is another Java query library, though the name could be a typo of JaQu. Development was started by Konstantin Triger in early 2008, and is still active [11]. It is licensed under the GNU General Public License version 3 or newer. JaQue takes another approach than the previously discussed APIs in that it makes use of *closures*, a language feature which may be part of Java 7.

Closures are functions with internal variables that bind to variables in the executing environment at run-time. In the prototype implementation for Java [2], closures can be used to wrap any block of code into a closures object, which may then be passed around, e.g. as an argument to a method. The execution of the wrapped code is deferred to the *invoke()* method is called on the closure object. The call to *invoke()* returns the result of the last statement in the wrapped code block. Two examples with explaining comments can be seen in Listing 4.5.

```
1 // A closure is of the form { arguments => code }.
2
3 // Closure which take no arguments and always returns 42.
4 int answerToEverything = { => 42 }.invoke();
5
6 // Returns the circumference of a circle with radius 5.
7 int circumference = { int radius => 2 * Math.PI * radius }.
      invoke(5);
```

Listing 4.5: Proposed syntax of Java closures

4.5.1 Usage

Listing 4.6 shows a very simple query using JaQue and closures. *data* is some collection of Integer objects. The *where()* call gets passed a closure object which given an integer from the collection returns true if it should be included and false otherwise. In this case the closure returns true if the integer is larger than five. Thus, all integers less than or equal to five are excluded from the result. Next up, *orderBy()* is passed another closure which takes two integers and returns a negative number, zero or a positive number, depending upon how the to integers compare to each other. Thus, the query returns a list of all integers in *data* that are larger than five, in ascending order.

```
1 Iterable<? extends Number> r =
2     from(data,
3         where({ Integer i => i > 5 },
4             orderBy({ Integer i1, Integer i2 => i1-i2 })));
```

Listing 4.6: Query example with JaQue (from [11])

This example is similar to SQL, while larger examples would be more analogous to LINQ, as the vision of JaQue is to provide the capabilities of LINQ on a Java platform. Though, as the same vocabulary is used in both SQL and LINQ, JaQue queries should be relatively easy to understand by anyone familiar with SQL. The exception to this claim is the use of closures, which at the time of writing is a rather unknown concept to most Java programmers, unless they are familiar with more functional programming languages. Though, if closures are included in Java 7, Java developers will also eventually become familiar with closures.

4.5.2 Refactoring

With regard to refactoring, JaQue is very promising. Strings are not used in any part of the queries, neither to describe the names of object attributes nor anything else. The use of closures, which essentially is real code executed at a later time, also makes it possible to describe filtering by using plain domain objects, and not some form of manually or auto-generated proxy object, which most other string-free approaches must resolve to.

Additionally, JaQue uses Java generics extensively to be as type safe as possible, which is generally of help for an IDE's understanding of the code and its semantics.

4.5.3 Integration

JaQue depends on a prototype of a Java closure proposal [2], and should thus itself be considered only a prototype for how future query APIs in Java may look like. Also, Java 7 is still in the future, but it will probably be years from it is released to its deployment base is large enough for developers to depend on the availability of new language features like closures.

At this time, JaQue supports querying objects and XML, but it does not support JPA fully. A subproject named *jaque2jpa* exists for bringing JaQue to JPA implementations, but it is still under development.

4.5.4 Conclusion

JaQue is an interesting and different approach to queries. Given more understanding of how closures works among developers, its queries are both readable to humans and well suited for statical analysis and refactoring

by IDEs. Though, use of JaQue in production code today should clearly be avoided, if it is at all technically possible considering the lack of JPA support.

4.6 Querydsl

Querydsl was created by the software consulting company Mysema in Helsinki, Finland, and as such is the sole query API in this comparison which is backed by a company. It is an open-source project, licensed under GNU Lesser General Public License 2.1 [16], which is the same license as Hibernate uses.

4.6.1 Usage

```
1  public List<Customer> retrieveCustomersWithTotalSalesLargerThan(
       int amount) {
2      QCustomer customer = new QCustomer("customer");
3      QInvoice invoice = new QInvoice("invoice");
4      QInvoiceLine invoiceLine = new QInvoiceLine("invoiceLine");
5      return from(customer)
6          .innerJoin(customer.invoices.as(invoice))
7          .innerJoin(invoice.invoiceLines.as(invoiceLine))
8          .groupBy(customer, customer.name)
9          .having(sum(invoiceLine.amount).gt(amount))
10         .list(customer);
11 }
12
13 private HqlQuery from(EEntity<?>... s) {
14     return new HqlQuery(getSession()).from(s);
15 }
```

Listing 4.7: Query example with Querydsl

Listing 4.7 shows an example of Querydsl in use. The query returns a list of customers who has received invoices with a total amount greater than a given threshold *amount*. The resemblances between the Querydsl API and SQL or HQL are obvious, as with all the other APIs except JaQue. The queries are easy to understand by somebody experienced with SQL, even without any experience in the Querydsl API.

A translation from Querydsl to SQL is easy to perform. With all SQL keywords capitalized, *from(customer)* becomes *FROM customer*, *innerJoin(customer.invoices.as(invoice))* becomes *INNER JOIN invoice ON invoice.customer_id = customer.id*, and analogous for the next join. *groupBy(customer, customer.name)* becomes *GROUP BY customer.id, customer.name*, *having(sum(invoiceLine.amount).gt(amount))* becomes e.g. *HAVING SUM(invoiceLine.amount) > 1000*, and finally *list(customer)*

becomes *SELECT customer.**. The Querydsl query and the SQL is almost identical, word for word. The method names in Querydsl generally match the keywords in SQL. The only large difference for the reader is in the syntactic sugar, i.e. spacing, parentheses, and punctuation.

4.6.2 Refactoring

As Listing 4.7 shows, the use of strings is contained to the creation of Querydsl *query types*—the classes whose names start with *Q*—which may be reused in multiple queries. The queries themselves are completely free of string usage. This makes code with queries written using the Querydsl API easily adaptable to changes in the DOM classes.

4.6.3 Integration

Querydsl comes with three back ends: Java collections, SQL and HQL. Through the HQL back end, Querydsl may be used as a query API on top of the Hibernate ORM, replacing confusing object-oriented Criteria queries and string-based HQL queries. That HQL is a supported back-end can also be seen in in Listing 4.7, where the *from()* method actually is a simplifying wrapper for a *HqlQuery()*.

4.6.4 Conclusion

Queries formed using Querydsl are readable for developers familiar with SQL. They do not use strings, which means that the IDE can both understand how to apply refactorings safely to the queries, provide code completion during development, and make most invalid queries fail at compile-time. The API supports Hibernate through a HQL back end, seems more mature than the other APIs considered and are also supported by a company.

4.7 Summary

Table 4.1 summarizes how the reviewed query APIs met the specific requirements of our usage, refactoring and integration criteria. For comparison, the Criteria API which are a part of the Hibernate distribution are also included in the summary table.

All the alternatives are inspired by SQL or LINQ. Quaere is the only one using strings, and thus also the worst alternative with regard to code completion. JaQu and Squill does not support using JPA as a back-end. JaQue are working on JPA support, but JaQue is only to be considered a prototype as it uses closures, a still not standardized future feature of the Java language.

API	Inspiration	Code compl.?	Strings?	Support JPA?
Criteria	Rel. algebra	Some	Yes	Yes
Quaere	SQL	Some	Yes	In development
JaQu	SQL	Yes	No	No
Squill	SQL	Yes	No	No
JaQue	LINQ	Yes	No	In development
Querydsl	SQL	Yes	No	Yes

Table 4.1: Summary of the Query APIs

The last one, Querydsl, is inspired by SQL, does not use strings, supports code completion, and are primarily targeted at JPA as the back end. In other words, Querydsl seems to provide a solution matching all our requirements. Though, we need to get more experience with Querydsl and test how it integrates into existing projects. In the next chapter, we will use Querydsl in two different projects.

Chapter 5

Applying Querydsl to Existing Projects

After reviewing five query APIs with various properties in Chapter 4 Querydsl emerged as the most interesting alternative. At the surface it seems like a good match for requirements we defined back in Chapter 3, but one point remains: How easy is it to put Querydsl to use in an existing project which already uses Hibernate with HQL or the Criteria API?

For the API to be a possible replacement of HQL and Criteria, it must be easy to start using for most projects, and it should be fair to assume that most projects have an existing code base. Thus, we need to know how well Querydsl integrates into existing code bases, and what changes are needed, if any, to the code base.

First, to get to know Querydsl in a familiar environment, we extend the playground project used for the Hibernate examples in Section 2.3. Then we will try Querydsl by reimplementing the persistence facade of a real-world project: Iterate's internal communication platform, known as LeanCast.

5.1 Setting up Querydsl

Before starting to write queries, we need to add Querydsl as a dependency of our project so that it becomes available in the class path of our project. Second, we need to setup automatic generation of the query types, which are based on our domain objects. We use Apache Maven [1] for both tasks, and then discuss the use of code generation.

5.1.1 Dependencies

To get started using Querydsl, we must add Querydsl's HQL adapter, *querydsl-hql*, and the part of Querydsl which generates Querydsl query types for our models, *querydsl-apt*, as dependencies in our Maven configuration file,

named *pom.xml*. The relevant excerpt from the configuration can be seen in Listing 5.1.

```
<project [...]>
    [...]
    <dependencies>
        [...]
        <dependency>
            <groupId>com.mysema.querydsl</groupId>
            <artifactId>querydsl-hql</artifactId>
            <version>0.2.9-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>com.mysema.querydsl</groupId>
            <artifactId>querydsl-apt</artifactId>
            <version>0.2.9-SNAPSHOT</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Listing 5.1: Querydsl dependencies in Maven

The *SNAPSHOT* postfix on the version number means that we are not using the released version 0.2.9, but the latest development snapshot of Querydsl. The reason for this is that we ran into a couple of problems with version 0.2.9, which was resolved in the newer snapshot versions. The problems will be described in Section 5.4.

Querydsl 0.3.0, which includes the needed fixes, will according to the main Querydsl developer, Timo Westkämper, hopefully be released within the middle of February 2009.

5.1.2 Code Generation

Continuing with the Maven configuration, we have to define some settings for the Maven APT plug-in which performs the code generation, as shown in Listing 5.2. The plug-in needs to know in which Maven lifetime phase it should run, in this case *generate-sources*, where our DOM classes are located, what factory it should use for generating the sources, and where the generated sources should be saved.

The Querydsl APT factory, *com.mysema.query.apt.APTFactory*, which is a part of the earlier mentioned *querydsl-apt* Maven module, analyzes the standard JPA annotations on the models. Using the annotations it generates a Querydsl *PEntity* class—also known as a query type—for each of the models. These generated classes make it possible to construct queries using actual objects and object fields, instead of using *class* objects for the entities and strings for the fields of the entities, like the Criteria API does.

```
<project [...]>
  [...]
  <build>
    [...]
    <plugins>
      <plugin>
        <groupId>org.apache.myfaces.tobago</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <version>1.0.20</version>
        <executions>
          <execution>
            <phase>generate-sources</phase>
            <goals>
              <goal>execute</goal>
            </goals>
            <configuration>
              <force>true</force>
              <fork>false</fork>
              <factory>com.mysema.query.apt.APTFactory</factory>
              <generated>target/generated-sources/apt</generated
                  >
              <nocompile>true</nocompile>
              <A>
                -AdestPackage=no.jodal.query.playground.domain,
                -AdtoPackage=no.jodal.query.playground.domain,
                -AnamePrefix=Q
              </A>
              <showWarnings>true</showWarnings>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 5.2: Querydsl code generation configuration in Maven

When the configuration is in place, Maven will generate Querydsl query types for all models whenever needed. Since only the Java package containing the models are defined in the Maven configuration, we do not need to change the Maven configuration when changing or adding models. As long as they are annotated with the standard JPA annotations, Querydsl will have all the metadata needed for generating updated or new query types.

The generated query type for the playground project's Customer model is shown in Listing 5.3, and the model itself in Listing B.6 at page 63 in the appendices. Clearly, the generated query type is not complex, only containing two constructors and three fields: *id*, *name*, and *invoices*. Thus Querydsl's amount of source code generation should be acceptable even by

```
1  package no.jodal.query.playground.domain;
2
3  import com.mysema.query.grammar.types.*;
4
5  /**
6   * QCustomer is a Querydsl query type
7   *
8   */
9  public class QCustomer extends Path.PEntity<no.jodal.query.
       playground.domain.Customer>{
10     public final Path.PString name = _string("name");
11     public final Path.PNumber<java.lang.Long> id = _number("id",
           java.lang.Long.class);
12     public final Path.PEntityCollection<no.jodal.query.
           playground.domain.Invoice> invoices = _entitycol("
           invoices", no.jodal.query.playground.domain.Invoice.class
           , "Invoice");
13
14     public QCustomer(java.lang.String path) {
15         super(no.jodal.query.playground.domain.Customer.class, "
               Customer", path);
16     }
17     public QCustomer(PathMetadata<?> metadata) {
18         super(no.jodal.query.playground.domain.Customer.class, "
               Customer", metadata);
19     }
20 }
```

Listing 5.3: A generated query type for the Customer model

developers who are not in favor of the concept of code generating code. The query types can of course be created manually, but since all the information needed to create them already exists in the form of models and JPA annotations, using code generation to produce them removes the need for manual duplication of information. Such duplication would surely lead to unnecessary bugs introduced by human error, which is avoided by using code generation.

5.1.3 Compatibility with HQL and Criteria

By introducing Querydsl into an existing project which is already using HQL or Criteria, we only add some Maven configuration and new generated classes. Since nothing else is changed, there is no incompatibility between Querydsl and HQL or Criteria. This means that Querydsl can be used in parallel with other query solutions, and thus can be introduced incrementally to existing projects, replacing one query at the time, or just using Querydsl for new queries.

5.2 Querydsl in the Playground Project

The playground project was created to explore Hibernate, HQL, and Criteria, and their limitations for Chapter 2 and 3. E.g. all the examples in those chapters are created and tested in the context of the playground project.

5.2.1 Setup

The project consists of a simple DOM with three entities or models, *Customer*, *Invoice*, and *InvoiceLine* (see Listings B.6, B.7, B.8), which are related in the way illustrated by Figure 2.2 on page 6. A Data Access Object (DAO) interface, *InvoicingDao* (see Listing B.9), defines some methods which either serves as good examples or illustrates limitations of the Criteria API. Originally, we had two implementations of this interface, one using HQL and one using Criteria (see Listing C.3 and C.2, respectively).

A suite of unit tests for the entities (see Listings B.12, B.13, B.14) and the interface (see Listing B.11) has been developed independently of the interface implementations, and are thus reusable for all implementations. The test suite lets us be sure that the methods are implemented correctly and that they yield the same results independent of the query mechanism used in their method implementations.

To get to know Querydsl we configured the playground project as described in Section 5.1, and extended it with a Querydsl implementation in addition to the existing HQL and Criteria implementations.

5.2.2 Implementation

The *InvoicingDao* interface we are implementing using Querydsl defines five methods, of which four involves aggregate functions, three involves grouping, and two involves grouping with *having* clauses. The reason for this focus on grouping and aggregations is to explore the limitations of the Criteria API, and how other APIs handle those queries compared to Criteria. Using Criteria we were not able to implement the two queries needing a *having* clause at all.

Listing 5.4, 5.5, and 5.6 shows the same query implemented in HQL, Criteria and Querydsl. The query joins three entities, limits the result to rows matching the customer name, and sums the amount of all the invoice lines, returning the total sales generated by the given customer.

In the HQL version, the entire query is obviously a string. The Criteria version uses strings for all fields and aliases, and combinations thereof, while the Querydsl version only uses strings for the instantiation of query types, outside the query itself.

When considering the amount of code completion support from the

```
1  @SuppressWarnings("unchecked")
2  public int retrieveTotalSalesByCustomerName(String name) {
3      String query = "select sum(invoiceLine.amount) "
4          + "from Customer as customer "
5          + "inner join customer.invoices as invoice "
6          + "inner join invoice.invoiceLines as invoiceLine "
7          + "where customer.name = :name";
8      List results = getSession().createQuery(query).setString("
           name", name)
9          .list();
10     return ((Long) results.get(0)).intValue();
11 }
```

Listing 5.4: Total sales query using HQL

```
1  @SuppressWarnings("unchecked")
2  public int retrieveTotalSalesByCustomerName(String name) {
3      List results = getSession().createCriteria(Customer.class, "
           customer")
4          .add(Restrictions.eq("name", name))
5          .createAlias("invoices", "invoice")
6          .createAlias("invoice.invoiceLines", "invoiceLine")
7          .setProjection(Projections.projectionList()
8                  .add(Projections.sum("invoiceLine.amount"), "
                       totalSales"))
9          .list();
10     return ((Integer) results.get(0)).intValue();
11 }
```

Listing 5.5: Total sales query using Criteria

IDE, the HQL version only supports code completion on line 9 where the query string and method argument *name* are combined and the query executed. Code completion is better in the Criteria version, but does not restrict many invalid combinations because of the amount of strings still left in the query. E.g. adding a sum of an entity instead of a field, like *Projections.sum("invoiceLine")*, would be allowed by the IDE, but would fail at run-time. Finally, for Querydsl, strings are moved out of the query and the maximum amount of code completion is achieved. Instead of manually writing *createAlias("invoice.invoiceLines", "invoiceLine")* one can use code completion the whole way through *innerJoin(invoice.invoiceLines.as(invoiceLine)*, and immediately know with some certainty that the clause is valid since the IDE does not complain about any non-existing fields or methods, or type mismatches.

With regard to using Querydsl as a drop-in replacement for HQL or Criteria, Querydsl fits right in without any type conversions or data structure changes. Actually, Querydsl does a better job in following the

```
1  public int retrieveTotalSalesByCustomerName(String name) {
2      QCustomer customer = new QCustomer("customer");
3      QInvoice invoice = new QInvoice("invoice");
4      QInvoiceLine invoiceLine = new QInvoiceLine("invoiceLine");
5      List<Integer> results = from(customer)
6          .innerJoin(customer.invoices.as(invoice))
7          .innerJoin(invoice.invoiceLines.as(invoiceLine))
8          .where(customer.name.eq(name))
9          .list(sum(invoiceLine.amount).intValue());
10     return results.get(0);
11 }
12 private HqlQuery from(EEntity<?>... s) {
13     return new HqlQuery(getSession()).from(s);
14 }
```

Listing 5.6: Total sales query using Querydsl

defined interface than HQL and Criteria simply because its a newer API and thus uses Java generics throughout the implementation. While *list()* in the HQL and Criteria queries returns a List with unknown content which requires casting before use and an *@SuppressWarnings("unchecked")* annotation to avoid warnings from the IDE and compiler, Querydsl guarantees that the list contains Integer elements, which means that we do not need to manually cast elements taken from the list before use, and thus risk a run-time *ClassCastException*.

Finally, the Querydsl query matches the HQL query almost word for word. The major difference is that the Querydsl version consists of API calls and objects, and not a long string. The similarity with HQL also means that Querydsl got large similarities with SQL, which again means that many developers will understand the query without having any previous knowledge of Querydsl. This is very important, as most code is read much more than it is written.

5.3 Querydsl in LeanCast

As we this far have only tested Querydsl on artificial examples, it would be interesting to try Querydsl in an existing real-world project. Also, we need to test if it is possible to easily replace queries implemented in Criteria using Querydsl without changing the method's signature, and thus not affect the rest of the application.

LeanCast is an application developed for internal use at Iterate. At its core, LeanCast is a messaging platform where users can send each other messages using a web interface, an instant messaging client, or by sending SMS messages from a mobile phone. On top of this platform Iterate has added support for registering hours worked on a project by sending a simple

message to LeanCast. The entire project consists of about thirteen thousand lines of code (excluding empty lines), of which more than half are unit tests. Thus, LeanCast represents a moderately sized and complex project, and it has good enough test coverage for us to be confident that everything is still working as expected after having replaced the Criteria queries with Querydsl.

5.3.1 Setup

To test Querydsl on LeanCast, we extracted the parts of LeanCast which was needed to run the query tests. The most important classes are included in Appendix E starting at page 87. This includes the interface *PersistenceFacade* in Listing E.1, the original Criteria implementation of the interface, *HibernatePersistenceFacade*, in Listing E.2, and its unit tests, *HibernatePersistenceFacadeTest*, in Listing E.4. The unit tests are—in contradiction to its name—actually independent of the Hibernate implementation, and apply to any implementation of the *PersistenceFacade* interface.

After we separated the relevant parts of LeanCast, Querydsl was added as an dependency in LeanCast's Maven configuration, and the APT code generation plug-in was configured, as described in Section 5.1.

To reuse the parts of *HibernatePersistenceFacade* (HPF) which are specific to Hibernate but not specific to the query language and thus are irrelevant for our tests, like *save(Object)* and *delete(Object)*, *QuerydslPersistenceFacade* (QPF) was created as a subclass of HPF. All methods in HPF which involves Criteria queries, twelve in total, are overridden in QPF with implementations using Querydsl. QPF and the test adapter which runs the *PersistenceFacade* tests using QPF can be found in Listings E.3 and E.5.

5.3.2 Implementation

Already knowing the LeanCast domain models and how they related to each other, implementation of most of QPF was done by observing the name of the method, what arguments it accepted, and what type of object it returned. After implementing the query, relying heavily on previous knowledge of SQL and the code completion provided by the IDE, we ran the unit test suite, and in many cases one or more tests for the method passed on the first try, and we could continue the implementation based on the descriptive names of the unit tests that were still failing.

Even though this is a subjective experience, it says something about Querydsl's ease of use. Pouring through documentation on how to formulate your query, or searching the net for similar examples becomes unnecessary when you can leverage on previous knowledge of HQL or SQL, combined

with the help of code completion. Stating your intention with the query becomes the focus, and not how to use the query API.

```
1  @SuppressWarnings("unchecked")
2  public List<Message> retrieveConversation(User user1, User user2
       ) {
3      Session session = getHibernateSession();
4      Criteria criteria = session.createCriteria(Message.class);
5      criteria.setResultTransformer(CriteriaSpecification.
           DISTINCT_ROOT_ENTITY);
6      criteria.createAlias("receivers", "r");
7      criteria.add(Restrictions.or(
8          Restrictions.and(Restrictions.eq("sender", user1),
9              Restrictions.eq("r.id", user2.getId())),
10         Restrictions.and(Restrictions.eq("sender", user2),
11             Restrictions.eq("r.id", user1.getId()))));
12     List<Message> list = criteria.list();
13     return list;
14 }
```

Listing 5.7: Original Criteria query

```
1  public List<Message> retrieveConversation(User user1, User user2
       ) {
2      QMessage message = new QMessage("message");
3      return from(message)
4          .where((message.sender.eq(user1)
5                  .and(message.receivers.contains(user2)))
6              .or(message.sender.eq(user2)
7                  .and(message.receivers.contains(user1))))
8          .list(message);
9  }
```

Listing 5.8: Same query rewritten using Querydsl

To further illustrate the point, compare the readability of Listing 5.7 and 5.8. Everything in Section 5.2.2 about writing queries for the playground project still applies here. It is easy to read, and to write. It is type safe, string free, and replaces Criteria without changing the interface, or adjusting argument or return values. And it is concise.

There is almost nothing to remove from the Querydsl version of the method. It has a very high signal-to-noise ratio, in contrast to the original Criteria version. The query type instance, *message*, could have been instantiated at class level and been reused across multiple methods and queries. And the *list()* method, when given no arguments, could have reused the arguments given to *from()*. Thus, *from(message).list()* would return a list of all messages. Requiring *from(message).list(message)* seems a bit redundant for the not too uncommon case where the projection should

include every relation in the query.

5.4 Problems

During the writing of Querydsl queries for the playground project and LeanCast we ran into some problems with Querydsl. The problems and their solutions are summarized in the following sections.

5.4.1 *sum(Integer)* results in a *Long*

In Querydsl 0.2.9, when using the *sum()* aggregate function on an *Integer* field as the result of the query, Querydsl claimed to return a *List<Integer>*, just as one would expect when taking the sum of some integers. Though, the unit test failed with a run-time *ClassCastException* on the statement *return result.get(0);*. The returned list actually contained *Long* objects, and the use of generics implicitly casted the list element to an *Integer* when calling *get(0)*.

```
1 List results = from(customer)
2     .innerJoin(customer.invoices.as(invoice))
3     .innerJoin(invoice.invoiceLines.as(invoiceLine))
4     .where(customer.name.eq(name))
5     .list(sum(invoiceLine.amount));
6 return ((Long) result.get(0)).intValue();
```

Listing 5.9: Workaround for Querydsl sum() bug

The workaround, as seen in Listing 5.9, was to remove all use of generics in the query, i.e. assign the result to a *List* without any information about its content. Then, on the line of the return statement explicitly cast the result to a *Long* object before finally using *Long.intValue()* to get an integer, which the method signature required for its return value.

Returning *Long* objects when taking the sum of *Integer* fields are actually the right behaviour according to the JPA standard, and Hibernate does this correctly [8]. The problem was that Querydsl's propagation of type information from the *list()* call to the actual return value did not change the return type from *Integer* to *Long*. The problem was reported to the Querydsl developers through their issue tracker at [17], and within a couple of rounds of fixes from Timo Westkämper, the Querydsl project leader, and testing from our side, the bug was finally resolved two days later. The final version of the query again uses Java generics and is type safe, as seen in Listing 5.6 back on page 35.

5.4.2 *java.util.Time* fields does not work

The next problem surfaced when generating the query types for the LeanCast project. One of the LeanCast models contained a *java.sql.Time* field for storing time without date information. *Time* is a subclass of *java.util.Date* and implements the interface *Comparable<Date>*, just like *Date* does.

While *Date* fields worked flawlessly, the *Time* field in the generated query type resulted in the compile-time error *Bound mismatch: The type Time is not a valid substitute for the bounded parameter <D extends Comparable<D>>of the type Path<C>.PComparable<D>"*. In other words, Querydsl required any type used by *PComparable* to be comparable with itself, and nothing else.

No easy workaround for this problem was found, except not using *Time* fields in the models, which could possibly require unwanted changes to the application. Instead, we looked for how to fix Querydsl by applying the *PECS* mnemonic from [27, Item 28] on the signature of the *PComparable* class. *PECS* is short for *producer extends, consumer super*. Since a comparator *consumes* objects, the signature could be extended from *<D extends Comparable<D>>* to the more flexible *<D extends Comparable<? super D>>*, which would allow *Time* to be comparable by any superclass of itself, including *Date*.

This problem and the analysis was submitted to the Querydsl issue tracker at [18]. A fix following the direction of our analysis was applied to the development version of Querydsl less than three hours later.

5.4.3 Minor issues

Additionally, two minor issues was submitted to Querydsl's issue tracker. One of them related to the generated query types, which had the same two imports in all the generated source code files, and in some cases one of them was not in use. Unused imports normally results in warnings both from the IDE and from the Java compiler, even though the import is of no obstruction. This was reported as an minor issue at [20] and was also fixed less than three hours later.

The last issue was a minor feature request. When constructing queries using Querydsl, one could compare numeric types using methods for both *less than*, *equal*, and *less than or equal*, and analogously for *greater than*. However, when querying date and time fields, the methods *before()*, *after()*, and *eq()* was available, but no shortcut for the combination of *eq()* with each of the two other methods. This felt like an inconsistency in the API, thus issue [19] was submitted to propose the addition of these two new methods. Parallel with the fixing of the unused import issue, the new methods was added to the development version less than three hours later.

5.4.4 Documentation

Querydsl's documentation is somewhat limited. The project wiki contains a getting starting guide for how to setup code generation and some query examples. Using this and code completion in the IDE, it is possible to figure out enough to make use of Querydsl, but improved and more complete documentation would make it easier to recommend Querydsl for adoption in existing projects. There have been done some work on improving the API's Javadoc recently, which is useful as Javadoc is easy accessible through the IDE during development. The scope of a query API is limited, and as such the need for extensive documentation is also limited when compared to more complex software components, like Hibernate itself.

5.5 Summary

We have applied Querydsl to the artificial playground project and to a real-world application, LeanCast. Querydsl has been easy to get running, and easy to write compact and readable queries in. Integration in an existing project has been as painless as it could be, and we have had almost no problems. Where we had problems, they were resolved in manner of hours by the Querydsl developers, which both seem dedicated and got a company backing the development. The API is complete, making all queries we have tried to express possible. When we had a request for a simplifying addition to the API, it was willingly accepted.

Chapter 6

Django and JPA2

After finding a solution to the limitations of HQL and Hibernate's Criteria API, the problem description requests that the solution is compared with how Django approaches the same problems. In this chapter we will discuss Querydsl in relation to queries in Django, and the future of queries in Java with the next JPA standard, which we became aware of during our work.

6.1 Django

Django [3] is a web development framework for the Python [14] programming language. It was originally conceived as an internal project of World Online, the web development section of the newspaper house Lawrence Journal-World in Kansas, US. In July 2005, the project was publicly released under the 3-clause BSD open-source license, and in June 2008 a non-profit organization, the Django Software Foundation was launched to promote and support the project financially. In the time since its initial public release, Django has grown increasingly popular both as a good web development framework promoting web standards and best practices, and as the Python language's alternative to the Ruby framework Ruby on Rails [21]. This without imitating Rails, like other popular frameworks such as Grails and CakePHP have done.

The Django framework is a collection of loosely coupled components, including a template language, a drop-in administration interface, multiple cache back-ends, internationalization support, form handling, and an ORM with a query API.

6.1.1 Queries in Django

The Django ORM and query API supports much of what one would expect from a general purpose ORM, but not everything. E.g. Django, which is currently in version 1.0.2, will not support aggregates before version 1.1 [4],

due for release in March 2009, almost four years after the Django project was made public. Because Django primarily targets web development, it has been possible for the Django developers to keep things a bit simpler for the most common use cases in a web development setting. Listing 6.1 uses Django's query API to implement the same functionality as we did with HQL and Criteria back in Listing 2.3 and 2.4.

```
1 def search_for_customer(name):
2     return Customer.objects.filter(name=name)
```

Listing 6.1: Example of Django ORM usage

As you can see, there is no explicit session object in play here. The starting point for the query is the uninstantiated class *Customer* and its static (in Java terminology) field *objects* as we want to retrieve instantiated objects of the class *Customer*. If we would like to retrieve all objects, we can simply replace *filter(name='Alice')* with *all()*.

In the call to *filter()* we use a very convenient Python feature called *keyword arguments* or *kwargs* for short [32]. Keyword arguments allow functions to accept arbitrary name-value pairs as arguments. These names and values are, as Python is a dynamic language, evaluated at run-time, which makes their use very flexible, and the Django query API uses them extensively as can be seen with a few examples in Listing 6.2.

```
1  # All invoice lines with an amount greater than 1000
2  # Translated to "amount > 1000" in SQL
3  InvoiceLine.objects.filter(amount__gt=1000)
4
5  # All customers with names which start with 'Alice'
6  # Translated to "name LIKE 'Alice%'" in SQL
7  Customer.objects.filter(name__startswith='Alice')
8
9  # All customers with an IDs in the list
10 # Translated to "id IN (1, 2, 3)" in SQL
11 Customer.objects.filter(id__in=[1, 2, 3])
12
13 # All invoices for customers whose name starts with 'Alice'
14 Invoice.objects.filter(customer__name__startswith='Alice')
15
16 # Traversing through customer 1 to his invoices
17 Customer.object.get(id=1).invoice_set.all()
```

Listing 6.2: Django field operations using keyword arguments

Two succeeding underscore characters in the kwargs name are used much in the same way as dot in normal object traversal. In most of the examples of Listing 6.2 they are used to separate data field names and operator names. While in the second to last example, *customer__name__startswith*, they are

actually used to walk the path from the *Invoice* class, through *customer*, its foreign key to the *Customer* class, and check if the customer's name starts with "Alice". This query is implicitly doing a join operation between *Invoice* and *Customer*, before filtering on the customer name. Instead of expressing yourself in joins and relational database operations, you use keyword arguments just as if you where traversing the real objects.

In the last example, a similar result is achieved by first getting the customer with ID 1. Then on the returned *Customer* instance, use the automatically added *invoice_set* field's *all()* method. This is analogous with the *all()* method on the *objects* field in some of the other examples.

6.1.2 Comparison with Querydsl

As the API properties considered good and important are widely different in the world of Python versus the world of Java, it has no purpose to compare Django queries with Querydsl queries to any great extent. Java is a compiled language with static typing normally developed in a large IDE, while Python is an interpreted language with dynamic typing usually developed using a simple text editor with rapid prototyping in an interactive shell which can execute Python statements.

In Java, refactoring is done with large amounts of help from the IDE, thus our focus on avoiding strings and using real objects wherever possible. In Python, changing code is mostly done by hand, thus using keyword arguments—which are seen by the called function as pairs of key strings and values—is not inferior to keeping to objects. In fact, using keyword arguments is usually regarded as a good practice in a Python context, since it helps explain to the reader what the arguments passed to a method are.

While Querydsl draws on SQL and HQL, Django's query API creates an abstraction layer which has few similarities with SQL. The major commonality between Querydsl and Django is that both APIs can be said to have a high signal-to-noise ratio and readability. They are both good query APIs, but in different environments.

6.2 Java Persistence API 2.0

During the work on this thesis we became aware of the currently ongoing work on Java Specification Request 317, which specifies Java Persistence API 2.0 [25, 24]. The latest development was the release of a version for public review November 14, 2008.

With regard to queries, the most interesting part is the inclusion of a Criteria API in the JPA2 standard, which is similar to Hibernate's Criteria API. Compared to Hibernate Criteria, JPA2 Criteria is quite different and addresses several of the requirements from Chapter 3, confirming that our problems are real.

```
1  EntityManager em = ...; // Instead of sessions in Hibernate
2  QueryBuilder queryBuilder = em.getQueryBuilder();
3
4  @SupressWarnings("unchecked")
5  public List<Customer> searchForCustomer(String name) {
6      DomainObject customer = queryBuilder.createQueryDefinition(
           Customer.class);
7      Query q = em.createQuery(customer
8          .where(customer.get("name").equal(name)));
9      return q.getResultList();
10 }
11
12 @SupressWarnings("unchecked")
13 public List<Object[]> retrieveInvoicesAndTotalAmounts() {
14     DomainObject invoice = queryBuilder.createQueryDefinition(
           Invoice.class);
15     DomainObject invoiceLine = invoice.join("invoiceLines");
16     Query q = em.createQuery(invoice
17         .select(invoice, invoiceLine.get("amount").sum()));
18     return q.getResultList();
19 }
```

Listing 6.3: Queries using the JPA2 Criteria API

Listing 6.3 shows two methods we have used earlier in this work, this time implemented using the new JPA2 Criteria API, as it is proposed in the public review version of JSR 317.

There are mainly three improvements in JPA2 Criteria over Hibernate Criteria. First, it supports everything supported by JPQL, thus the lack of e.g. *having* clauses, as we mentioned in Section 3.1.4, are fixed. Second, the vocabulary of SQL/JPQL/HQL is used instead of the relational algebra vocabulary used by Hibernate Criteria. Third, the amount of strings are reduced to only single field names, meaning no more strings along the lines of *"alias.relatedModel.field"*.

Some problems still remain. Because the use of strings is not entirely eliminated, the API is still not typesafe, which is obvious from the *@SupressWarnings("unchecked")* annotations in the listing. The signal-to-noise ratio is still quite low. The query itself has become better by removing the extensive use of factory methods, but the surrounding lines of code needed to implement these methods have an even lower signal-to-noise ratio than queries implemented with Hibernate Criteria.

In summary, JPA2 seems promising, and as it still is not finalized, additional improvements to the Criteria API are still possible. If large changes, such as getting rid of the last remaining strings and attain type safety, will happen before JPA2 is published in its final form remains to see.

Chapter 7

Discussion

When the problem description was given and we started working on this thesis, we depicted that our contribution could be in the form of improvements to the Hibernate Criteria API which we could contribute back to the Hibernate project, or a new query API which would work as a wrapper around HQL or Criteria, providing the properties we wanted from a query API.

When looking closer at the limitations with today's solutions in Criteria and HQL, we realized that incremental improvements to the Criteria API could only introduce the missing technical features. Improving the usability aspects of the Criteria API—which would include removing the use of strings, making SQL knowledge more applicable, and increasing the signal-to-noise ratio of the queries—would not be possible without breaking backwards compatibility for existing users of the Criteria API.

For the option of creating a new query API on top of Hibernate, we discovered several existing APIs which seemed to do approximately what we wanted. We looked at how the .NET framework has solved this with LINQ, but concluded that the approach could not easily be applied to Java. We studied five query APIs with various properties. All was inspired by SQL, except JaQue which looked to LINQ for inspiration and used a prototype of closures, which probably will be included as a new language feature in Java 7. In addition to JaQue, Quaere and Querydsl had existing or emerging support for JPA, and thus support for Hibernate. Quaere was a step in the right direction with regards to readability, but still used strings heavily and thus had poor support for both code completion and refactoring. Querydsl scored well on all requirements we defined.

Querydsl's syntax reminisce SQL and is almost identical with HQL. Thus, queries written in Querydsl can be understood by many developers without previous experience with Querydsl, and large and complex queries remains readable to a larger degree than queries formulated with other APIs.

At the same time, Querydsl is both free of strings and type safe,

which has many benefits. First, invalid queries fail fast. Since the query is static, i.e. entirely made out of objects and method invocations, most invalid queries will fail at compile-time, and, when developing in an IDE, the developer gets immediate feedback when introducing syntactic errors. Second, the developer can use the code completion feature of her IDE to rapidly construct the query and to explore what operations are available without referring to documentation. Third, when the query is static, the IDE can automatically and safely modify the query when e.g. the name of an entity or field is changed, making refactoring a native operation.

To learn more about Querydsl and to verify that it actually is viable for use in existing real-world projects, we applied it in two different settings. The first was a test project originally created for identifying limitations of creating queries with HQL and Criteria. Since Iterate's consultants had experienced problems with expressing queries which involved aggregates and grouping using Criteria, the test project focused on that area. The second experiment involved rewriting all the Criteria queries in Iterate's internal messaging platform, LeanCast, using Querydsl. The queries in LeanCast were generally simpler than the artificial queries of the playground project, but they were also more diverse, and keeping the interface with the rest of the application unchanged was critical for a smooth transition from Criteria to Querydsl. In both settings, having a high percentage of the code covered by unit test was of immense help to build confidence that the new queries both returned the correct results and kept to the existing interface.

During the experiments with Querydsl we met some technical difficulties. In the one case where a simple work around was not possible we proposed a change to the Querydsl developers, which implemented the change in the development version in a matter of hours. The rest of our problems was also reported as issues to the Querydsl developers, who were very responsive and promptly implemented the needed changes.

Querydsl is reasonably easy to get started with, and integrates well into existing projects which have been using HQL or the Criteria API. The Querydsl project is active and is continually making improvements to the software. The developers are very responsive and handle issues rapidly, while being open to new ideas for improvement. The project is also backed by a software development consultancy which can provide commercial support if needed.

The Querydsl API seems mature enough to be a helpful tool in any Hibernate project, and should thus be considered for use in both new and existing projects.

Finally, we briefly compared Querydsl to queries in the web development framework Django, and looked at what the new JPA2 standard will bring to the area of constructing queries in Java.

Chapter 8

Conclusion

In this thesis we have identified the limitations of the two most commonly used approaches for constructing queries in software development projects that use Hibernate.

We have shown that Querydsl provide solutions to the identified limitations, and we recommend Querydsl as a viable alternative to HQL and Hibernate's Criteria API for use in both new and existing projects.

We have briefly compared Querydsl to queries in the web development framework Django, but because of the difference in nature between Python and Java environments, we concluded that this had little meaning.

Our work provides a solution that can make developers more effective while producing better code with less defects. Querydsl is easily integrated with existing projects, and can be used in addition to HQL and Criteria allowing for an incremental introduction.

8.1 Future Work

For future work, it would be interesting to empirically confirm that Querydsl is an improvement over the Criteria API and HQL. This could be done by collecting experiences from projects which are using Querydsl, or by using Querydsl in a larger project—with more complex domain object models and queries—and look at what benefits and problems the project teams experiences.

Looking a bit wider, a closer look at JPA2, and how it will change the area of persistence in Java, is a potential path to take. Also, queries against in-memory collections are a related area to look into.

Bibliography

[1] Apache Maven Project. http://maven.apache.org/.

[2] Closures for the Java Programming Language. Retrieved from http://www.javac.info/ at 2009-02-02.

[3] Django project. http://www.djangoproject.com/.

[4] Django ticket #3566: ORM aggregation support. Retrieved from http://code.djangoproject.com/ticket/3566 at 2009-01-11.

[5] Eclipse IDE. http://www.eclipse.org/.

[6] Hibernate. http://www.hibernate.org/.

[7] Hibernate issue tracker: [#HHH-1043] Added HAVING Support to Criteria. Retrieved from http://opensource.atlassian.com/projects/hibernate/browse/HHH-1043 at 2009-01-19.

[8] Hibernate issue tracker: [#HHH-1538] aggregations functions in EJBQL queries does not return the appropriate types. Retrieved from http://opensource.atlassian.com/projects/hibernate/browse/HHH-1538 at 2009-02-07.

[9] IntelliJ IDEA. http://www.jetbrains.com/idea/.

[10] JaQu. Retrieved from http://h2database.com/html/jaqu.html at 2009-01-22.

[11] JaQue. Retrieved from http://code.google.com/p/jaque/ at 2009-01-22.

[12] Java Persistence API. Retrieved from http://java.sun.com/javaee/technologies/persistence.jsp at 2009-01-14.

[13] NetBeans IDE. http://www.netbeans.org/.

[14] Python programming language. http://www.python.org/.

[15] Quaere. Retrieved from http://quaere.codehaus.org/ at 2009-01-22.

[16] Querydsl. Retrieved from `http://source.mysema.com/display/querydsl/Querydsl` at 2009-01-22.

[17] Querydsl bug #326650: HqlGrammar.sum() on an Integer field should return a Long. Retrieved from `https://bugs.launchpad.net/querydsl/+bug/326650` at 2009-02-10.

[18] Querydsl bug #327113: java.sql.Time fields does not work. Retrieved from `https://bugs.launchpad.net/querydsl/+bug/327113` at 2009-02-09.

[19] Querydsl bug #327552: Add PComparable<Date>.boe() and .aoe(). Retrieved from `https://bugs.launchpad.net/querydsl/+bug/327552` at 2009-02-10.

[20] Querydsl bug #327555: Unused import in generated classes. Retrieved from `https://bugs.launchpad.net/querydsl/+bug/327555` at 2009-02-10.

[21] Ruby on Rails framework. `http://www.rubyonrails.org/`.

[22] Squill. Retrieved from `https://squill.dev.java.net/` at 2009-01-22.

[23] Java Specification Request 220: Enterprise JavaBeans 3.0. Retrieved from `http://www.jcp.org/en/jsr/detail?id=220` at 2009-01-14, May 2006.

[24] Java Specification Request: Java Persistence 2.0, Public Review. Retrieved from `http://jcp.org/en/jsr/detail?id=317` at 2009-01-02, November 2008.

[25] Linda DeMichiel's Blog: Java Persistence 2.0 Public Draft: Criteria API. Retrieved from `http://blogs.sun.com/ldemichiel/entry/java\_persistence\_2\_0\_public1` at 2009-01-02, November 2008.

[26] Mike Andrews and James A. Whittaker. *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley Professional, February 2006.

[27] Joshua Bloch. *Effective Java*. Prentice Hall PTR, second edition, May 2008.

[28] Don Box and Anders Hejlsberg. LINQ: .NET Language-Integrated Query. Retrieved from `http://msdn.microsoft.com/en-us/library/bb308959.aspx` at 2009-01-16, February 2007.

[29] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, first edition, October 2001.

[30] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, October 1999.

[31] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, August 2008.

[32] Python Software Foundation. *The Python Tutorial (Python v2.6.1)*, January 2009.

[33] Red Hat, Inc. *Hibernate Reference Documentation v3.3.1*, 2008.

[34] James Shore and Shane Warden. *The Art of Agile Development*. O'Reilly Media, Inc., October 2007.

[35] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, March 2004.

Appendices

Appendix A

Playground Project

The playground project is used to test various query methods and APIs—including HQL and Criteria—on the same set of example domain object models.

This appendix contains the main Maven configuration for the main or mother project, i.e. the *pom.xml* that defines the dependency configuration which is common between the subprojects.

playground/

playground/playground-core/
: Common framework. See Appendix B.

playground/playground-hibernate/
: Criteria and HQL implementations. See Appendix C.

playground/playground-querydsl/
: Querydsl implementation. See Appendix D.

pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
       www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
         apache.org/maven-v4_0_0.xsd">
3
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>no.jodal.query</groupId>
6    <artifactId>playground</artifactId>
7    <version>1.0-SNAPSHOT</version>
8    <packaging>pom</packaging>
9    <name>Playground</name>
10
11   <organization>
12     <name>Stein Magnus Jodal</name>
13   </organization>
```

```
14
15  <modules>
16    <module>playground-core</module>
17    <module>playground-hibernate</module>
18    <module>playground-querydsl</module>
19  </modules>
20
21  <repositories>
22    <repository>
23      <id>jboss.com.maven.2</id>
24      <url>http://repository.jboss.com/maven2
25      </url>
26    </repository>
27    <repository>
28      <id>java.net.maven.2</id>
29      <url>http://download.java.net/maven/2
30      </url>
31    </repository>
32    <repository>
33      <id>java.net.maven.1</id>
34      <url>http://download.java.net/maven/1
35      </url>
36      <layout>legacy</layout>
37    </repository>
38    <repository>
39      <id>mysema.com.maven.2.releases</id>
40      <url>http://source.mysema.com/maven2/releases
41      </url>
42    </repository>
43    <repository>
44      <id>mysema.com.maven.2.snapshots</id>
45      <url>http://source.mysema.com/maven2/snapshots
46      </url>
47    </repository>
48  </repositories>
49
50  <dependencies>
51    <dependency>
52      <groupId>junit</groupId>
53      <artifactId>junit</artifactId>
54      <version>4.5</version>
55    </dependency>
56  </dependencies>
57
58  <dependencyManagement>
59    <dependencies>
60      <dependency>
61        <groupId>no.jodal.query</groupId>
62        <artifactId>playground-core</artifactId>
63        <version>${project.version}</version>
64      </dependency>
65
66      <dependency>
67        <groupId>no.jodal.query</groupId>
68        <artifactId>playground-hibernate</artifactId>
69        <version>${project.version}</version>
70      </dependency>
71
72      <dependency>
73        <groupId>no.jodal.query</groupId>
74        <artifactId>playground-querydsl</artifactId>
75        <version>${project.version}</version>
```

```
76        </dependency>
77
78        <dependency>
79          <groupId>org.hibernate</groupId>
80          <artifactId>hibernate</artifactId>
81          <version>3.2.6.ga</version>
82        </dependency>
83
84        <dependency>
85          <groupId>org.hibernate</groupId>
86          <artifactId>hibernate-annotations</artifactId>
87          <version>3.3.1.GA</version>
88        </dependency>
89      </dependencies>
90    </dependencyManagement>
91
92    <build>
93      <plugins>
94        <plugin>
95          <groupId>org.apache.maven.plugins</groupId>
96          <artifactId>maven-compiler-plugin</artifactId>
97          <configuration>
98            <source>1.5</source>
99            <target>1.5</target>
100         </configuration>
101       </plugin>
102
103       <plugin>
104         <artifactId>maven-eclipse-plugin</artifactId>
105         <configuration>
106           <downloadJavadocs>true</downloadJavadocs>
107         </configuration>
108       </plugin>
109     </plugins>
110   </build>
111 </project>
```

Listing A.1: pom.xml

Appendix B

Playground Core Project

This appendix contains the core playground project, which includes all the parts the other subprojects have in common.

B.1 playground-core/

pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
       www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
         apache.org/maven-v4_0_0.xsd">
3
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>no.jodal.query</groupId>
6    <artifactId>playground-core</artifactId>
7    <version>1.0-SNAPSHOT</version>
8    <packaging>jar</packaging>
9    <name>Playground Core</name>
10
11   <organization>
12     <name>Stein Magnus Jodal</name>
13   </organization>
14
15   <parent>
16     <groupId>no.jodal.query</groupId>
17     <artifactId>playground</artifactId>
18     <version>1.0-SNAPSHOT</version>
19   </parent>
20
21   <dependencies>
22     <dependency>
23       <groupId>javax.persistence</groupId>
24       <artifactId>persistence-api</artifactId>
25       <version>1.0</version>
26     </dependency>
27
28     <dependency>
29       <groupId>org.springframework</groupId>
30       <artifactId>spring</artifactId>
31       <version>2.5.6</version>
```

```
32      </dependency>
33
34      <dependency>
35        <groupId>org.hibernate</groupId>
36        <artifactId>hibernate</artifactId>
37      </dependency>
38
39      <dependency>
40        <groupId>log4j</groupId>
41        <artifactId>log4j</artifactId>
42        <version>1.2.14</version>
43      </dependency>
44
45      <dependency>
46        <groupId>hsqldb</groupId>
47        <artifactId>hsqldb</artifactId>
48        <version>1.8.0.7</version>
49        <scope>runtime</scope>
50      </dependency>
51
52      <dependency>
53        <groupId>org.hibernate</groupId>
54        <artifactId>hibernate-annotations</artifactId>
55        <scope>runtime</scope>
56      </dependency>
57
58      <dependency>
59        <groupId>com.mysema.querydsl</groupId>
60        <artifactId>querydsl-hql</artifactId>
61        <version>0.2.9-SNAPSHOT</version>
62      </dependency>
63
64      <dependency>
65        <groupId>com.mysema.querydsl</groupId>
66        <artifactId>querydsl-apt</artifactId>
67        <version>0.2.9-SNAPSHOT</version>
68        <scope>provided</scope>
69      </dependency>
70    </dependencies>
71
72    <build>
73      <plugins>
74        <plugin>
75          <groupId>org.apache.myfaces.tobago
76          </groupId>
77          <artifactId>maven-apt-plugin</artifactId>
78          <version>1.0.20</version>
79          <executions>
80            <execution>
81              <phase>generate-sources</phase>
82              <goals>
83                <goal>execute</goal>
84              </goals>
85              <configuration>
86                <force>true</force>
87                <fork>false</fork>
88                <factory>
89                  com.mysema.query.apt.APTFactory
90                </factory>
91                <generated>
92                  target/generated-sources/apt
93                </generated>
```

```
94                <nocompile>true</nocompile>
95                <A>
96                  -AdestPackage=no.jodal.query.playground.domain,
97                  -AdtoPackage=no.jodal.query.playground.domain,
98                  -ANamePrefix=Q
99                </A>
100               <showWarnings>true</showWarnings>
101               <verbose>false</verbose>
102             </configuration>
103           </execution>
104         </executions>
105       </plugin>
106     </plugins>
107   </build>
108 </project>
```

Listing B.1: pom.xml

B.2 playground-core/src/main/resources/

applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
3      "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
4
5  <beans>
6    <bean id="propertyPlaceholderConfigurer"
7      class="org.springframework.beans.factory.config.
           PropertyPlaceholderConfigurer">
8      <property name="location" value="classpath:database.properties"></
           property>
9    </bean>
10   <bean id="dataSource"
11     class="org.springframework.jdbc.datasource.DriverManagerDataSource
         ">
12     <property name="driverClassName" value="${database.driverClassName
         }" />
13     <property name="url" value="${database.url}" />
14     <property name="username" value="${database.username}" />
15     <property name="password" value="${database.password}" />
16   </bean>
17   <bean id="sessionFactory"
18     class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
         >
19     <property name="dataSource">
20       <ref bean="dataSource" />
21     </property>
22     <property name="configLocation">
23       <value>classpath:hibernate.cfg.xml</value>
24     </property>
25     <property name="configurationClass">
26       <value>org.hibernate.cfg.AnnotationConfiguration</value>
27     </property>
28     <property name="hibernateProperties">
29       <props>
30         <prop key="hibernate.dialect">${database.dialect}</prop>
31       </props>
32     </property>
```

```
33   </bean>
34 </beans>
```

Listing B.2: applicationContext.xml

database.properties

```
1 # In-memory Java database
2 database.driverClassName=org.hsqldb.jdbcDriver
3 database.url=jdbc:hsqldb:mem:playground
4 database.username=sa
5 database.password=
6 database.dialect=org.hibernate.dialect.HSQLDialect
```

Listing B.3: database.properties

hibernate.cfg.xml

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <!DOCTYPE hibernate-configuration PUBLIC
3          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.
               dtd">
5
6  <hibernate-configuration>
7    <session-factory>
8      <property name="hibernate.connection.pool_size">1</property>
9      <property name="show_sql">false</property>
10     <property name="hibernate.hbm2ddl.auto">create</property>
11
12     <!-- Mapping using annotations -->
13     <mapping class="no.jodal.query.playground.domain.Customer" />
14     <mapping class="no.jodal.query.playground.domain.Invoice" />
15     <mapping class="no.jodal.query.playground.domain.InvoiceLine" />
16   </session-factory>
17 </hibernate-configuration>
```

Listing B.4: hibernate.cfg.xml

log4j.properties

```
1  # Loggers
2  log4j.rootLogger=DEBUG, Console
3
4  log4j.logger.org.hibernate=WARN
5  log4j.logger.org.springframework=WARN
6  log4j.logger.com.mysema.query=WARN
7
8  # Console logging
9  log4j.appender.Console=org.apache.log4j.ConsoleAppender
10 log4j.appender.Console.layout=org.apache.log4j.PatternLayout
11 log4j.appender.Console.layout.ConversionPattern=%-4r [%t] %-5p %c %x -
       %m%n
```

Listing B.5: log4j.properties

B.3 playground-core/src/main/java/

B.3.1 .../no/jodal/query/playground/domain/

Customer.java

```
1  package no.jodal.query.playground.domain;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  import javax.persistence.CascadeType;
7  import javax.persistence.Entity;
8  import javax.persistence.GeneratedValue;
9  import javax.persistence.GenerationType;
10 import javax.persistence.Id;
11 import javax.persistence.OneToMany;
12 import javax.persistence.Table;
13
14 /**
15  * Customer domain model
16  *
17  * @author Stein Magnus Jodal (Master's Thesis)
18  */
19 @Entity
20 @Table(name = "customer")
21 public class Customer {
22   @Id
23   @GeneratedValue(strategy = GenerationType.AUTO)
24   private Long id;
25
26   private String name;
27
28   @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
29   private Set<Invoice> invoices;
30
31   public void addInvoice(Invoice invoice) {
32     if (invoices == null) {
33       invoices = new HashSet<Invoice>();
34     }
35     invoices.add(invoice);
36     invoice.setCustomer(this);
37   }
38
39   public Long getId() {
40     return id;
41   }
42
43   public Set<Invoice> getInvoices() {
44     if (invoices == null) {
45       return new HashSet<Invoice>();
46     } else {
47       return invoices;
48     }
49   }
50
51   public String getName() {
52     return name;
53   }
54
55   public void setId(Long id) {
```

```
56      this.id = id;
57    }
58
59    public void setInvoices(Set<Invoice> invoices) {
60      this.invoices = invoices;
61    }
62
63    public void setName(String name) {
64      this.name = name;
65    }
66
67    @Override
68    public String toString() {
69      return "[Customer: id=" + getId() + ", name=" + getName() + "]";
70    }
71  }
```

Listing B.6: Customer.java

Invoice.java

```
1   package no.jodal.query.playground.domain;
2
3   import java.util.HashSet;
4   import java.util.Set;
5
6   import javax.persistence.CascadeType;
7   import javax.persistence.Entity;
8   import javax.persistence.GeneratedValue;
9   import javax.persistence.GenerationType;
10  import javax.persistence.Id;
11  import javax.persistence.ManyToOne;
12  import javax.persistence.OneToMany;
13  import javax.persistence.Table;
14
15  /**
16   * Invoice domain model
17   *
18   * @author Stein Magnus Jodal (Master's Thesis)
19   */
20  @Entity
21  @Table(name = "invoice")
22  public class Invoice {
23    @Id
24    @GeneratedValue(strategy = GenerationType.AUTO)
25    private Long id;
26
27    @ManyToOne
28    private Customer customer;
29
30    @OneToMany(mappedBy = "invoice", cascade = CascadeType.ALL)
31    private Set<InvoiceLine> invoiceLines;
32
33    public void addInvoiceLine(InvoiceLine invoiceLine) {
34      if (invoiceLines == null) {
35        invoiceLines = new HashSet<InvoiceLine>();
36      }
37      invoiceLines.add(invoiceLine);
38      invoiceLine.setInvoice(this);
39    }
```

```
40
41   public Customer getCustomer() {
42     return customer;
43   }
44
45   public Long getId() {
46     return id;
47   }
48
49   public Set<InvoiceLine> getInvoiceLines() {
50     if (invoiceLines == null) {
51       return new HashSet<InvoiceLine>();
52     } else {
53       return invoiceLines;
54     }
55   }
56
57   public void setCustomer(Customer customer) {
58     this.customer = customer;
59   }
60
61   public void setId(Long id) {
62     this.id = id;
63   }
64
65   public void setInvoiceLines(Set<InvoiceLine> lines) {
66     this.invoiceLines = lines;
67   }
68
69   @Override
70   public String toString() {
71     return "[Invoice: id=" + getId() + ", customer=" + getCustomer()
72         + ", numLines=" + getInvoiceLines().size() + "]";
73   }
74 }
```

Listing B.7: Invoice.java

InvoiceLine.java

```
1  package no.jodal.query.playground.domain;
2
3  import javax.persistence.Entity;
4  import javax.persistence.GeneratedValue;
5  import javax.persistence.GenerationType;
6  import javax.persistence.Id;
7  import javax.persistence.ManyToOne;
8  import javax.persistence.Table;
9
10 /**
11  * InvoiceLine domain model
12  *
13  * @author Stein Magnus Jodal (Master's Thesis)
14  */
15 @Entity
16 @Table(name = "invoice_line")
17 public class InvoiceLine {
18   @Id
19   @GeneratedValue(strategy = GenerationType.AUTO)
20   private Long id;
```

```
21
22    private int amount;
23
24    @ManyToOne
25    private Invoice invoice;
26
27    public int getAmount() {
28      return amount;
29    }
30
31    public Long getId() {
32      return id;
33    }
34
35    public Invoice getInvoice() {
36      return invoice;
37    }
38
39    public void setAmount(int amount) {
40      this.amount = amount;
41    }
42
43    public void setId(Long id) {
44      this.id = id;
45    }
46
47    public void setInvoice(Invoice invoice) {
48      this.invoice = invoice;
49    }
50
51    public String toString() {
52      return "[InvoiceLine: id=" + getId() + ", invoice=" + getInvoice()
53          + ", amount=" + getAmount() + "]";
54    }
55  }
```

Listing B.8: InvoiceLine.java

B.3.2 .../no/jodal/query/playground/dao/

InvoicingDao.java

```
1  package no.jodal.query.playground.dao;
2
3  import java.util.List;
4
5  import org.hibernate.SessionFactory;
6
7  import no.jodal.query.playground.domain.Customer;
8  import no.jodal.query.playground.domain.Invoice;
9  import no.jodal.query.playground.domain.InvoiceLine;
10
11 /**
12  * Interface to be implemented in all query variants.
13  *
14  * @author Stein Magnus Jodal (Master's Thesis)
15  */
16 public interface InvoicingDao {
17   public void setSessionFactory(SessionFactory sessionFactory);
18
```

```
19    public void save(Object domainObject);
20
21    public Customer retrieveCustomerById(Long customerId);
22    public List<Customer> retrieveCustomersWithTotalSalesLargerThan(int
          amount);
23    public List<Customer> searchForCustomer(String name);
24    public int retrieveTotalSalesByCustomerName(String name);
25
26    public Invoice retrieveInvoiceById(Long invoiceId);
27    public List<Invoice> retrieveInvoicesByCustomerId(Long customerId);
28    public List<Invoice> retrieveInvoicesWithTotalAmountLargerThan(int
          amount);
29    public List<Object[]> retrieveInvoicesAndTotalAmounts();
30
31    public InvoiceLine retrieveInvoiceLineByid(Long invoiceLineId);
32    public List<InvoiceLine> retrieveInvoiceLinesByInvoiceId(Long
          invoiceId);
33  }
```

Listing B.9: InvoicingDao.java

InvoicingDaoImpl.java

```
1   package no.jodal.query.playground.dao;
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   import no.jodal.query.playground.domain.Customer;
7   import no.jodal.query.playground.domain.Invoice;
8   import no.jodal.query.playground.domain.InvoiceLine;
9
10  import org.hibernate.Session;
11  import org.hibernate.SessionFactory;
12
13  /**
14   * Partial implementation of the {@link InvoicingDao} interface.
15   *
16   * @author Stein Magnus Jodal (Master's Thesis)
17   */
18  public abstract class InvoicingDaoImpl implements
19      InvoicingDao {
20    SessionFactory sessionFactory;
21    Session session;
22
23    public void setSessionFactory(SessionFactory sessionFactory) {
24      this.sessionFactory = sessionFactory;
25    }
26
27    public Session getSession() {
28      if (session == null || !session.isOpen()) {
29        session = sessionFactory.openSession();
30      }
31      return session;
32    }
33
34    public Customer retrieveCustomerById(Long customerId) {
35      return (Customer) getSession().get(Customer.class, customerId);
36    }
37
```

```
38    public Invoice retrieveInvoiceById(Long invoiceId) {
39      return (Invoice) getSession().get(Invoice.class, invoiceId);
40    }
41
42    public InvoiceLine retrieveInvoiceLineByid(Long invoiceLineId) {
43      return (InvoiceLine) getSession().get(InvoiceLine.class,
            invoiceLineId);
44    }
45
46    public List<InvoiceLine> retrieveInvoiceLinesByInvoiceId(Long
          invoiceId) {
47      Invoice invoice = (Invoice) getSession().get(Invoice.class,
            invoiceId);
48      return new ArrayList<InvoiceLine>(invoice.getInvoiceLines());
49    }
50
51    public List<Invoice> retrieveInvoicesByCustomerId(Long customerId) {
52      Customer customer = (Customer) getSession().get(Customer.class,
53          customerId);
54      return new ArrayList<Invoice>(customer.getInvoices());
55    }
56
57    public void save(Object domainObject) {
58      getSession().save(domainObject);
59    }
60  }
```

Listing B.10: InvoicingDaoImpl.java

InvoicingDaoTest.java

```
1   package no.jodal.query.playground.dao;
2
3   import static org.junit.Assert.assertEquals;
4   import static org.junit.Assert.assertFalse;
5   import static org.junit.Assert.assertNotNull;
6   import static org.junit.Assert.assertTrue;
7
8   import java.util.List;
9
10  import no.jodal.query.playground.domain.Customer;
11  import no.jodal.query.playground.domain.Invoice;
12  import no.jodal.query.playground.domain.InvoiceLine;
13
14  import org.hibernate.SessionFactory;
15  import org.junit.Before;
16  import org.junit.Test;
17  import org.springframework.context.ApplicationContext;
18  import org.springframework.context.support.
        ClassPathXmlApplicationContext;
19
20  /**
21   * Integration tests for implementations of InvoicingDao.
22   *
23   * @author Stein Magnus Jodal (Master's Thesis)
24   */
25  public abstract class InvoicingDaoTest {
26    InvoicingDao invoicingBusinessFacade;
27    Customer customer1, customer2;
28    Invoice invoice1, invoice2;
```

```
29    InvoiceLine invoiceLine1, invoiceLine2, invoiceLine3, invoiceLine4;
30
31    public void setInvoicingBusinessFacade(InvoicingDao
          invoicingBusinessFacade) {
32      this.invoicingBusinessFacade = invoicingBusinessFacade;
33    }
34
35    public SessionFactory getSessionFactory() {
36      ApplicationContext applicationContext = new
            ClassPathXmlApplicationContext(
37          "applicationContext.xml");
38      return (SessionFactory) applicationContext.getBean("sessionFactory
            ");
39    }
40
41    @Before
42    public void setUp() throws Exception {
43      invoiceLine1 = new InvoiceLine();
44      invoiceLine1.setAmount(900);
45      invoiceLine2 = new InvoiceLine();
46      invoiceLine2.setAmount(200);
47      invoice1 = new Invoice();
48      invoice1.addInvoiceLine(invoiceLine1);
49      invoice1.addInvoiceLine(invoiceLine2);
50      customer1 = new Customer();
51      customer1.setName("Alice");
52      customer1.addInvoice(invoice1);
53
54      invoiceLine3 = new InvoiceLine();
55      invoiceLine3.setAmount(750);
56      invoiceLine4 = new InvoiceLine();
57      invoiceLine4.setAmount(250);
58      invoice2 = new Invoice();
59      invoice2.addInvoiceLine(invoiceLine3);
60      invoice2.addInvoiceLine(invoiceLine4);
61      customer2 = new Customer();
62      customer2.setName("Bob");
63      customer2.addInvoice(invoice2);
64
65      invoicingBusinessFacade.save(customer1);
66      invoicingBusinessFacade.save(customer2);
67    }
68
69    @Test
70    public void testSave() {
71      assertNotNull(customer1.getId());
72      assertNotNull(invoice1.getId());
73      assertNotNull(invoiceLine1.getId());
74    }
75
76    @Test
77    public void testRetrieveCustomerById() {
78      Customer customer2 = invoicingBusinessFacade
79          .retrieveCustomerById(customer1.getId());
80      assertEquals(customer1, customer2);
81    }
82
83    @Test
84    public void testRetrieveCustomersWithTotalSalesLargerThan() {
85      int amount = 1000;
86      List<Customer> customers = invoicingBusinessFacade
87          .retrieveCustomersWithTotalSalesLargerThan(amount);
```

```
88      assertTrue(customers.contains(customer1));
89      assertFalse(customers.contains(customer2));
90    }
91
92    @Test
93    public void testRetrieveInvoicesWithTotalAmountLargerThan() {
94      int amount = 1000;
95      List<Invoice> invoices = invoicingBusinessFacade
96          .retrieveInvoicesWithTotalAmountLargerThan(amount);
97      assertTrue(invoices.contains(invoice1));
98      assertFalse(invoices.contains(invoice2));
99    }
100
101   @Test
102   public void testRetrieveInvoicesAndTotalAmounts() {
103     List<Object[]> invoicesAndTotalAmounts = invoicingBusinessFacade
104         .retrieveInvoicesAndTotalAmounts();
105     assertEquals(2, invoicesAndTotalAmounts.size());
106     for (Object[] tuple : invoicesAndTotalAmounts) {
107       int totalAmount;
108       if (tuple[0] instanceof Long) {
109         // HQL returns a Long
110         totalAmount = ((Long) tuple[0]).intValue();
111       } else {
112         // Criteria returns an Integer
113         totalAmount = ((Integer) tuple[0]).intValue();
114       }
115       // Invoice invoice = (Invoice) tuple[1];
116       assertTrue(totalAmount == 1000 || totalAmount == 1100);
117     }
118   }
119
120   @Test
121   public void testSearchForCustomerNull() {
122     List<Customer> customers = invoicingBusinessFacade
123         .searchForCustomer(null);
124     assertEquals(0, customers.size());
125   }
126
127   @Test
128   public void testSearchForCustomerFail() {
129     List<Customer> customers = invoicingBusinessFacade
130         .searchForCustomer("Bob");
131     assertFalse(customers.contains(customer1));
132   }
133
134   @Test
135   public void testSearchForCustomerSuccess() {
136     List<Customer> customers = invoicingBusinessFacade
137         .searchForCustomer(customer1.getName());
138     assertTrue(customers.contains(customer1));
139   }
140
141   @Test
142   public void testRetrieveInvoiceById() {
143     Invoice invoice2 = invoicingBusinessFacade.retrieveInvoiceById(
          invoice1
144         .getId());
145     assertEquals(invoice1, invoice2);
146   }
147
148   @Test
```

```
149   public void testRetrieveInvoicesByCustomerId() {
150     List<Invoice> invoices = invoicingBusinessFacade
151         .retrieveInvoicesByCustomerId(customer1.getId());
152     assertEquals(1, invoices.size());
153     assertTrue(invoices.contains(invoice1));
154   }
155
156   @Test
157   public void testRetrieveInvoiceLineByid() {
158     InvoiceLine invoiceLine2 = invoicingBusinessFacade
159         .retrieveInvoiceLineByid(invoiceLine1.getId());
160     assertEquals(invoiceLine1, invoiceLine2);
161   }
162
163   @Test
164   public void testRetrieveInvoiceLinesByInvoiceId() {
165     List<InvoiceLine> invoiceLines = invoicingBusinessFacade
166         .retrieveInvoiceLinesByInvoiceId(invoice1.getId());
167     assertEquals(2, invoiceLines.size());
168     assertTrue(invoiceLines.contains(invoiceLine1));
169     assertTrue(invoiceLines.contains(invoiceLine2));
170   }
171
172   @Test
173   public void testRetrieveTotalSalesByCustomerName() {
174     assertEquals(1100, invoicingBusinessFacade
175         .retrieveTotalSalesByCustomerName("Alice"));
176     assertEquals(1000, invoicingBusinessFacade
177         .retrieveTotalSalesByCustomerName("Bob"));
178   }
179 }
```

Listing B.11: InvoicingDaoTest.java

B.4 playground-core/src/test/java/

B.4.1 .../no/jodal/query/playground/domain/

CustomerTest.java

```
1  package no.jodal.query.playground.domain;
2
3  import static org.junit.Assert.*;
4
5  import java.util.Collection;
6  import java.util.HashSet;
7  import java.util.Set;
8
9  import org.junit.Before;
10 import org.junit.Test;
11
12 /**
13  * Unit tests for {@link Customer}
14  *
15  * @author Stein Magnus Jodal (Master's Thesis)
16  */
17 public class CustomerTest {
18   Customer customer;
```

```
19
20    @Before
21    public void setUp() throws Exception {
22      customer = new Customer();
23    }
24
25    @Test
26    public void testAddInvoice() {
27      Invoice invoice1 = new Invoice();
28
29      customer.addInvoice(invoice1);
30
31      Set<Invoice> invoices = customer.getInvoices();
32      assertTrue(invoices.contains(invoice1));
33      assertEquals(customer, invoice1.getCustomer());
34    }
35
36    @Test
37    public void testCustomer() {
38      assertNull(customer.getId());
39      assertNull(customer.getName());
40      assertNotNull(customer.getInvoices());
41    }
42
43    @Test
44    public void testGetInvoicesEmpty() {
45      Set<Invoice> invoices = customer.getInvoices();
46
47      assertNotNull(invoices);
48      assertTrue(invoices instanceof Collection);
49      assertEquals(0, invoices.size());
50    }
51
52    @Test
53    public void testSetId() {
54      Long id = 1337L;
55
56      customer.setId(id);
57
58      assertEquals(id, customer.getId());
59    }
60
61    @Test
62    public void testSetInvoices() {
63      Set<Invoice> invoices1 = new HashSet<Invoice>();
64      invoices1.add(new Invoice());
65
66      customer.setInvoices(invoices1);
67      Set<Invoice> invoices2 = customer.getInvoices();
68
69      assertNotNull(invoices2);
70      assertTrue(invoices2.containsAll(invoices1));
71    }
72
73    @Test
74    public void testSetName() {
75      String name = "Alice";
76
77      customer.setName(name);
78
79      assertEquals(name, customer.getName());
80    }
```

```
81
82    @Test
83    public void testToString() {
84      String result = customer.toString();
85
86      assertTrue(result.contains("id="));
87      assertTrue(result.contains("name="));
88    }
89  }
```

Listing B.12: CustomerTest.java

InvoiceTest.java

```
1   package no.jodal.query.playground.domain;
2
3   import static org.junit.Assert.*;
4
5   import java.util.Collection;
6   import java.util.HashSet;
7   import java.util.Set;
8
9   import org.junit.Before;
10  import org.junit.Test;
11
12  /**
13   * Unit tests for {@link Invoice}
14   *
15   * @author Stein Magnus Jodal (Master's Thesis)
16   */
17  public class InvoiceTest {
18    Invoice invoice;
19
20    @Before
21    public void setUp() throws Exception {
22      invoice = new Invoice();
23    }
24
25    @Test
26    public void testAddInvoiceLine() {
27      InvoiceLine invoiceLine1 = new InvoiceLine();
28
29      invoice.addInvoiceLine(invoiceLine1);
30
31      Set<InvoiceLine> invoiceLines = invoice.getInvoiceLines();
32      assertTrue(invoiceLines.contains(invoiceLine1));
33      assertEquals(invoice, invoiceLine1.getInvoice());
34    }
35
36    @Test
37    public void testGetInvoiceLinesEmpty() {
38      Set<InvoiceLine> invoiceLines = invoice.getInvoiceLines();
39
40      assertNotNull(invoiceLines);
41      assertTrue(invoiceLines instanceof Collection);
42      assertEquals(0, invoiceLines.size());
43    }
44
45    @Test
46    public void testInvoice() {
```

```
47      assertNull(invoice.getId());
48      assertNull(invoice.getCustomer());
49      assertNotNull(invoice.getInvoiceLines());
50    }
51
52    @Test
53    public void testSetCustomer() {
54      Customer customer = new Customer();
55      customer.setName("Alice");
56
57      invoice.setCustomer(customer);
58
59      assertEquals(customer, invoice.getCustomer());
60    }
61
62    @Test
63    public void testSetId() {
64      Long id = 1337L;
65
66      invoice.setId(id);
67
68      assertEquals(id, invoice.getId());
69    }
70
71    @Test
72    public void testSetInvoiceLines() {
73      Set<InvoiceLine> invoiceLines1 = new HashSet<InvoiceLine>();
74      invoiceLines1.add(new InvoiceLine());
75
76      invoice.setInvoiceLines(invoiceLines1);
77      Set<InvoiceLine> invoiceLines2 = invoice.getInvoiceLines();
78
79      assertNotNull(invoiceLines2);
80      assertTrue(invoiceLines2.containsAll(invoiceLines1));
81    }
82
83    @Test
84    public void testToString() {
85      String result = invoice.toString();
86
87      assertTrue(result.contains("id="));
88      assertTrue(result.contains("customer="));
89      assertTrue(result.contains("numLines="));
90    }
91  }
```

Listing B.13: InvoiceTest.java

InvoiceLineTest.java

```
1   package no.jodal.query.playground.domain;
2
3   import static org.junit.Assert.*;
4
5   import org.junit.Before;
6   import org.junit.Test;
7
8   /**
9    * Unit tests for {@link InvoiceLine}
10   *
```

```
11  * @author Stein Magnus Jodal (Master's Thesis)
12  */
13 public class InvoiceLineTest {
14   InvoiceLine invoiceLine;
15
16   @Before
17   public void setUp() throws Exception {
18     invoiceLine = new InvoiceLine();
19   }
20
21   @Test
22   public void testInvoiceLine() {
23     assertNull(invoiceLine.getId());
24     assertEquals(0, invoiceLine.getAmount());
25     assertNull(invoiceLine.getInvoice());
26   }
27
28   @Test
29   public void testSetAmount() {
30     int amount = 12345;
31
32     invoiceLine.setAmount(amount);
33
34     assertEquals(amount, invoiceLine.getAmount());
35   }
36
37   @Test
38   public void testSetId() {
39     Long id = 1337L;
40
41     invoiceLine.setId(id);
42
43     assertEquals(id, invoiceLine.getId());
44   }
45
46   @Test
47   public void testSetInvoice() {
48     Invoice invoice = new Invoice();
49
50     invoiceLine.setInvoice(invoice);
51
52     assertEquals(invoice, invoiceLine.getInvoice());
53   }
54
55   @Test
56   public void testToString() {
57     String result = invoiceLine.toString();
58
59     assertTrue(result.contains("id="));
60     assertTrue(result.contains("invoice="));
61     assertTrue(result.contains("amount="));
62   }
63 }
```

Listing B.14: InvoiceLineTest.java

Appendix C

Playground Hibernate Project

This appendix contains the Hibernate playground project, which tests HQL and the Criteria API.

C.1 playground-hibernate/

pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
       www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
         apache.org/maven-v4_0_0.xsd">
3
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>no.jodal.query</groupId>
6    <artifactId>playground-hibernate</artifactId>
7    <version>1.0-SNAPSHOT</version>
8    <packaging>jar</packaging>
9    <name>Playground Hibernate</name>
10
11   <organization>
12     <name>Stein Magnus Jodal</name>
13   </organization>
14
15   <parent>
16     <groupId>no.jodal.query</groupId>
17     <artifactId>playground</artifactId>
18     <version>1.0-SNAPSHOT</version>
19   </parent>
20
21   <dependencies>
22     <dependency>
23       <groupId>no.jodal.query</groupId>
24       <artifactId>playground-core</artifactId>
25     </dependency>
26
27     <dependency>
28       <groupId>org.hibernate</groupId>
```

```
29        <artifactId>hibernate</artifactId>
30      </dependency>
31    </dependencies>
32  </project>
```

Listing C.1: pom.xml

C.2 playground-hibernate/src/main/java/

C.2.1 .../no/jodal/query/playground/dao/

InvoicingDaoCriteriaImpl.java

```
1  package no.jodal.query.playground.dao;
2
3  import java.util.List;
4
5  import no.jodal.query.playground.domain.Customer;
6  import no.jodal.query.playground.domain.Invoice;
7
8  import org.hibernate.criterion.Projections;
9  import org.hibernate.criterion.Restrictions;
10
11 /**
12  * Criteria implementation of {@link InvoicingDao}
13  *
14  * @author Stein Magnus Jodal (Master's Thesis)
15  */
16 public class InvoicingDaoCriteriaImpl extends InvoicingDaoImpl {
17   public List<Customer> retrieveCustomersWithTotalSalesLargerThan(int
         amount) {
18     // XXX This is not possible to Implement using Criteria, since
19     // it does not support the HAVING clause in SQL
20     throw new RuntimeException("Method not implemented");
21   }
22
23   @SuppressWarnings("unchecked")
24   public List<Object[]> retrieveInvoicesAndTotalAmounts() {
25     return getSession().createCriteria(Invoice.class, "invoice")
26       .createAlias("invoiceLines", "invoiceLine")
27       .setProjection(Projections.projectionList()
28         .add(Projections.sum("invoiceLine.amount"), "totalAmount")
29         .add(Projections.groupProperty("invoiceLine.invoice")))
30       .list();
31   }
32
33   public List<Invoice> retrieveInvoicesWithTotalAmountLargerThan(int
         amount) {
34     // XXX This is not possible to Implement using Criteria, since
35     // it does not support the HAVING clause in SQL
36     throw new RuntimeException("Method not implemented");
37   }
38
39   @SuppressWarnings("unchecked")
40   public int retrieveTotalSalesByCustomerName(String name) {
41     List results = getSession().createCriteria(Customer.class, "
           customer")
42       .add(Restrictions.eq("name", name))
```

```
43          .createAlias("invoices", "invoice")
44          .createAlias("invoice.invoiceLines", "invoiceLine")
45          .setProjection(Projections.projectionList()
46              .add(Projections.sum("invoiceLine.amount"), "totalSales"))
47          .list();
48      return ((Integer) results.get(0)).intValue();
49    }
50
51    @SuppressWarnings("unchecked")
52    public List<Customer> searchForCustomer(String name) {
53      return getSession().createCriteria(Customer.class)
54          .add(Restrictions.eq("name", name))
55          .list();
56    }
57  }
```

Listing C.2: InvoicingDaoCriteriaImpl.java

InvoicingDaoHqlImpl.java

```
1   package no.jodal.query.playground.dao;
2
3   import java.util.List;
4
5   import no.jodal.query.playground.domain.Customer;
6   import no.jodal.query.playground.domain.Invoice;
7
8   /**
9    * HQL implementation of {@link InvoicingDao}
10   *
11   * @author Stein Magnus Jodal (Master's Thesis)
12   */
13  public class InvoicingDaoHqlImpl extends InvoicingDaoImpl {
14    @SuppressWarnings("unchecked")
15    public List<Customer> retrieveCustomersWithTotalSalesLargerThan(int
          amount) {
16      // XXX Why must we "group by customer, customer.name" and not just
17      // "group by customer"?
18      String query = "select customer " + "from Customer as customer "
19          + "inner join customer.invoices as invoice "
20          + "inner join invoice.invoiceLines as invoiceLine "
21          + "group by customer, customer.name "
22          + "having sum(invoiceLine.amount) > :amount";
23      return getSession().createQuery(query).setInteger("amount", amount
            )
24          .list();
25    }
26
27    @SuppressWarnings("unchecked")
28    public List<Object[]> retrieveInvoicesAndTotalAmounts() {
29      // XXX Why must we "group by invoice, invoice.customer" and not
            just
30      // "group by invoice"?
31      String query = "select sum(invoiceLine.amount), invoice "
32          + "from Invoice as invoice "
33          + "inner join invoice.invoiceLines as invoiceLine "
34          + "group by invoice, invoice.customer";
35      return getSession().createQuery(query).list();
36    }
37
```

```
38    @SuppressWarnings("unchecked")
39    public List<Invoice> retrieveInvoicesWithTotalAmountLargerThan(int
          amount) {
40      // XXX Why must we "group by invoice, invoice.customer" and not
            just
41      // "group by invoice"?
42      String query = "select invoice " + "from Invoice as invoice "
43          + "inner join invoice.invoiceLines as invoiceLine "
44          + "group by invoice, invoice.customer "
45          + "having sum(invoiceLine.amount) > :amount";
46      return getSession().createQuery(query).setInteger("amount", amount
            )
47          .list();
48    }
49
50    @SuppressWarnings("unchecked")
51    public int retrieveTotalSalesByCustomerName(String name) {
52      String query = "select sum(invoiceLine.amount) "
53          + "from Customer as customer "
54          + "inner join customer.invoices as invoice "
55          + "inner join invoice.invoiceLines as invoiceLine "
56          + "where customer.name = :name";
57      List results = getSession().createQuery(query).setString("name",
            name)
58          .list();
59      return ((Long) results.get(0)).intValue();
60    }
61
62    @SuppressWarnings("unchecked")
63    public List<Customer> searchForCustomer(String name) {
64      String query = "from Customer as customer where customer.name = :
            name";
65      return getSession().createQuery(query).setString("name", name).
            list();
66    }
67  }
```

Listing C.3: InvoicingDaoHqlImpl.java

C.3 playground-hibernate/src/test/java/

C.3.1 .../no/jodal/query/playground/dao/

InvoicingDaoCriteriaImplTest.java

```
1   package no.jodal.query.playground.dao;
2
3   import org.junit.Before;
4   import org.junit.Ignore;
5   import org.junit.Test;
6
7   /**
8    * Integration test adapter for {@link InvoicingDaoCriteriaImpl}
9    *
10   * @author Stein Magnus Jodal (Master's Thesis)
11   */
12  public class InvoicingDaoCriteriaImplTest extends InvoicingDaoTest {
13    @Before
```

```
14    public void setUp() throws Exception {
15      InvoicingDao invoicingDao = (InvoicingDao) new
            InvoicingDaoCriteriaImpl();
16      invoicingDao.setSessionFactory(getSessionFactory());
17      setInvoicingBusinessFacade(invoicingDao);
18      super.setUp();
19    }
20
21    @Ignore
22    @Override
23    @Test
24    public void testRetrieveCustomersWithTotalSalesLargerThan() {
25      // Not possible with Criteria
26    }
27
28    @Ignore
29    @Override
30    @Test
31    public void testRetrieveInvoicesWithTotalAmountLargerThan() {
32      // Not possible with Criteria
33    }
34  }
```

Listing C.4: InvoicingDaoCriteriaImplTest.java

InvoicingDaoHqlImplTest.java

```
1   package no.jodal.query.playground.dao;
2
3   import org.junit.Before;
4
5   /**
6    * Integration test adapter for {@link InvoicingDaoHqlImpl}
7    *
8    * @author Stein Magnus Jodal (Master's Thesis)
9    */
10  public class InvoicingDaoHqlImplTest extends
11      InvoicingDaoTest {
12    @Before
13    public void setUp() throws Exception {
14      InvoicingDao invoicingDao = (InvoicingDao) new InvoicingDaoHqlImpl
            ();
15      invoicingDao.setSessionFactory(getSessionFactory());
16      setInvoicingBusinessFacade(invoicingDao);
17      super.setUp();
18    }
19  }
```

Listing C.5: InvoicingDaoHqlImplTest.java

Appendix D

Playground Querydsl Project

This appendix contains the Querydsl playground project, which tests the Querydsl API on top of Hibernate.

D.1 playground-querydsl/

pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
       www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
         apache.org/maven-v4_0_0.xsd">
3
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>no.jodal.query</groupId>
6    <artifactId>playground-querydsl</artifactId>
7    <version>1.0-SNAPSHOT</version>
8    <packaging>jar</packaging>
9    <name>Playground Querydsl</name>
10
11   <organization>
12     <name>Stein Magnus Jodal</name>
13   </organization>
14
15   <parent>
16     <groupId>no.jodal.query</groupId>
17     <artifactId>playground</artifactId>
18     <version>1.0-SNAPSHOT</version>
19   </parent>
20
21   <dependencies>
22     <dependency>
23       <groupId>no.jodal.query</groupId>
24       <artifactId>playground-core</artifactId>
25     </dependency>
26   </dependencies>
27 </project>
```

Listing D.1: pom.xml

D.2 playground-querydsl/src/main/java/

D.2.1 .../no/jodal/query/playground/dao/

InvoicingDaoQuerydslImpl.java

```
1  package no.jodal.query.playground.dao;
2
3  import static com.mysema.query.grammar.HqlGrammar.*;
4
5  import java.util.Collections;
6  import java.util.List;
7
8  import no.jodal.query.playground.domain.Customer;
9  import no.jodal.query.playground.domain.Invoice;
10 import no.jodal.query.playground.domain.QCustomer;
11 import no.jodal.query.playground.domain.QInvoice;
12 import no.jodal.query.playground.domain.QInvoiceLine;
13
14 import com.mysema.query.grammar.types.Expr.EEntity;
15 import com.mysema.query.hql.HqlQuery;
16
17 /**
18  * Querydsl implementation of {@link InvoicingDao}
19  *
20  * @author Stein Magnus Jodal (Master's Thesis)
21  */
22 public class InvoicingDaoQuerydslImpl extends InvoicingDaoImpl {
23   public List<Customer> retrieveCustomersWithTotalSalesLargerThan(int
         amount) {
24     QCustomer customer = new QCustomer("customer");
25     QInvoice invoice = new QInvoice("invoice");
26     QInvoiceLine invoiceLine = new QInvoiceLine("invoiceLine");
27     // I had some problems here, but they are now solved.
28     // See https://bugs.launchpad.net/querydsl/+bug/326650
29     List<Customer> results = from(customer)
30       .innerJoin(customer.invoices.as(invoice))
31       .innerJoin(invoice.invoiceLines.as(invoiceLine))
32       .groupBy(customer, customer.name)
33       .having(sum(invoiceLine.amount).gt(amount))
34       .list(customer);
35     return results;
36   }
37
38   public List<Object[]> retrieveInvoicesAndTotalAmounts() {
39     QInvoice invoice = new QInvoice("invoice");
40     QInvoiceLine invoiceLine = new QInvoiceLine("invoiceLine");
41     List<Object[]> results = from(invoice)
42       .innerJoin(invoice.invoiceLines.as(invoiceLine))
43       .groupBy(invoice, invoice.customer)
44       .list(sum(invoiceLine.amount), invoice);
45     return results;
46   }
47
48   public List<Invoice> retrieveInvoicesWithTotalAmountLargerThan(int
         amount) {
49     QInvoice invoice = new QInvoice("invoice");
50     QInvoiceLine invoiceLine = new QInvoiceLine("invoiceLine");
51     // I had some problems here, but they are now solved.
52     // See https://bugs.launchpad.net/querydsl/+bug/326650
53     List<Invoice> results = from(invoice)
```

```
54        .innerJoin(invoice.invoiceLines.as(invoiceLine))
55        .groupBy(invoice, invoice.customer)
56        .having(sum(invoiceLine.amount).gt(amount))
57        .list(invoice);
58      return results;
59    }
60
61    public int retrieveTotalSalesByCustomerName(String name) {
62      QCustomer customer = new QCustomer("customer");
63      QInvoice invoice = new QInvoice("invoice");
64      QInvoiceLine invoiceLine = new QInvoiceLine("invoiceLine");
65      // I had some problems here, but they are now solved.
66      // See https://bugs.launchpad.net/querydsl/+bug/326650
67      List<Integer> results = from(customer)
68        .innerJoin(customer.invoices.as(invoice))
69        .innerJoin(invoice.invoiceLines.as(invoiceLine))
70        .where(customer.name.eq(name))
71        .list(sum(invoiceLine.amount).intValue());
72      return results.get(0);
73    }
74
75    public List<Customer> searchForCustomer(String name) {
76      if (name == null) {
77        return Collections.emptyList();
78      } else {
79        QCustomer customer = new QCustomer("customer");
80        return from(customer)
81          .where(customer.name.eq(name))
82          .list(customer);
83      }
84    }
85
86    private HqlQuery from(EEntity<?>... s) {
87      return new HqlQuery(getSession()).from(s);
88    }
89  }
```

Listing D.2: InvoicingDaoQuerydslImpl.java

D.3 playground-querydsl/src/test/java/

D.3.1 .../no/jodal/query/playground/dao/

InvoicingDaoQuerydslImplTest.java

```
1  package no.jodal.query.playground.dao;
2
3  import org.junit.Before;
4
5  /**
6   * Integration test adapter for {@link InvoicingDaoQuerydslImpl}
7   *
8   * @author Stein Magnus Jodal (Master's Thesis)
9   */
10 public class InvoicingDaoQuerydslImplTest extends InvoicingDaoTest {
11   @Before
12   public void setUp() throws Exception {
13     InvoicingDao invoicingDao = (InvoicingDao) new
           InvoicingDaoQuerydslImpl();
```

```
14      invoicingDao.setSessionFactory(getSessionFactory());
15      setInvoicingBusinessFacade(invoicingDao);
16      super.setUp();
17    }
18  }
```

Listing D.3: InvoicingDaoQuerydslImplTest.java

Appendix E

Iterate LeanCast

This appendix contains the relevant part of Iterate's LeanCast project. *PersistenceFacade*, *HibernatePersistenceFacade*, and *HibernatePersistence-FacadeTest* has been taken almost unchanged from the LeanCast project. *QuerydslPersistenceFacade* and *QuerydslPersistenceFacadeTest* has been implemented as an alternative implementation of *PersistenceFacade* using the Querydsl API. See Chapter 5 for more information on this work.

E.1 leancast-business/src/main/java

E.1.1 .../no/iterate/leancast/dao/

PersistenceFacade.java

```
1  package no.iterate.leancast.dao;
2
3  import java.util.Date;
4  import java.util.List;
5
6  import no.iterate.leancast.business.exceptions.
       IntervalOutOfRangeException;
7  import no.iterate.leancast.domain.Message;
8  import no.iterate.leancast.domain.Nag;
9  import no.iterate.leancast.domain.SmsMessage;
10 import no.iterate.leancast.domain.User;
11 import no.iterate.leancast.exceptions.ConstraintViolationException;
12 import no.iterate.leancast.exceptions.SmsMessageNotFoundException;
13 import no.iterate.leancast.exceptions.UserNotFoundException;
14
15 /**
16  * Interface between the the business layer and the persistence store.
17  *
18  * @author Iterate Summer Project 2008
19  */
20 public interface PersistenceFacade {
21   public void save(Object domainObject) throws
         ConstraintViolationException;
22
23   public void delete(Object domainObject);
```

```
24
25    public List<User> retrieveAllUsers();
26
27    public List<Message> retrieveConversation(User user1, User user2);
28
29    public List<Message> retrieveMessages(Date timestamp);
30
31    public List<Message> retrieveMessages(User user);
32
33    public List<Nag> retrieveNags(Date fromDate, Date toDate)
34        throws IntervalOutOfRangeException;
35
36    public List<SmsMessage> retrieveNewSmsMessages();
37
38    public SmsMessage retrieveSmsMessage(String identifier)
39        throws SmsMessageNotFoundException;
40
41    public User retrieveUser(int userId) throws UserNotFoundException;
42
43    public User retrieveUser(String emailAddress) throws
          UserNotFoundException;
44
45    public User retrieveUserByPhoneNumber(String phoneNumberCountryCode,
46        String phoneNumber) throws UserNotFoundException;
47
48    public User retrieveUserByXmppAddress(String xmppAddress)
49        throws UserNotFoundException;
50
51    public List<User> retrieveUsers(Date lastModified);
52 }
```

Listing E.1: PersistenceFacade.java

HibernatePersistenceFacade.java

```
1  package no.iterate.leancast.dao;
2
3  import java.sql.Time;
4  import java.util.Calendar;
5  import java.util.Date;
6  import java.util.List;
7
8  import no.iterate.leancast.business.exceptions.
       IntervalOutOfRangeException;
9  import no.iterate.leancast.domain.Message;
10 import no.iterate.leancast.domain.Nag;
11 import no.iterate.leancast.domain.SmsMessage;
12 import no.iterate.leancast.domain.User;
13 import no.iterate.leancast.exceptions.ConstraintViolationException;
14 import no.iterate.leancast.exceptions.SmsMessageNotFoundException;
15 import no.iterate.leancast.exceptions.UserNotFoundException;
16
17 import org.hibernate.Criteria;
18 import org.hibernate.HibernateException;
19 import org.hibernate.Session;
20 import org.hibernate.Transaction;
21 import org.hibernate.criterion.CriteriaSpecification;
22 import org.hibernate.criterion.LogicalExpression;
23 import org.hibernate.criterion.Restrictions;
24 import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
```

```
25 import org.springframework.transaction.annotation.Transactional;
26
27 /**
28  * Hibernate implementation of the PersistenceFacade interface.
29  *
30  * @author Iterate Summer Project 2008
31  * @see PersistenceFacade
32  */
33 @Transactional
34 public class HibernatePersistenceFacade extends HibernateDaoSupport
       implements
35     PersistenceFacade {
36
37   private Session hibernateSession;
38
39   public HibernatePersistenceFacade() {
40   }
41
42   public HibernatePersistenceFacade(Session session) {
43     setHibernateSession(session);
44   }
45
46   /**
47    * Delete an entry from the database
48    *
49    * @param domainObject
50    *            the object to be deleted
51    */
52   public void delete(Object domainObject) {
53     Session session = getHibernateSession();
54     Transaction transaction = session.beginTransaction();
55     session.delete(domainObject);
56     transaction.commit();
57     session.flush();
58   }
59
60   public Session getHibernateSession() {
61     if (hibernateSession == null || !hibernateSession.isOpen()) {
62       return getSession();
63     }
64     return hibernateSession;
65   }
66
67   @SuppressWarnings("unchecked")
68   public List<User> retrieveAllUsers() {
69     Session session = getHibernateSession();
70     Criteria criteria = session.createCriteria(User.class);
71
72     List<User> list = criteria.list();
73     return list;
74   }
75
76   @SuppressWarnings("unchecked")
77   public List<Message> retrieveConversation(User user1,
78       User user2) {
79     Session session = getHibernateSession();
80     Criteria criteria = session.createCriteria(Message.class);
81     criteria
82         .setResultTransformer(CriteriaSpecification.
               DISTINCT_ROOT_ENTITY);
83     criteria.createAlias("receivers", "r");
84
```

```
85      criteria.add(Restrictions.or(Restrictions.and(Restrictions.eq("
            sender",
86          user1), Restrictions.eq("r.id", user2.getId())), Restrictions
87          .and(Restrictions.eq("sender", user2), Restrictions.eq("r.id",
88              user1.getId()))));
89      List<Message> list = criteria.list();
90
91      return list;
92    }
93
94    @SuppressWarnings("unchecked")
95    public List<Message> retrieveMessages(Date timestamp) {
96      Session session = getHibernateSession();
97      Criteria criteria = session.createCriteria(Message.class);
98      criteria.add(Restrictions.gt("timestamp", timestamp));
99      return (List<Message>) criteria.list();
100   }
101
102   @SuppressWarnings("unchecked")
103   public List<Message> retrieveMessages(User user) {
104     Session session = getHibernateSession();
105     Criteria criteria = session.createCriteria(Message.class);
106     criteria
107         .setResultTransformer(CriteriaSpecification.
                DISTINCT_ROOT_ENTITY);
108     criteria.createAlias("receivers", "r");
109
110     LogicalExpression userEqualsSenderOrReceiver = Restrictions.or(
111         Restrictions.eq("sender", user), (Restrictions.eq("r.id", user
                .getId())));
113
114     criteria.add(userEqualsSenderOrReceiver);
115
116     return (List<Message>) criteria.list();
117   }
118
119   /**
120    * Retrieve a list of nags which should be sent within the given
           interval.
121    * The interval can not be greater than 59 minutes since this could
           cause
122    * ambiguous nags.
123    *
124    * The precision is in minutes.
125    *
126    * @throws IntervalOutOfRangeException
127    */
128   @SuppressWarnings("unchecked")
129   public List<Nag> retrieveNags(Date fromDate, Date toDate)
130       throws IntervalOutOfRangeException {
131     if (toDate.getTime() - fromDate.getTime() >= 60 * 60 * 1000) {
132       throw new IntervalOutOfRangeException(
133           "The interval was too big. Must be 59 minutes or less.");
134     } else if (toDate.before(fromDate)) {
135       throw new IntervalOutOfRangeException(
136           "The to date was set before the from date in the interval.")
                ;
137     }
138
139     Session session = getHibernateSession();
140     Transaction transaction = session.beginTransaction();
141
```

```
142     List<Nag> resultset = null;
143     try {
144       Calendar fromCalender = Calendar.getInstance();
145       fromCalender.setTime(fromDate);
146 
147       Calendar toCalender = Calendar.getInstance();
148       toCalender.setTime(toDate);
149 
150       // Get the weekday for the period
151       String[] days = new String[] { "sunday", "monday", "tuesday",
152           "wednesday", "thursday", "friday", "saturday" };
153       String fromDateDayOfWeekField = days[fromCalender
154           .get(Calendar.DAY_OF_WEEK) - 1];
155       String toDateDayOfWeekField = days[toCalender
156           .get(Calendar.DAY_OF_WEEK) - 1];
157 
158       // The nag must be active
159       Criteria criteria = session.createCriteria(Nag.class);
160       criteria.add(Restrictions.or(Restrictions.isNull("start"),
161           Restrictions.le("start", fromDate)));
162       criteria.add(Restrictions.or(Restrictions.isNull("end"),
163           Restrictions.gt("end", toDate)));
164 
165       // Remove date, second and milliseconds from calendar intervals
166       toCalender.set(0, 0, 0);
167       toCalender.set(Calendar.SECOND, 0);
168       toCalender.set(Calendar.MILLISECOND, 0);
169       fromCalender.set(0, 0, 0);
170       fromCalender.set(Calendar.SECOND, 0);
171       fromCalender.set(Calendar.MILLISECOND, 0);
172 
173       // Create time objects from the calendar intervals
174       Time fromTime = new Time(fromCalender.getTimeInMillis());
175       Time toTime = new Time(toCalender.getTimeInMillis());
176 
177       if (!fromDateDayOfWeekField.equals(toDateDayOfWeekField)) { //
              If
178         // the interval is not within the same day
179         LogicalExpression before = Restrictions.and(Restrictions.ge(
180             "time", fromTime), Restrictions.eq(
181             fromDateDayOfWeekField, true));
182         LogicalExpression after = Restrictions.and(Restrictions.lt(
183             "time", toTime), Restrictions.eq(toDateDayOfWeekField,
184             true));
185         criteria.add(Restrictions.or(before, after));
186       } else if (fromTime.equals(toTime)) {
187         criteria.add(Restrictions.between("time", fromTime, toTime));
188         criteria.add(Restrictions.eq(fromDateDayOfWeekField, true));
189       } else {
190         criteria.add(Restrictions.ge("time", fromTime));
191         criteria.add(Restrictions.lt("time", toTime));
192         criteria.add(Restrictions.eq(fromDateDayOfWeekField, true));
193       }
194 
195       // Retrieve the list of results
196       resultset = criteria.list();
197       transaction.commit();
198     } catch (RuntimeException e) {
199       transaction.rollback();
200     } finally {
201       session.flush();
202     }
```

```
203     return resultset;
204   }
205
206   /**
207    * Retrieve all SMS messages which has not previously been processed
           .
208    */
209   @SuppressWarnings("unchecked")
210   public List<SmsMessage> retrieveNewSmsMessages() {
211     Session session = getHibernateSession();
212     Criteria criteria = session.createCriteria(SmsMessage.class);
213     criteria.add(Restrictions.eq("processed", false));
214     return (List<SmsMessage>) criteria.list();
215   }
216
217   @SuppressWarnings("unchecked")
218   public SmsMessage retrieveSmsMessage(String identifier)
219       throws SmsMessageNotFoundException {
220     Criteria criteria = getHibernateSession().createCriteria(
221         SmsMessage.class);
222     criteria.add(Restrictions.eq("identifier", identifier));
223     criteria.setMaxResults(1);
224
225     List<SmsMessage> resultSet = criteria.list();
226     if (resultSet.size() == 0) {
227       throw new SmsMessageNotFoundException("identifier=" + identifier
              );
228     } else {
229       return resultSet.get(0);
230     }
231   }
232
233   @SuppressWarnings("unchecked")
234   public User retrieveUser(int userId)
235       throws UserNotFoundException {
236     Criteria userCriteria = getHibernateSession()
237         .createCriteria(User.class);
238     userCriteria.add(Restrictions.eq("id", userId));
239     userCriteria.setMaxResults(1);
240
241     List<User> userResultSet = userCriteria.list();
242     if (userResultSet.size() == 0) {
243       throw new UserNotFoundException("The user id was not found.");
244     } else {
245       return userResultSet.get(0);
246     }
247   }
248
249   @SuppressWarnings("unchecked")
250   public User retrieveUser(String emailAddress)
251       throws UserNotFoundException {
252     Criteria userCriteria = getHibernateSession()
253         .createCriteria(User.class);
254     userCriteria.add(Restrictions.eq("emailAddress", emailAddress));
255     userCriteria.setMaxResults(1);
256
257     List<User> userResultSet = userCriteria.list();
258     if (userResultSet.size() == 0) {
259       throw new UserNotFoundException("The email address was not found
              .");
260     } else {
261       return userResultSet.get(0);
```

```
262       }
263     }
264
265     @SuppressWarnings("unchecked")
266     public User retrieveUserByPhoneNumber(String phoneNumberCountryCode,
267         String phoneNumber) throws UserNotFoundException {
268       Criteria userCriteria = getHibernateSession()
269           .createCriteria(User.class);
270       userCriteria.add(Restrictions.like("phoneNumberCountryCode",
271           phoneNumberCountryCode + "%"));
272       userCriteria.add(Restrictions.like("phoneNumber", "%" +
              phoneNumber));
273       userCriteria.setMaxResults(1);
274
275       List<User> userResultSet = userCriteria.list();
276       if (userResultSet.size() == 0) {
277         throw new UserNotFoundException("User with phone number \""
278             + phoneNumber + "\" was not found.");
279       } else {
280         return userResultSet.get(0);
281       }
282     }
283
284     @SuppressWarnings("unchecked")
285     public User retrieveUserByXmppAddress(String xmppAddress)
286         throws UserNotFoundException {
287       Criteria userCriteria = getHibernateSession()
288           .createCriteria(User.class);
289       userCriteria.add(Restrictions.eq("xmppAddress", xmppAddress));
290       userCriteria.setMaxResults(1);
291
292       List<User> userResultSet = userCriteria.list();
293       if (userResultSet.size() == 0) {
294         throw new UserNotFoundException("User with XMPP address \""
295             + xmppAddress + "\" was not found.");
296       } else {
297         return userResultSet.get(0);
298       }
299     }
300
301     @SuppressWarnings("unchecked")
302     public List<User> retrieveUsers(Date lastModified) {
303       Session session = getHibernateSession();
304       Criteria criteria = session.createCriteria(User.class);
305       criteria.add(Restrictions.gt("lastModified", lastModified));
306       return (List<User>) criteria.list();
307     }
308
309     public void save(Object domainObject)
310         throws ConstraintViolationException {
311       Session session = getHibernateSession();
312       Transaction transaction = session.beginTransaction();
313       try {
314         session.saveOrUpdate(domainObject);
315       } catch (org.hibernate.exception.ConstraintViolationException e) {
316         transaction.rollback();
317         throw new ConstraintViolationException(e.getMessage());
318       } catch (HibernateException e) {
319         // XXX: This is a hack needed because of bad session handling in
                  the
320         // daemon system.
321         // Will first try saveOrUpdate, but catch HibernateException
```

```
322         // to handle
323         // "Illegal attempt to associate a collection with two open
324         // sessions". The hack is done by merging the domainObject with
                the
325         // database object and bind it to the current session. Which
                allows
326         // us to update the object in the database.
327         domainObject = session.merge(domainObject);
328         session.saveOrUpdate(domainObject);
329       }
330       session.flush();
331       transaction.commit();
332     }
333
334     public void setHibernateSession(Session hibernateSession) {
335       this.hibernateSession = hibernateSession;
336     }
337 }
```

Listing E.2: HibernatePersistenceFacade.java

QuerydslPersistenceFacade.java

```
1  package no.iterate.leancast.dao;
2
3  import java.sql.Time;
4  import java.util.Calendar;
5  import java.util.Date;
6  import java.util.List;
7
8  import no.iterate.leancast.business.exceptions.
       IntervalOutOfRangeException;
9  import no.iterate.leancast.domain.Message;
10 import no.iterate.leancast.domain.Nag;
11 import no.iterate.leancast.domain.QMessage;
12 import no.iterate.leancast.domain.QNag;
13 import no.iterate.leancast.domain.QSmsMessage;
14 import no.iterate.leancast.domain.QUser;
15 import no.iterate.leancast.domain.SmsMessage;
16 import no.iterate.leancast.domain.User;
17 import no.iterate.leancast.exceptions.SmsMessageNotFoundException;
18 import no.iterate.leancast.exceptions.UserNotFoundException;
19
20 import org.hibernate.classic.Session;
21
22 import com.mysema.query.grammar.types.Expr.EBoolean;
23 import com.mysema.query.grammar.types.Expr.EEntity;
24 import com.mysema.query.grammar.types.Path.PBoolean;
25 import com.mysema.query.hql.HqlQuery;
26
27 /**
28  * Hibernate implementation of the PersistenceFacade interface,
29  * using Querydsl for building queries.
30  *
31  * @author Stein Magnus Jodal (Master's Thesis)
32  * @see PersistenceFacade
33  */
34 public class QuerydslPersistenceFacade extends
       HibernatePersistenceFacade
35     implements PersistenceFacade {
```

```
36    Session hibernateSession;
37
38    public QuerydslPersistenceFacade() {
39    }
40
41    public QuerydslPersistenceFacade(Session hibernateSession) {
42      setHibernateSession(hibernateSession);
43    }
44
45    protected HqlQuery from(EEntity<?>... s) {
46      return new HqlQuery(getHibernateSession()).from(s);
47    }
48
49    @Override
50    public List<User> retrieveAllUsers() {
51      QUser user = new QUser("user");
52      return from(user).list(user);
53    }
54
55    @Override
56    public List<Message> retrieveConversation(User user1, User user2) {
57      QMessage message = new QMessage("message");
58      return from(message)
59        .where((message.sender.eq(user1)
60            .and(message.receivers.contains(user2)))
61          .or(message.sender.eq(user2)
62            .and(message.receivers.contains(user1))))
63        .list(message);
64    }
65
66    @Override
67    public List<Message> retrieveMessages(Date timestamp) {
68      QMessage message = new QMessage("message");
69      return from(message)
70        .where(message.timestamp.after(timestamp))
71        .list(message);
72    }
73
74    @Override
75    public List<Message> retrieveMessages(User user) {
76      QMessage message = new QMessage("message");
77      return from(message)
78        .where(message.sender.eq(user)
79            .or(message.receivers.contains(user)))
80        .list(message);
81    }
82
83    @Override
84    public List<Nag> retrieveNags(Date fromDate, Date toDate)
85        throws IntervalOutOfRangeException {
86
87      if (toDate.getTime() - fromDate.getTime() >= 60 * 60 * 1000) {
88        throw new IntervalOutOfRangeException(
89            "The interval was too big. Must be 59 minutes or less.");
90      }
91      if (toDate.before(fromDate)) {
92        throw new IntervalOutOfRangeException(
93            "The to date was set before the from date in the interval.")
                ;
94      }
95
96      QNag nag = new QNag("nag");
```

```
97
98      Calendar fromCalender = Calendar.getInstance();
99      fromCalender.setTime(fromDate);
100     Calendar toCalender = Calendar.getInstance();
101     toCalender.setTime(toDate);
102
103     // Get the weekday for the period
104     PBoolean[] weekdays = new PBoolean[] { nag.sunday, nag.monday,
105         nag.tuesday, nag.wednesday, nag.thursday, nag.friday,
106         nag.saturday };
107     PBoolean fromDateDayOfWeek = weekdays[fromCalender
108         .get(Calendar.DAY_OF_WEEK) - 1];
109     PBoolean toDateDayOfWeek = weekdays[toCalender
110         .get(Calendar.DAY_OF_WEEK) - 1];
111
112     // Remove date, second and milliseconds from calendar intervals
113     toCalender.set(0, 0, 0);
114     toCalender.set(Calendar.SECOND, 0);
115     toCalender.set(Calendar.MILLISECOND, 0);
116     fromCalender.set(0, 0, 0);
117     fromCalender.set(Calendar.SECOND, 0);
118     fromCalender.set(Calendar.MILLISECOND, 0);
119
120     // Create time objects from the calendar intervals
121     Time fromTime = new Time(fromCalender.getTimeInMillis());
122     Time toTime = new Time(toCalender.getTimeInMillis());
123
124     EBoolean nagIsActive = (nag.start.isnull()
125           .or(nag.start.before(fromDate)
126           .or(nag.start.eq(fromDate))))
127         .and(nag.end.isnull()
128           .or(nag.end.after(toDate)));
129     EBoolean nagTimeInInterval;
130
131     if (!fromDateDayOfWeek.equals(toDateDayOfWeek)) {
132       EBoolean after = (nag.time.after(fromTime)
133           .or(nag.time.eq(fromTime)))
134         .and(fromDateDayOfWeek.eq(true));
135       EBoolean before = nag.time.before(toTime)
136         .and(toDateDayOfWeek.eq(true));
137       nagTimeInInterval = before.or(after);
138     } else if (fromTime.equals(toTime)) {
139       nagTimeInInterval = nag.time.eq(fromTime)
140         .and(fromDateDayOfWeek.eq(true));
141     } else {
142       nagTimeInInterval = (nag.time.after(fromTime)
143           .or(nag.time.eq(fromTime)))
144         .and((nag.time.before(toTime)))
145         .and(fromDateDayOfWeek.eq(true));
146     }
147
148     return from(nag)
149       .where(nagIsActive.and(nagTimeInInterval))
150       .list(nag);
151   }
152
153   @Override
154   public List<SmsMessage> retrieveNewSmsMessages() {
155     QSmsMessage smsMessage = new QSmsMessage("smsMessage");
156     return from(smsMessage)
157       .where(smsMessage.processed.eq(false))
158       .list(smsMessage);
```

```
159   }
160
161   @Override
162   public SmsMessage retrieveSmsMessage(String identifier)
163       throws SmsMessageNotFoundException {
164     QSmsMessage smsMessage = new QSmsMessage("smsMessage");
165     List<SmsMessage> results = from(smsMessage)
166       .where(smsMessage.identifier.eq(identifier))
167       .list(smsMessage);
168     if (results.size() == 1) {
169       return results.get(0);
170     } else {
171       throw new SmsMessageNotFoundException("SMS message ("
172           + identifier + ") not found.");
173     }
174   }
175
176   @Override
177   public User retrieveUser(int userId) throws UserNotFoundException {
178     QUser user = new QUser("user");
179     List<User> results = from(user)
180       .where(user.id.eq(userId))
181       .list(user);
182     if (results.size() == 1) {
183       return results.get(0);
184     } else {
185       throw new UserNotFoundException("User (" + userId + ") not found
              .");
186     }
187   }
188
189   @Override
190   public User retrieveUser(String emailAddress) throws
          UserNotFoundException {
191     QUser user = new QUser("user");
192     List<User> results = from(user)
193       .where(user.emailAddress.eq(emailAddress))
194       .list(user);
195     if (results.size() == 1) {
196       return results.get(0);
197     } else {
198       throw new UserNotFoundException("User (" + emailAddress + ") not
               found.");
199     }
200   }
201
202   @Override
203   public User retrieveUserByPhoneNumber(String phoneNumberCountryCode,
204       String phoneNumber) throws UserNotFoundException {
205     QUser user = new QUser("user");
206     List<User> results = from(user)
207       .where(user.phoneNumberCountryCode.eq(phoneNumberCountryCode)
208           .and(user.phoneNumber.eq(phoneNumber)))
209       .list(user);
210     if (results.size() == 1) {
211       return results.get(0);
212     } else {
213       throw new UserNotFoundException("User ("
214           + phoneNumberCountryCode + phoneNumber + ") not found.");
215     }
216   }
217
```

```
218   @Override
219   public User retrieveUserByXmppAddress(String xmppAddress)
220       throws UserNotFoundException {
221     QUser user = new QUser("user");
222     List<User> results = from(user)
223       .where(user.xmppAddress.eq(xmppAddress))
224       .list(user);
225     if (results.size() == 1) {
226       return results.get(0);
227     } else {
228       throw new UserNotFoundException("User (" + xmppAddress + ") not
              found.");
229     }
230   }
231
232   @Override
233   public List<User> retrieveUsers(Date lastModified) {
234     QUser user = new QUser("user");
235     return from(user)
236       .where(user.lastModified.after(lastModified))
237       .list(user);
238   }
239 }
```

Listing E.3: QuerydslPersistenceFacade.java

E.2 leancast-business/src/test/java

E.2.1 .../no/iterate/leancast/dao/

HibernatePersistenceFacadeTest.java

```
1  package no.iterate.leancast.dao;
2
3  import static org.junit.Assert.assertArrayEquals;
4  import static org.junit.Assert.assertEquals;
5  import static org.junit.Assert.assertNotSame;
6  import static org.junit.Assert.assertTrue;
7
8  import java.util.ArrayList;
9  import java.util.Calendar;
10 import java.util.Date;
11 import java.util.GregorianCalendar;
12 import java.util.HashSet;
13 import java.util.Iterator;
14 import java.util.List;
15 import java.util.Set;
16
17 import no.iterate.leancast.Settings;
18 import no.iterate.leancast.business.exceptions.
       IntervalOutOfRangeException;
19 import no.iterate.leancast.domain.Message;
20 import no.iterate.leancast.domain.Nag;
21 import no.iterate.leancast.domain.SmsMessage;
22 import no.iterate.leancast.domain.Tag;
23 import no.iterate.leancast.domain.TagType;
24 import no.iterate.leancast.domain.User;
25 import no.iterate.leancast.domain.exceptions.
       InvalidDomainDataException;
```

```
26 import no.iterate.leancast.domain.exceptions.
       InvalidEmailAddressException;
27 import no.iterate.leancast.domain.exceptions.InvalidMessageException;
28 import no.iterate.leancast.exceptions.ConstraintViolationException;
29 import no.iterate.leancast.exceptions.SmsMessageNotFoundException;
30 import no.iterate.leancast.exceptions.UserNotFoundException;
31
32 import org.hibernate.SessionFactory;
33 import org.hibernate.cfg.AnnotationConfiguration;
34 import org.hibernate.cfg.Environment;
35 import org.hibernate.classic.Session;
36 import org.junit.Before;
37 import org.junit.Test;
38
39 /**
40  * Unit test for {@link HibernatePersistenceFacade}.
41  *
42  * @author Iterate Summer Project 2008
43  */
44 public class HibernatePersistenceFacadeTest {
45   PersistenceFacade persistenceFacade;
46
47   /**
48    * Sets up the configuration environment for the Hibernate testing.
         The
49    * tests run on a "real" database connection and not a mock facade.
50    *
51    * @throws Exception
52    */
53   @Before
54   public void setUp() throws Exception {
55     AnnotationConfiguration configuration = new
           AnnotationConfiguration();
56
57     configuration.setProperty(Environment.DRIVER, "org.hsqldb.
           jdbcDriver");
58     configuration.setProperty(Environment.URL,
59         "jdbc:hsqldb:mem:ehourdaotest");
60     configuration.setProperty(Environment.USER, "sa");
61     configuration.setProperty(Environment.PASS, "");
62     configuration.setProperty(Environment.DIALECT,
63         "org.hibernate.dialect.HSQLDialect");
64     configuration.setProperty(Environment.POOL_SIZE, "1");
65     configuration.setProperty(Environment.HBM2DDL_AUTO, "create");
66
67     configuration.setProperty(Environment.SHOW_SQL, "false");
68     configuration.setProperty(Environment.HBM2DDL_AUTO, "create");
69     configuration.addAnnotatedClass(Message.class);
70     configuration.addAnnotatedClass(Nag.class);
71     configuration.addAnnotatedClass(SmsMessage.class);
72     configuration.addAnnotatedClass(Tag.class);
73     configuration.addAnnotatedClass(TagType.class);
74     configuration.addAnnotatedClass(User.class);
75
76     SessionFactory sessionFactory = configuration.buildSessionFactory
           ();
77
78     Session session = sessionFactory.openSession();
79     persistenceFacade = getPersistenceFacade(session);
80   }
81
82   public PersistenceFacade getPersistenceFacade(Session session) {
```

```
83      return new HibernatePersistenceFacade(session);
84    }
85
86    @Test(expected = ConstraintViolationException.class)
87    public void testAddDuplicateEmail() throws
          InvalidEmailAddressException,
88        ConstraintViolationException {
89      String email = "testaddress@iterate.no";
90      User user = new User(email);
91      persistenceFacade.save(user);
92      User user2 = new User(email);
93      persistenceFacade.save(user2);
94    }
95
96    @Test
97    public void testDelete() throws InvalidDomainDataException,
98        ConstraintViolationException {
99      assertEquals(0, persistenceFacade.retrieveAllUsers().size());
100     User user = new User("test2@email.com");
101     persistenceFacade.save(user);
102     assertEquals(1, persistenceFacade.retrieveAllUsers().size());
103     persistenceFacade.delete(user);
104     assertEquals(0, persistenceFacade.retrieveAllUsers().size());
105   }
106
107   @Test
108   public void testRetrieveAllUsers() throws InvalidDomainDataException
          ,
109       ConstraintViolationException {
110     assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
111     persistenceFacade.save(new User("test1@email.com"));
112     persistenceFacade.save(new User("test2@email.com"));
113     assertEquals(persistenceFacade.retrieveAllUsers().size(), 2);
114
115     Iterator<User> usersRetrieved = persistenceFacade.retrieveAllUsers
          ()
116         .iterator();
117     assertNotSame(usersRetrieved.next(), usersRetrieved.next());
118
119     persistenceFacade.save(new User("test3@email.com"));
120     persistenceFacade.save(new User("test4@email.com"));
121     assertEquals(persistenceFacade.retrieveAllUsers().size(), 4);
122   }
123
124   @Test(expected = UserNotFoundException.class)
125   public void testRetrieveNonExistingUser() throws
          UserNotFoundException {
126     assertEquals(0, persistenceFacade.retrieveAllUsers().size());
127     persistenceFacade.retrieveUser("test@example.com");
128   }
129
130   /**
131    * Creates a new user, saves it and checks that the same user is
           found, when
132    * using the retrieveUserByEmailAddress method.
133    *
134    * @throws InvalidDomainDataException
135    * @throws ConstraintViolationException
136    * @throws UserNotFoundException
137    */
138   @Test
139   public void testRetrieveUser() throws InvalidDomainDataException,
```

```
140         ConstraintViolationException, UserNotFoundException {
141       String email = "testaddress@iterate.no";
142       User user = new User(email);
143       assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
144       persistenceFacade.save(user);
145       assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
146       User savedUser = persistenceFacade.retrieveUser(email);
147       assertEquals(user, savedUser);
148     }
149
150     /**
151      * Creates a new user, saves it and checks that the same user is
             found, when
152      * using the retrieveUserById method.
153      *
154      * @throws InvalidDomainDataException
155      * @throws ConstraintViolationException
156      * @throws UserNotFoundException
157      */
158     @Test
159     public void testRetrieveUserById() throws InvalidDomainDataException
           ,
160         ConstraintViolationException, UserNotFoundException {
161       String email = "testaddress@iterate.no";
162       User user = new User(email);
163       assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
164       persistenceFacade.save(user);
165       assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
166       User savedUser = persistenceFacade.retrieveUser(1);
167       assertEquals(user, savedUser);
168     }
169
170     /**
171      * Creates a new user, saves it. Tries to retrieve a new user with a
             id
172      * which does not exist in the database and expects to get an
             exception.
173      *
174      * @throws InvalidDomainDataException
175      * @throws ConstraintViolationException
176      * @throws UserNotFoundException
177      */
178     @Test(expected = UserNotFoundException.class)
179     public void testRetrieveUserByNonExistingId()
180         throws InvalidDomainDataException, ConstraintViolationException,
181         UserNotFoundException {
182       String email = "testaddress@iterate.no";
183       User user = new User(email);
184       assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
185       persistenceFacade.save(user);
186       assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
187       User savedUser = persistenceFacade.retrieveUser(3);
188       assertEquals(user, savedUser);
189     }
190
191     /**
192      * Creates a new user, saves it and checks that the same user is
             found, when
193      * using the retrieveUserByXmppAddress method.
194      *
195      * @throws InvalidDomainDataException
196      * @throws ConstraintViolationException
```

```
197     * @throws UserNotFoundException
198     */
199    @Test
200    public void testRetrieveUserByXmppAddress()
201        throws InvalidDomainDataException, ConstraintViolationException,
202        UserNotFoundException {
203      String email = "testaddress@iterate.no";
204      User user = new User(email);
205      user.setXmppAddress("testxmpp@iterate.no");
206      assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
207      persistenceFacade.save(user);
208      assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
209      User savedUser = persistenceFacade.retrieveUserByXmppAddress(user
210          .getXmppAddress());
211      assertEquals(user, savedUser);
212    }
213
214    /**
215     * Creates a new user, saves it and checks that the same user is
            found, when
216     * using the retrieveUserByPhoneNumber method.
217     *
218     * @throws InvalidDomainDataException
219     * @throws ConstraintViolationException
220     * @throws UserNotFoundException
221     */
222    @Test
223    public void testRetrieveUserByPhoneNumber()
224        throws InvalidDomainDataException, ConstraintViolationException,
225        UserNotFoundException {
226      String email = "testaddress@iterate.no";
227      User user = new User(email);
228      user.setPhoneNumber("12345678");
229      user.setPhoneNumberCountryCode("+49");
230      assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
231      persistenceFacade.save(user);
232      assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
233      User savedUser = persistenceFacade.retrieveUserByPhoneNumber("+49"
234          ,
234          "12345678");
235      assertEquals(user, savedUser);
236    }
237
238    /**
239     * Creates a new user, saves it and checks that no user is found
            when trying
240     * to find a user with non-existing phone number and correct country
            code.
241     *
242     * @throws InvalidDomainDataException
243     * @throws ConstraintViolationException
244     * @throws UserNotFoundException
245     */
246    @Test(expected = UserNotFoundException.class)
247    public void testRetrieveUserByNonExistingPhoneNumber()
248        throws InvalidDomainDataException, ConstraintViolationException,
249        UserNotFoundException {
250      String email = "testaddress@iterate.no";
251      User user = new User(email);
252      user.setPhoneNumber("12345678");
253      user.setPhoneNumberCountryCode("+49");
254      assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
```

```
255     persistenceFacade.save(user);
256     assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
257     User savedUser = persistenceFacade.retrieveUserByPhoneNumber("+49"
            ,
258         "87654321");
259     assertEquals(user, savedUser);
260   }
261
262   /**
263    * Creates a new user, saves it and checks that no user is found
           when trying
264    * to find a user with non-existing country code and correct phone
           number.
265    *
266    * @throws InvalidDomainDataException
267    * @throws ConstraintViolationException
268    * @throws UserNotFoundException
269    */
270   @Test(expected = UserNotFoundException.class)
271   public void testRetrieveUserByNonExistingPhoneNumberCountryCode()
272       throws InvalidDomainDataException, ConstraintViolationException,
273       UserNotFoundException {
274     String email = "testaddress@iterate.no";
275     User user = new User(email);
276     user.setPhoneNumber("12345678");
277     user.setPhoneNumberCountryCode("+49");
278     assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
279     persistenceFacade.save(user);
280     assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
281     User savedUser = persistenceFacade.retrieveUserByPhoneNumber("+47"
            ,
282         "12345678");
283     assertEquals(user, savedUser);
284   }
285
286   /**
287    * Creates a new user, saves it and checks that an exception is
           thrown when
288    * trying to get a user with non-existent XMPP address.
289    *
290    * @throws InvalidDomainDataException
291    * @throws ConstraintViolationException
292    * @throws UserNotFoundException
293    */
294   @Test(expected = UserNotFoundException.class)
295   public void testRetrieveUserByNonExistingXmppAddress()
296       throws InvalidDomainDataException, ConstraintViolationException,
297       UserNotFoundException {
298     String email = "testaddress@iterate.no";
299     User user = new User(email);
300     user.setXmppAddress("testxmpp@iterate.no");
301     assertEquals(persistenceFacade.retrieveAllUsers().size(), 0);
302     persistenceFacade.save(user);
303     assertEquals(persistenceFacade.retrieveAllUsers().size(), 1);
304     User savedUser = persistenceFacade
305         .retrieveUserByXmppAddress("testaddress@iterate.no");
306     assertEquals(user, savedUser);
307   }
308
309   /**
310    * Test retrieval of nags within a too big interval.
311    *
```

```
312     * @throws ConstraintViolationException
313     * @throws InvalidEmailAddressException
314     * @throws IntervalOutOfRangeException
315     */
316    @Test(expected = IntervalOutOfRangeException.class)
317    public void testRetrieveNagsTooBigInterval()
318        throws ConstraintViolationException,
               InvalidEmailAddressException,
319        IntervalOutOfRangeException {
320      persistenceFacade.retrieveNags(new Date(0), new Date(1000 * 60 *
           60));
321    }
322
323    /**
324     * Test retrieval of nags within a negative interval.
325     *
326     * @throws ConstraintViolationException
327     * @throws InvalidEmailAddressException
328     * @throws IntervalOutOfRangeException
329     */
330    @Test(expected = IntervalOutOfRangeException.class)
331    public void testRetrieveNagsNegativeInterval()
332        throws ConstraintViolationException,
               InvalidEmailAddressException,
333        IntervalOutOfRangeException {
334      persistenceFacade.retrieveNags(new Date(1), new Date(0));
335    }
336
337    @Test
338    public void testRetrieveNags() throws ConstraintViolationException,
339        InvalidEmailAddressException, IntervalOutOfRangeException {
340      Calendar date = Calendar.getInstance();
341      date.set(2015, 11, 30);
342      Nag nag = new Nag();
343      date.set(2008, 0, 1);
344      nag.setStart(date.getTime());
345      date.set(2018, 0, 1);
346      nag.setEnd(date.getTime());
347      nag.setTime(23, 59);
348      nag.setMonday(true);
349      nag.setChannel("xmpp");
350      User user = new User();
351      user.setEmailAddress("test@example.org");
352      persistenceFacade.save(user);
353      nag.setUser(user);
354      persistenceFacade.save(nag);
355      Nag nag2 = new Nag();
356      date.set(2008, 0, 1);
357      nag2.setStart(date.getTime());
358      nag2.setTime(0, 2);
359      nag2.setMonday(true);
360      nag2.setTuesday(true);
361      nag2.setUser(user);
362      nag2.setChannel("xmpp");
363      persistenceFacade.save(nag2);
364
365      GregorianCalendar cal = new GregorianCalendar();
366      cal.set(2008, 6, 14, 15, 23);
367      GregorianCalendar calTo = new GregorianCalendar();
368      calTo.set(2008, 6, 14, 15, 23);
369      List<Nag> test = persistenceFacade.retrieveNags(cal.getTime(),
           calTo
```

```
370             .getTime());
371         assertEquals(0, test.size());
372 
373         cal.set(2008, 6, 14, 23, 59);
374         calTo.set(2008, 6, 14, 23, 59);
375         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
376         assertEquals(1, test.size());
377 
378         cal.set(2008, 6, 14, 23, 58);
379         calTo.set(2008, 6, 15, 0, 3);
380         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
381         assertEquals(2, test.size());
382 
383         cal.set(2008, 6, 14, 23, 58);
384         calTo.set(2008, 6, 15, 0, 2);
385         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
386         assertEquals(1, test.size());
387 
388         cal.set(2008, 6, 14, 23, 59);
389         calTo.set(2008, 6, 15, 0, 2);
390         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
391         assertEquals(1, test.size());
392 
393         cal.set(2008, 6, 15, 0, 0);
394         calTo.set(2008, 6, 15, 0, 2);
395         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
396         assertEquals(0, test.size());
397 
398         cal.set(2008, 6, 15, 0, 0);
399         calTo.set(2008, 6, 15, 0, 3);
400         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
401         assertEquals(1, test.size());
402 
403         cal.set(2008, 6, 14, 14, 23);
404         calTo.set(2008, 6, 14, 14, 59);
405         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
406         assertEquals(0, test.size());
407 
408         // Do not get nags on a monday, if monday is set to false.
409         nag.setMonday(false);
410         persistenceFacade.save(nag);
411         cal.set(2008, 6, 14, 23, 59);
412         calTo.set(2008, 6, 14, 23, 59);
413         test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
                ());
414         assertEquals(0, test.size());
415     }
416 
417     /**
418      * Test that we don't get a nag with end date in the past.
419      */
420     @Test
421     public void testRetrieveNagsEnded() throws
            ConstraintViolationException,
422             InvalidEmailAddressException, IntervalOutOfRangeException {
```

```
423     Calendar date = Calendar.getInstance();
424     date.set(2015, 11, 30);
425     Nag nag = new Nag();
426     date.set(2008, 0, 1);
427     nag.setStart(date.getTime());
428     date.set(2009, 0, 1);
429     nag.setEnd(date.getTime());
430     nag.setTime(23, 59);
431     nag.setMonday(true);
432     nag.setTuesday(true);
433     nag.setWednesday(true);
434     nag.setThursday(true);
435     nag.setChannel("xmpp");
436     User user = new User();
437     user.setEmailAddress("test@example.org");
438     persistenceFacade.save(user);
439     nag.setUser(user);
440     persistenceFacade.save(nag);
441
442     GregorianCalendar cal = new GregorianCalendar();
443     cal.set(2008, 6, 14, 23, 59);
444     GregorianCalendar calTo = new GregorianCalendar();
445     calTo.set(2008, 6, 14, 23, 59);
446     List<Nag> test = persistenceFacade.retrieveNags(cal.getTime(),
            calTo
447         .getTime());
448     assertEquals(nag, test.get(0));
449
450     // Should not get any when interval end is before nag end date.
451     date.set(2008, 0, 1);
452     nag.setEnd(date.getTime());
453     persistenceFacade.save(nag);
454     test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
            ());
455     assertEquals(0, test.size());
456
457     // End date should be greater than interval end date.
458     nag.setEnd(calTo.getTime());
459     persistenceFacade.save(nag);
460     test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
            ());
461     assertEquals(0, test.size());
462   }
463
464   /**
465    * Test that we don't get a nag with start date in the future.
466    */
467   @Test
468   public void testRetrieveNagsNotStarted()
469       throws ConstraintViolationException,
                InvalidEmailAddressException,
470        IntervalOutOfRangeException {
471     Calendar date = Calendar.getInstance();
472     date.set(2015, 11, 30);
473     Nag nag = new Nag();
474     date.set(2008, 0, 1);
475     nag.setStart(date.getTime());
476     date.set(2015, 0, 1);
477     nag.setEnd(date.getTime());
478     nag.setTime(23, 59);
479     nag.setMonday(true);
480     nag.setTuesday(true);
```

```
481     nag.setWednesday(true);
482     nag.setThursday(true);
483     nag.setChannel("xmpp");
484     User user = new User();
485     user.setEmailAddress("test@example.org");
486     persistenceFacade.save(user);
487     nag.setUser(user);
488     persistenceFacade.save(nag);
489
490     GregorianCalendar cal = new GregorianCalendar();
491     cal.set(2008, 6, 14, 23, 59);
492     GregorianCalendar calTo = new GregorianCalendar();
493     calTo.set(2008, 6, 14, 23, 59);
494
495     List<Nag> test = persistenceFacade.retrieveNags(cal.getTime(),
            calTo
496         .getTime());
497     assertEquals(nag, test.get(0));
498
499     // Do not get nags if it is in the future
500     date.set(2009, 0, 1);
501     nag.setStart(date.getTime());
502     persistenceFacade.save(nag);
503     test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
            ());
504     assertEquals(0, test.size());
505
506     // Get a nag if the interval is on start date.
507     nag.setStart(cal.getTime());
508     persistenceFacade.save(nag);
509     test = persistenceFacade.retrieveNags(cal.getTime(), calTo.getTime
            ());
510     assertEquals(nag, test.get(0));
511   }
512
513   @Test
514   public void retrieveMessagesByUser() throws InvalidMessageException,
515       ConstraintViolationException, InvalidEmailAddressException,
516       UserNotFoundException {
517     User systemUser = new User(Settings.SYSTEM_USER_EMAIL);
518     User user1 = new User("test@test.no");
519     User user2 = new User("test2@test.no");
520
521     Set<User> setofUsers = new HashSet<User>();
522     setofUsers.add(user1);
523     setofUsers.add(user2);
524
525     Message message1 = new Message(user1, setofUsers, "Testing");
526     message1.setTimestamp(new Date());
527     Message message2 = new Message(user1, user2, "Testing");
528     message2.setTimestamp(new Date());
529     Message message3 = new Message(user1, user1, "Testing");
530     message3.setTimestamp(new Date());
531
532     Message systemMessage1 = new Message(user1, systemUser,
533         "Sent system message");
534     Message systemMessage2 = new Message(systemUser, user1,
535         "Sent system message");
536
537     List<Message> listOfUser1Messages = new ArrayList<Message>();
538     listOfUser1Messages.add(message1);
539     listOfUser1Messages.add(message2);
```

```
540     listOfUser1Messages.add(message3);
541
542     List<Message> listOfUser2Messages = new ArrayList<Message>();
543     listOfUser2Messages.add(message1);
544     listOfUser2Messages.add(message2);
545
546     persistenceFacade.save(systemUser);
547     persistenceFacade.save(user1);
548     persistenceFacade.save(user2);
549
550     persistenceFacade.save(message1);
551     persistenceFacade.save(message2);
552     persistenceFacade.save(message3);
553     persistenceFacade.save(systemMessage1);
554     persistenceFacade.save(systemMessage2);
555
556     List<Message> listOfUser1SystemMessages = new ArrayList<Message>()
            ;
557     listOfUser1SystemMessages.add(systemMessage1);
558     listOfUser1SystemMessages.add(systemMessage2);
559
560     List<Message> result1 = persistenceFacade.retrieveMessages(user1);
561     List<Message> result2 = persistenceFacade.retrieveMessages(user2);
562
563     List<Message> result3 = persistenceFacade.retrieveConversation(
            user1,
564         user2);
565     List<Message> result4 = persistenceFacade.retrieveConversation(
            user1,
566         user1);
567     List<Message> result5 = persistenceFacade.retrieveConversation(
            user1,
568         systemUser);
569     List<Message> result6 = persistenceFacade.retrieveConversation(
570         systemUser, user1);
571
572     // result1 further down
573     assertEquals("result2 assertion: ", listOfUser2Messages, result2);
574     assertEquals("result3 assertion: ", listOfUser2Messages, result3);
575     // result4 further down
576     assertEquals("result5 assertion: ", listOfUser1SystemMessages,
            result5);
577     assertEquals("result6 assertion: ", listOfUser1SystemMessages,
            result6);
578
579     listOfUser1SystemMessages.addAll(listOfUser1Messages);
580     assertTrue(result1.containsAll(listOfUser1SystemMessages)
581         && result1.size() == listOfUser1SystemMessages.size());
582
583     listOfUser1Messages.remove(message2);
584     assertTrue(result4.containsAll(listOfUser1Messages)
585         && result4.size() == listOfUser1Messages.size());
586
587   }
588
589   /**
590    * Tests that the correct messages are retrieved when retrieving
           messages
591    * saved after a given timestamp.
592    *
593    * @throws InvalidMessageException
594    * @throws ConstraintViolationException
```

```
595     * @throws InvalidEmailAddressException
596     * @throws UserNotFoundException
597     */
598    @Test
599    public void retrieveMessagesByTimestamp() throws
           InvalidMessageException,
600        ConstraintViolationException, InvalidEmailAddressException,
601        UserNotFoundException {
602      long currentTime = System.currentTimeMillis();
603      User user1 = new User("test@test.no");
604
605      Message message1 = new Message(user1, user1, "Testing");
606      message1.setTimestamp(new Date(currentTime - 1000));
607      Message message2 = new Message(user1, user1, "Testing");
608      message2.setTimestamp(new Date(currentTime));
609      Message message3 = new Message(user1, user1, "Testing");
610      message3.setTimestamp(new Date(currentTime + 1000));
611
612      List<Message> listOfMessages = new ArrayList<Message>();
613      listOfMessages.add(message1);
614      listOfMessages.add(message2);
615      listOfMessages.add(message3);
616
617      persistenceFacade.save(user1);
618
619      persistenceFacade.save(message1);
620      persistenceFacade.save(message2);
621      persistenceFacade.save(message3);
622
623      // Three new messages
624      List<Message> result = persistenceFacade.retrieveMessages(new Date
           (
625          currentTime - 2000));
626      assertArrayEquals(result.toArray(), listOfMessages.toArray());
627
628      // Two new messages
629      result = persistenceFacade
630          .retrieveMessages(new Date(currentTime - 1000));
631      listOfMessages.remove(message1);
632      assertArrayEquals(result.toArray(), listOfMessages.toArray());
633
634      // No new messages
635      result = persistenceFacade
636          .retrieveMessages(new Date(currentTime + 1000));
637      listOfMessages.remove(message2);
638      listOfMessages.remove(message3);
639      assertArrayEquals(result.toArray(), listOfMessages.toArray());
640    }
641
642    /**
643     * Tests that the correct users are retrieved when retrieving users
           saved
644     * after a given timestamp.
645     *
646     * @throws InvalidMessageException
647     * @throws ConstraintViolationException
648     * @throws InvalidEmailAddressException
649     * @throws UserNotFoundException
650     */
651    @Test
652    public void retrieveUsersByTimestamp() throws
           InvalidMessageException,
```

```
653         ConstraintViolationException, InvalidEmailAddressException,
654         UserNotFoundException {
655       long currentTime = System.currentTimeMillis();
656       User user1 = new User("test1@test.no");
657       user1.setLastModified(new Date(currentTime - 1000));
658       User user2 = new User("test2@test.no");
659       user2.setLastModified(new Date(currentTime));
660       User user3 = new User("test3@test.no");
661       user3.setLastModified(new Date(currentTime + 1000));
662
663       List<User> listOfUsers = new ArrayList<User>();
664       listOfUsers.add(user1);
665       listOfUsers.add(user2);
666       listOfUsers.add(user3);
667
668       persistenceFacade.save(user1);
669       persistenceFacade.save(user2);
670       persistenceFacade.save(user3);
671
672       // Three new messages
673       List<User> result = persistenceFacade.retrieveUsers(new Date(
674           currentTime - 2000));
675       assertArrayEquals(result.toArray(), listOfUsers.toArray());
676
677       // Two new messages
678       result = persistenceFacade.retrieveUsers(new Date(currentTime -
              1000));
679       listOfUsers.remove(user1);
680       assertArrayEquals(result.toArray(), listOfUsers.toArray());
681
682       // No new messages
683       result = persistenceFacade.retrieveUsers(new Date(currentTime +
              1000));
684       listOfUsers.remove(user2);
685       listOfUsers.remove(user3);
686       assertArrayEquals(result.toArray(), listOfUsers.toArray());
687     }
688
689     /**
690      * Test that only sms messages which are not processed are retrieved
             .
691      *
692      * @throws ConstraintViolationException
693      * @throws InvalidDomainDataException
694      */
695     @Test
696     public void testRetrieveNewSmsMessages()
697         throws ConstraintViolationException, InvalidDomainDataException
                {
698       SmsMessage smsMessage1 = new SmsMessage("12345678", "Message1");
699       smsMessage1.setIdentifier("Message1");
700       smsMessage1.setProcessed(true);
701       SmsMessage smsMessage2 = new SmsMessage("12345678", "Message2");
702       smsMessage2.setIdentifier("Message2");
703       smsMessage2.setProcessed(false);
704
705       persistenceFacade.save(smsMessage1);
706       persistenceFacade.save(smsMessage2);
707
708       List<SmsMessage> smsMessages = new ArrayList<SmsMessage>();
709       smsMessages.add(smsMessage2);
710
```

```
711     List<SmsMessage> result = persistenceFacade.retrieveNewSmsMessages
            ();
712     assertArrayEquals(smsMessages.toArray(), result.toArray());
713
714     smsMessage2.setProcessed(true);
715     persistenceFacade.save(smsMessage2);
716     result = persistenceFacade.retrieveNewSmsMessages();
717     smsMessages.remove(smsMessage2);
718     assertArrayEquals(smsMessages.toArray(), result.toArray());
719   }
720
721   @Test
722   public void testRetrieveSmsMessageById() throws
          InvalidDomainDataException,
723       ConstraintViolationException, SmsMessageNotFoundException {
724     SmsMessage smsMessage1 = new SmsMessage("12345678", "Message1");
725     smsMessage1.setIdentifier("Message1");
726
727     smsMessage1.setProcessed(true);
728     persistenceFacade.save(smsMessage1);
729     SmsMessage result = persistenceFacade.retrieveSmsMessage("Message1
            ");
730     assertEquals(smsMessage1, result);
731
732     smsMessage1.setProcessed(false);
733     persistenceFacade.save(smsMessage1);
734     result = persistenceFacade.retrieveSmsMessage("Message1");
735     assertEquals(smsMessage1, result);
736   }
737
738   @Test(expected = SmsMessageNotFoundException.class)
739   public void testRetrieveSmsMessageByNonExistingId()
740       throws InvalidDomainDataException, ConstraintViolationException,
741       SmsMessageNotFoundException {
742     SmsMessage smsMessage1 = new SmsMessage("12345678", "Message1");
743     smsMessage1.setIdentifier("Message1");
744     smsMessage1.setProcessed(false);
745     persistenceFacade.save(smsMessage1);
746
747     SmsMessage result = persistenceFacade.retrieveSmsMessage("Message2
            ");
748     assertEquals(smsMessage1, result);
749   }
750 }
```

Listing E.4: HibernatePersistenceFacadeTest.java

QuerydslPersistenceFacadeTest.java

```
1  package no.iterate.leancast.dao;
2
3  import org.hibernate.classic.Session;
4
5  /**
6   * Subclass of {@link HibernatePersistenceFacadeTest} which
7   * uses {@link QuerydslPersistenceFacade} as the implementation.
8   *
9   * @author Stein Magnus Jodal (Master's Thesis)
10  */
11 public class QuerydslPersistenceFacadeTest extends
```

```
12      HibernatePersistenceFacadeTest {
13
14    @Override
15    public PersistenceFacade getPersistenceFacade(Session session) {
16      return new QuerydslPersistenceFacade(session);
17    }
18  }
```

Listing E.5: QuerydslPersistenceFacadeTest.java