Internet Applications Design and Implementation

(Lecture 4 - MVC & Persistence : JPA & Hibernate)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)

(with previous participations of Jácome Cunha (jacome@fct.unl.pt) and João Leitão (jc.leitao@fct.unl.pt))

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Outline

- Data Sources in the MVC Pattern
- Object Relational Mapping
- Spring & Data Abstraction
- JPA Associations
- Optimization of data fetching

Internet Applications Design and Implementation 2020 - 2021

(Lecture 4 - Part 2 - Data Sources in MVC)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)

(with previous participations of Jácome Cunha (jacome@fct.unl.pt) and João Leitão (jc.leitao@fct.unl.pt))

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Data Abstraction: The M in MVC

- An application layer that abstracts how information is stored, related, and protected.
- Examples of database languages, libraries and frameworks
  - JDBC
  - LINQ (in MVC ASP.NET)
  - ORMs (ActiveRecord in Rails, Hibernate in Java*)
  - NoSQL: MongoDB
  - External Web-services

JDBC

```
Connection conn = DriverManager.getConnection(
     "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
     "myLogin",
     "myPassword" );
try {
     /* you use the connection here */
} finally {
    //It's important to close the connection when you are done with it
    try { conn.close(); } catch (Throwable ignore) { /* Propagate the original exception
instead of this one that you may want just logged */ }
}
```

```
try (Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery( "SELECT * FROM MyTable" )
) {
    while ( rs.next() ) {
        int numColumns = rs.getMetaData().getColumnCount();
        for ( int i = 1 ; i <= numColumns ; i++ ) {
           // Column numbers start at 1.
           // Also there are many methods on the result set to return
           //  the column as a particular type. Refer to the Sun documentation
           //  for the list of valid conversions.
           System.out.println( "COLUMN " + i + " = " + rs.getObject(i) );
        }
    }
}
```

JDBC

- Basic API for Java defining an access to a database
- Does not know the database schema
- Programmer needs to “manually” translate between data formats

```
try (Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery( "SELECT * FROM MyTable" )
) {
     while ( rs.next() ) {
          int numColumns = rs.getMetaData().getColumnCount();
          for ( int i = 1 ; i <= numColumns ; i++ ) {
             // Column numbers start at 1.
             // Also there are many methods on the result set to return
             //  the column as a particular type. Refer to the Sun documentation
             //  for the list of valid conversions.
             System.out.println( "COLUMN " + i + " = " + rs.getObject(i) );
          }
     }
}
```

LINQ .NET

- Language based (integrated)
- Works in memory, xml, databases, etc
  - Based on the notion of **Provider**, it is extensible
- Programmers need to explicitly build objects that matches the database schema

```
var results =  from c in SomeCollection
               where c.SomeProperty < 10
               select new {c.SomeProperty, c.OtherProperty};

foreach (var result in results)
{
        Console.WriteLine(result);
}
```

LINQ .NET

- Language based (integrated)
- Works in memory, xml, databases, etc
  - Based on the notion of **Provider**, it is extensible
- Programmers need to explicitly build objects that matches the database schema

```
[Table(Name="Customers")]
public class Customer
{
      [Column(IsPrimaryKey = true)]
      public int CustID;

      [Column]
      public string CustName;
}
```

LINQ .NET

- Language based (integrated)
- Allows navigation using associations

```
// Query for customers who have placed orders.
var custQuery =
    from cust in Customers
    where cust.Orders.Any()
    select cust;

foreach (var custObj in custQuery)
{
    Console.WriteLine("ID={0}, Qty={1}", custObj.CustomerID,
        custObj.Orders.Count);
}
```

ActiveRecord in Rails

- Follows the active record pattern to implement an ORM. AR objects link to persistent data and define behaviour.
- Well integrated with the Model in the Rails MVC pattern
- Completely abstracts the database management by:
  - Object-mappings
  - Inheritance
  - Associations
  - Validations
  - Migrations (w/ Rails)

```
def create
  @author = Author.new(author_params)

  respond_to do |format|
    if @author.save
      format.html { redirect_to @author,
      format.json { render action: 'show'
    else
      format.html { render action: 'new'
      format.json { render json: @author.
    end
  end
end
```

NoSQL databases

- Not only SQL.
  - Document-based, Key-Value, Graph-based
- Designed for horizontal scaling (replication),
- Usually compromise data consistency
- Limited transactional support (real concurrency control)… use of shards
- Very good for querying but harder to get right on “writes”, indexes, etc.
- Queries are written in JavaScript, Java, SPARK, Map reduce algorithms…

Example: IndexedDB

```
let transaction = db.transaction("myObjectStore", "readwrite");
let objectStore = transaction.objectStore("myObjectStore");

let data = { id: 1, name: "John Doe", age: 25 };
let addRequest = objectStore.add(data);

addRequest.onsuccess = function(event) {
  // Data added successfully
};

let getRequest = objectStore.get(1);

getRequest.onsuccess = function(event) {
  let result = event.target.result;
  // Access the retrieved data
};
```

Example: MongoDB

```
db.orders.find({ status: "pending" })
```

```
{
    "_id": ObjectId("5f47a8ccef1f3b3dfb11b98c"),
    "clientId": 1,
    "orderDate": ISODate("2023-10-01T09:00:00Z"),
    "status": "pending",
    "items": [
      {"product": "Laptop", "quantity": 1},
      {"product": "Mouse", "quantity": 2}
    ]
  },
  {
    "_id": ObjectId("5f47a8cdef1f3b3dfb11b98d"),
    "clientId": 2,
    "orderDate": ISODate("2023-09-15T12:30:00Z"),
    "status": "shipped",
    "items": [
      {"product": "Monitor", "quantity": 1}
    ]
  },
  {
    "_id": ObjectId("5f47a8ceef1f3b3dfb11b98e"),
    "clientId": 1,
    "orderDate": ISODate("2023-10-05T10:15:00Z"),
    "status": "pending",
    "items": [
      {"product": "Keyboard", "quantity": 1}
    ]
  }
```

Source: Asked ChatGPT for an example

Example: MongoDB

```
db.orders.aggregate([
    {
      $match: { status: "pending" }
    },
    {
      $lookup: {
        from: "clients",
        localField: "clientId",
        foreignField: "clientId",
        as: "clientInfo"
      }
    },
    {
      $unwind: "$clientInfo"
    },
    {
      $project: {
        _id: 1,
        clientId: 1,
        "clientInfo.name": 1,
        "clientInfo.email": 1,
        orderDate: 1,
        status: 1,
        items: 1
      }
    }
  ])
```

```
[
    {
      "_id": ObjectId("5f47a8ccef1f3b3dfb11b98c"),
      "clientId": 1,
      "clientInfo": {
        "name": "Alice Johnson",
        "email": "alice@example.com"
      },
      "orderDate": ISODate("2023-10-01T09:00:00Z"),
      "status": "pending",
      "items": [
        {"product": "Laptop", "quantity": 1},
        {"product": "Mouse", "quantity": 2}
      ]
    },
    {
      "_id": ObjectId("5f47a8ceef1f3b3dfb11b98e"),
      "clientId": 1,
      "clientInfo": {
        "name": "Alice Johnson",
        "email": "alice@example.com"
      },
      "orderDate": ISODate("2023-10-05T10:15:00Z"),
      "status": "pending",
      "items": [
        {"product": "Keyboard", "quantity": 1}
      ]
    }
]
```

Data abstraction in Internet and Web Apps

- Abstraction over the actual data model
  - Hides the actual database engine running beneath
  - Integrates smoothly with the programming model (objects instead of string-based results).
- Independent configuration modes
  - Allows different execution modes with the same code (e.g. in-memory, transactional, replicated, etc.)
- Does not really cover all data models smoothly:
  - SQL model (e.g. Hibernate, ActiveRecord (Rn'R))
  - NoSQL model (e.g. google app engine framework)

Abstraction levels

- Connectivity (e.g. JDBC)
  - abstracts the connectivity and execution of queries.
  - you have to build queries, and parse and translate results
- Data translation (e.g. JPA, ActiveRecord, LINQ)
  - Translates data formats between program and database.
  - Integrates the language values typefully.
- Implementation and execution modes
  - Example: Google Cloud Datastore is a NoSQL document based storage that allows different implementations (cassandra, mongodb)

https://cloud.google.com/appengine/docs/java/datastore/

Example with Google Data Store and Kotlin

```kotlin
// Query for pending orders
val orderQuery = Query.newEntityQueryBuilder()
    .setKind("Order")
    .setFilter(StructuredQuery.PropertyFilter.eq("status", "pending"))
    .build()

// Fetch the orders
val pendingOrders = datastore.run(orderQuery).asSequence().toList()

// Print pending orders
pendingOrders.forEach { order ->
    val orderId = order.key.id
    val clientId = order.getLong("clientId")
    val status = order.getString("status")
    val items = order.getList<Struct>("items")

    println("Order ID: $orderId, Client ID: $clientId, Status: $status, Items: $items")

    // Optionally fetch and print client data
    val clientKey: Key = datastore.newKeyFactory().setKind("Client").newKey(clientId)
    val clientEntity: Entity? = datastore.get(clientKey)

    if (clientEntity != null) {
        val clientName = clientEntity.getString("name")
        val clientEmail = clientEntity.getString("email")
        println("Client Name: $clientName, Client Email: $clientEmail")
    }
}
```

Example with MeteorJS (tierless)

```
import React from "react";
import { useTracker } from "meteor/react-meteor-data";
import { TasksCollection } from "/imports/api/TasksCollection";
import { Task } from "./Task";

export const App = () => {
  const tasks = useTracker(() => TasksCollection.find({}).fetch());

  return (
    <div>
      <h1>Welcome to Meteor!</h1>

      <ul>
        {tasks.map((task) => (
          <Task key={task._id} task={task} />
        ))}
      </ul>
    </div>
  );
};
```

https://docs.meteor.com/tutorials/react/

Internet Applications Design and Implementation 2020 - 2021

(Lecture 4 - Part 3 - Object Relational Mapping)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)

(with previous participations of Jácome Cunha (jacome@fct.unl.pt) and João Leitão (jc.leitao@fct.unl.pt))

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Database usage panorama

- Most databases are based on relational algebra
- Applications define object oriented representations (object graphs)

397 systems in ranking, October 2022

Rank			DBMS	Database Model	Score		
Oct 2022	Sep 2022	Oct 2021			Oct 2022	Sep 2022	Oct 2021
1.	1.	1.	Oracle	Relational, Multi-model	1236.37	-1.88	-33.98
2.	2.	2.	MySQL	Relational, Multi-model	1205.38	-7.09	-14.39
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	924.68	-1.62	-45.93
4.	4.	4.	PostgreSQL	Relational, Multi-model	622.72	+2.26	+35.75
5.	5.	5.	MongoDB	Document, Multi-model	486.23	-3.40	-7.32
6.	6.	6.	Redis	Key-value, Multi-model	183.38	+1.91	+12.03
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	151.07	-0.37	-7.19
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	149.66	-1.73	-16.30
9.	9.	↑ 11.	Microsoft Access	Relational	138.17	-1.87	+21.79
10.	10.	↓ 9.	SQLite	Relational	137.80	-1.02	+8.43

https://db-engines.com/en/ranking

Database usage panorama

- Most databases are based on relational algebra
- Applications define object oriented representations (object graphs)

415 systems in ranking, October 2023

Rank			DBMS	Database Model	Score		
Oct 2023	Sep 2023	Oct 2022			Oct 2023	Sep 2023	Oct 2022
1.	1.	1.	Oracle	Relational, Multi-model	1261.42	+20.54	+25.05
2.	2.	2.	MySQL	Relational, Multi-model	1133.32	+21.83	-72.06
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	896.88	-5.34	-27.80
4.	4.	4.	PostgreSQL	Relational, Multi-model	638.82	+18.06	+16.10
5.	5.	5.	MongoDB	Document, Multi-model	431.42	-8.00	-54.81
6.	6.	6.	Redis	Key-value, Multi-model	162.96	-0.72	-20.41
7.	7.	7.	Elasticsearch	Search engine, Multi-model	137.15	-1.84	-13.92
8.	8.	8.	IBM Db2	Relational, Multi-model	134.87	-1.85	-14.79
9.	9.	↑ 10.	SQLite	Relational	125.14	-4.06	-12.66
10.	10.	↓ 9.	Microsoft Access	Relational	124.31	-4.25	-13.85

https://db-engines.com/en/ranking

Database usage panorama

- Most databases are based on relational algebra
- Applications define object oriented representations (object graphs)

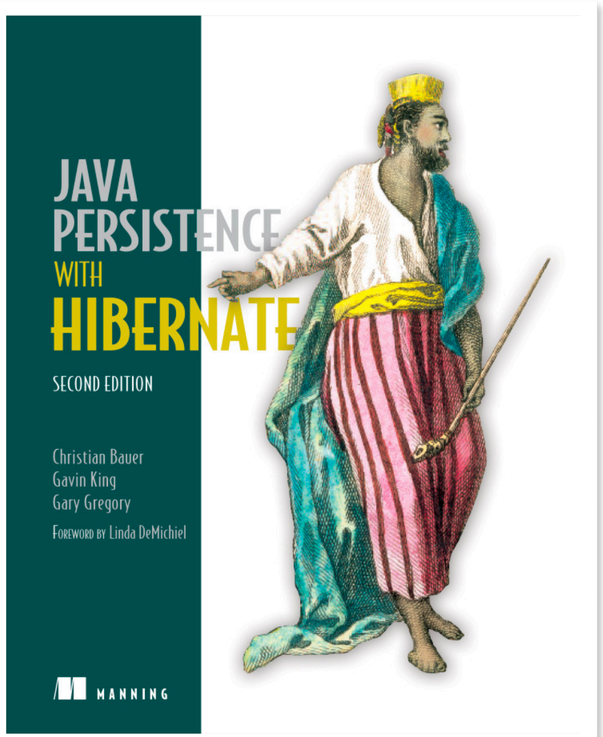
423 systems in ranking, October 2024

Rank Oct 2024	Rank Sep 2024	Rank Oct 2023	DBMS	Database Model	Score Oct 2024	Score Sep 2024	Score Oct 2023
1.	1.	1.	Oracle	Relational, Multi-model	1309.45	+22.85	+48.03
2.	2.	2.	MySQL	Relational, Multi-model	1022.76	-6.73	-110.56
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	802.09	-5.67	-94.79
4.	4.	4.	PostgreSQL	Relational, Multi-model	652.16	+7.80	+13.34
5.	5.	5.	MongoDB	Document, Multi-model	405.21	-5.02	-26.21
6.	6.	6.	Redis	Key-value, Multi-model	149.63	+0.20	-13.33
7.	7.	↑ 11.	Snowflake	Relational	140.60	+6.88	+17.36
8.	8.	↓ 7.	Elasticsearch	Multi-model	131.85	+3.06	-5.30
9.	9.	↓ 8.	IBM Db2	Relational, Multi-model	122.77	-0.28	-12.10
10.	10.	↓ 9.	SQLite	Relational	101.91	-1.43	-23.23

https://db-engines.com/en/ranking

Object-Relational Mapping

- Manages persistence of data in the OO realm
- Abstracts the use of SQL in connection to RDBMSs
- Covers most injection attacks on queries
  (except with APIs that allow creation with string)
- Specified by a common and standard API (JPA)
- Implementations differ by JPA providers




```
public List<AccountDTO> unsafeJpaFindAccountsByCustomerId(String customerId) {
    String jql = "from Account where customerId = '" + customerId + "'";
    TypedQuery<Account> q = em.createQuery(jql, Account.class);
    return q.getResultList()
      .stream()
      .map(this::toAccountDTO)
      .collect(Collectors.toList());
}
```

Object-Relational Impedance Mismatch

- Identity
  - Object: a == b and a.equals(b)
  - Relational: primary key based
- Inheritance
  - Object: natural relation
  - Relational: does not exist, idioms are necessary
- Accessing data
  - Object: through the object interface
  - Relational: select queries (with joins)
- Associations/Navigation
  - Object: unidirectional references through objects' interface
  - Relational: through foreign keys
- Granularity
  - In some cases there may be a difference in the granularity level

JPA - Java Persistence API

- Java Persistence API (Jakarta Persistence since Set 2019)
- Provides a OO interface to a relational database
- Defines JPQL (Java Persistence Query Language)
- The reference implementation is EclipseLink.
- Hibernate is a JPA provider (extended API and query language HQL)

Data Abstraction layers

- Advantages
  - Hides the complexity of a particular query language
  - Allows the portability of database engines (not really models)
  - Prevents attacks such as SQL injection, or XSS
  - Avoids runtime errors in the construction of queries
  - May isolate efficient implementations (previously, with prepared and compiled parameterised SQL queries)
  - Allows scalability via customised runtime configurations (distribution, transactional behaviour, indexing, …)
  - Avoids early optimization pitfalls, develop first, configure later.
- Pitfalls
  - Lack of access to proprietary features of providers
  - May lead to inefficient data transmissions: more queries (N+1 queries), more data than needed.

OpenJPA

JDO
Java Data Objects

eclipselink

CakePHP

Microsoft .NET
Entity
Framework

ORM

Object-Relational Impedance Mismatch

- What's the difference?
  - Encapsulation
  - Interface, class, polymorphism
  - Mapping relational concepts
  - Data type differences
  - Structural and integrity differences
  - Transactional differences

http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

https://dev.to/alagrede/why-i-dont-want-use-jpa-anymore-fl

Object-Relational Impedance Mismatch

RECAP

- Identity
  - Object: a == b and a.equals(b)
  - Relational: primary key based
- Inheritance
  - Object: natural relation
  - Relational: does not exist, idioms are necessary
- Accessing data
  - Object: through the object interface
  - Relational: select queries (with joins)
- Associations/Navigation
  - Object: unidirectional references through objects' interface
  - Relational: through foreign keys
- Granularity
  - In some cases there may be a difference in the granularity level

http://hibernate.org/orm/what-is-an-orm/

Object-Relational Impedance Mismatch

- Identity based on Primary Keys,
- We must define method `equals` (or utils like Lombok).

```java
@Entity
public class Person {

    @Id
    @GeneratedValue
    private long id;

    private String name;

    public Person() {}

    public long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Object-Relational Impedance Mismatch

- Inheritance: optional support for inheritance is provided by some frameworks

```
@Entity
@Inheritance(strategy= InheritanceType.JOINED)
@JsonSubTypes({
       @JsonSubTypes.Type(value = Professor.class, name = "PROFESSOR"),
        @JsonSubTypes.Type(value = Staff.class, name = "STAFF"),
        @JsonSubTypes.Type(value = Student.class, name = "STUDENT")
})
public class User {

   @Id
   @GeneratedValue
   private Long id_login;

   private String name;
…
}
```

```
@Entity
public class Student extends User {

    @Column
    private int number;

    public Student() {
        super();
    }

    public Student(String login,
                   String password,
                   String name,
                   String tel,
                   String email,
                   String address,
                   String type,
                   int number) {

        super( login,
               password,
               name,
               tel,
               email,
               address,
               type);
        this.number = number;
    }

    public int getNumber() {
        return number;
    }
}
```

Object-Relational Impedance Mismatch

- Inheritance: optional support for inheritance is provided by some frameworks

```
@Entity
@Inheritance(strategy= InheritanceType.JOINED)
@JsonSubTypes({
       @JsonSubTypes.Type(value = Professor.class, name = "PROFESSOR"),
        @JsonSubTypes.Type(value = Staff.class, name = "STAFF"),
        @JsonSubTypes.Type(value = Student.class, name = "STUDENT")
})
public class User {

   @Id
   @GeneratedValue
   private Long id_login;

   private
…
}
```

```
@Entity
public class Student extends User {

    @Column
    private int number;

    public Student() {
        super();
    }

    public Student(String login,
                   String password,
                   String name,
                   String tel,
                   String email,
                   String address,
                   String type,
                   int number) {

        super( login,
               password,
               name,
               tel,
               email,
               address,
```

- *MappedSuperclass* – the parent classes, can't be entities
- Single Table – the entities from different classes with a common ancestor are placed in a single table
- Joined Table – each class has its table and querying a subclass entity requires joining the tables
- Table-Per-Class – all the properties of a class, are in its table, so no join is required

https://www.baeldung.com/hibernate-inheritance

Object-Relational Impedance Mismatch

- Access to data by object navigation
- All fields are retrieved to memory instead of explicitly selected
- May result in navigation queries

```
Organization o = organizationRepository.findById(id).get();
System.out.println(o.getName() + o.getContactInfo());
```

```
@Entity
public class Organization {

    @Id
    @GeneratedValue
    private Long id_entity;

    @Column
    private String name;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "contact_id")
    private ContactInfo contactInfo;

    public Organization(){ }

   public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public ContactInfo getContactInfo() {
        return contactInfo;
    }

    public void setContactInfo(ContactInfo contactInfo) {
        this.contactInfo = contactInfo;
    }
    …
}
```

ActiveRecord Example

```ruby
class Album < ActiveRecord::Base
    has_many :tracks
end

class Track < ActiveRecord::Base
    belongs_to :album
end

album = Album.create(:title => 'Black and Blue', :performer => 'The Rolling Stones')
album.tracks.create(:track_number => 1, :title => 'Hot Stuff')
album.tracks.create(:track_number => 2, :title => 'Hand Of Fate')
album.tracks.create(:track_number => 3, :title => 'Cherry Oh Baby ')
album.tracks.create(:track_number => 4, :title => 'Memory Motel ')
album.tracks.create(:track_number => 5, :title => 'Hey Negrita')
album.tracks.create(:track_number => 6, :title => 'Fool To Cry')
album.tracks.create(:track_number => 7, :title => 'Crazy Mama')
album.tracks.create(:track_number => 8, :title => 'Melody (Inspiration By Billy
Preston)')

album = Album.create(:title => 'Sticky Fingers',:performer => 'The Rolling Stones')
album.tracks.create(:track_number => 1, :title => 'Brown Sugar')
album.tracks.create(:track_number => 2, :title => 'Sway')
album.tracks.create(:track_number => 3, :title => 'Wild Horses')
album.tracks.create(:track_number => 4,:title => 'Can\'t You Hear Me Knocking')
album.tracks.create(:track_number => 5, :title => 'You Gotta Move')
album.tracks.create(:track_number => 6, :title => 'Bitch')
album.tracks.create(:track_number => 7, :title => 'I Got The Blues')
album.tracks.create(:track_number => 8, :title => 'Sister Morphine')
album.tracks.create(:track_number => 9, :title => 'Dead Flowers')
album.tracks.create(:track_number => 10, :title => 'Moonlight Mile')
```

https://dzone.com/articles/simple-ruby-activerecord

Object-Relational Mapping

- In Java you navigate the object network.
- Not efficient to retrieve data from a RDBMS.
- Minimize the number of SQL queries by using JOINs and selecting the targeted entities from the start (pre-fetching).

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html#queryhql-joins

Internet Applications Design and Implementation 2020 - 2021

(Lecture 4 - Part 4 - Spring & Data Abstraction)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)

(with previous participations of Jácome Cunha (jacome@fct.unl.pt) and João Leitão (jc.leitao@fct.unl.pt))

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

JDBC & Spring

- SpringBoot provides direct support for JDBC
- As in any JDBC setting, it is necessary to define a DataSource (DB, CSV file, etc.)
- SpringBoot offers the JdbcTemplate class to assist programmers
- Spring easily integrates JPA support (later)

JDBC & Spring & In-memory DBs

- JdbcTemplate handles the setup and connection to the DataSource based on the POM dependencies (when possible)

- For instance, for H2 in-memory database (Spring also supports HSQL and Derby):

```
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
```

- This handles the connection and exceptions

JDBC & Spring

- It also supports “regular” relational databases adding the necessary properties to the **`application.properties`** file, e.g.:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
```

JDBC & Spring

```
@Component
public class Hotels {

    private JdbcTemplate jdbc;

    @Autowired
    public Hotels(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
        createTable();
    }

    public Hotels(){}

    public void createTable() {
        jdbc.execute("drop table tablea if exists");
        jdbc.execute("create table tablea(id SERIAL, attributea VARCHAR(16))");
    }

    public List<Map<String,Object>> select() {
        return jdbc.query("select * from tablea", new ColumnMapRowMapper());
    }

    public int save(String att) {
        return jdbc.update("insert into tablea(attributea) values (?)", att);
    }
}
```

JPA & Spring

- Spring-boot-starter-data-jpa provides the necessary dependencies:
  - Hibernate — One of the most popular JPA implementations
  - Spring Data JPA — Makes it easy to implement JPA-based repositories
  - Spring ORMs — Core ORM support from the Spring Framework

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
```

Declare an Entity (in Java)

```java
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    @Override
    public String toString() {
        return String.format(
                "Customer[id=%d, firstName='%s', lastName='%s']",
                id, firstName, lastName);
    }
}
```

Declare a Repository and plug it in… (in Java)

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

    List<Customer> findByLastName(String lastName);
}
```

```
@Controller
public class CustomerController
{
    @Autowired
    CustomerRepository customers;

   … findByLastName(“Smith”);…
}
```

Declare an Entity and a Repository and plug it in…

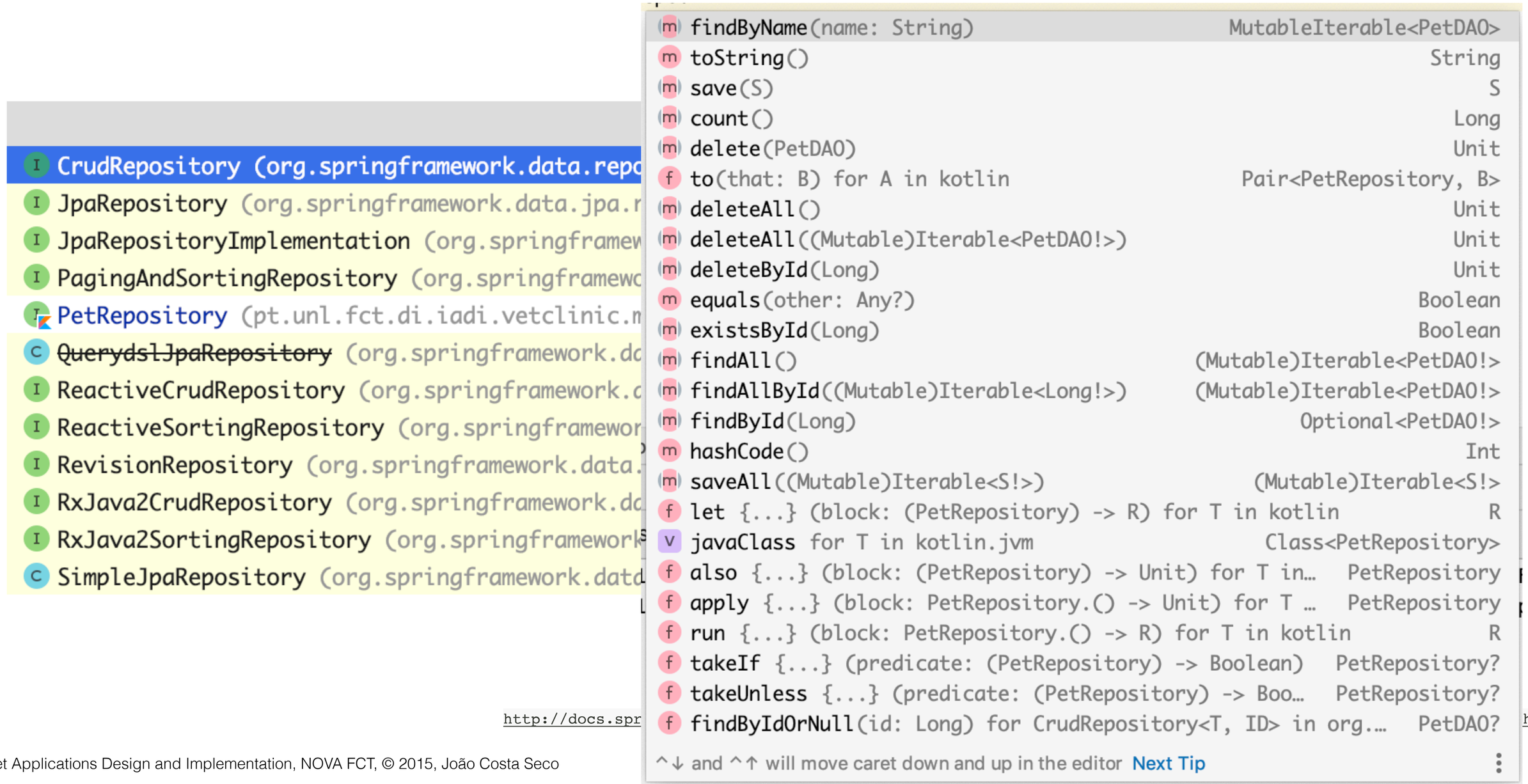
```
@Entity
data class PetDAO(
        @Id @GeneratedValue val id:Long,
        var name: String,
        var species: String
)
```

```
interface PetRepository : CrudRepository<PetDAO, Long> {
    fun findByName(name:String): MutableIterable<PetDAO>
}
```

```
@Autowired
lateinit var repo:PetRepository

fun someFunction() {

…repo.findByName(“pantufas”)…

}
```

```
findByName(name: String)                                  MutableIterable<PetDAO>
toString()                                                                 String
save(S)                                                                         S
count()                                                                     Long
delete(PetDAO)                                                              Unit
to(that: B) for A in kotlin                                 Pair<PetRepository, B>
deleteAll()                                                                 Unit
deleteAll((Mutable)Iterable<PetDAO!>)                                       Unit
                                                                            Unit
                                                                         Boolean
                                                                         Boolean
                                                  (Mutable)Iterable<PetDAO!>
                                (Mutable)Iterable<Long!>)  (Mutable)Iterable<PetDAO!>
                                                          Optional<PetDAO!>
hashCode()                                                                  Int
saveAll((Mutable)Iterable<S!>)                        (Mutable)Iterable<S!>
let {...} (block: (PetRepository) -> R) for T in kotlin                     R
javaClass for T in kotlin.jvm                          Class<PetRepository>
also {...} (block: (PetRepository) -> Unit) for T in…    PetRepository
apply {...} (block: PetRepository.() -> Unit) for T …    PetRepository
run {...} (block: PetRepository.() -> R) for T in kotlin                    R
takeIf {...} (predicate: (PetRepository) -> Boolean)     PetRepository?
takeUnless {...} (predicate: (PetRepository) -> Boo…     PetRepository?
findByIdOrNull(id: Long) for CrudRepository<T, ID> in org.…    PetDAO?
^↓ and ^↑ will move caret down and up in the editor  Next Tip
```

interface CrudRepository<T , ID extends Serializable>



http://docs.spr html

An Example

```
@Entity
@NamedQuery(name = "User.findByTheUsersName",
            query = "from User u where u.username = ?1")
class User(
            @Column(unique = true)
            val username:String,
            val firstname:String,
            val lastname:String
)
```

```
interface SimpleUserRepository : CrudRepository<User, Long> {

    fun findByTheUsersName(username:String):User

    fun findByLastname(lastname:String):List<User>

    @Query("select u from User u where u.firstname = ?")
    fun findByFirstname(firstname:String):List<User>

    @Query("select u from User u where u.firstname = :name or u.lastname = :name")
    fun findByFirstnameOrLastname(@Param("name") name:String):List<User>
}
```

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname =
Is,Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = 1?
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age ⇐ ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
TRUE	findByActiveTrue()	… where x.active = true
FALSE	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?

Spring Data (and other ORM Implementations)

SUMMARY

- Simple declaration of generic queries
- Specific declaration of custom queries (JPQL)
- Richer Behaviour from Repositories (e.g. paged)

```
public interface StudentRepository extends PagingAndSortingRepository<Student, Long> {

    List<Student> findByName(String name);

    @Query("select s from Student s where s.name like CONCAT(?,'%')")
    List<Student> search(String name);

    Page<Student> findByName(String name, Pageable pageable);
}
```

- Dependency Injection and component assembly

```
public class StudentController {

    @Autowired
    StudentRepository students;
```

Spring Data (and other ORM Implementations)

SUMMARY

- Simple declaration of generic queries
- Specific declaration of custom queries (JPQL)
- Richer Behaviour from Repositories (e.g. paged)

```
public interface StudentRepository extends PagingAndSortingRepository<Student, Long> {

    List<Student> findByName(String name);
```

```
@RequestMapping(value= "/page")
public @ResponseBody Page<Student> getStudentsPaged(
     @RequestParam(required=false, defaultValue =  "0") Integer page,
     @RequestParam(required=false, defaultValue = "3") Integer size) {

   return students.findAll(new PageRequest(page, size));
}
```

- Depe

```
    @Autowired
    StudentRepository students;
```

Spring Data (and other ORM Imp

SUMMARY

- Simple declaration of generic queries
- Specific declaration of custom queries (JP
- Richer Behaviour from Repositories (e.g. p
- Depe

```
public interface StudentRepository extends Pag

    List<Student> findByName(String name);
```

```
@RequestMapping(value= "/page")
public @ResponseBody Page<Student> ge
     @RequestParam(required=false, de
     @RequestParam(required=false, de

   return students.findAll(new PageRe
}
```

```
@Autowired
StudentRepository students;
```

```
{
  "content": [
    {
      "id": 1,
      "name": "Ingrid Daubechies",
      "age": 19
    },
    {
      "id": 2,
      "name": "Jacqueline K. Barton",
      "age": 18
    },
    {
      "id": 3,
      "name": "Jane Goodall",
      "age": 20
    }
  ],
  "last": false,
  "totalElements": 6,
  "totalPages": 2,
  "size": 3,
  "number": 0,
  "sort": null,
  "first": true,
  "numberOfElements": 3
}
```

Spring Data - Features

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Implementation domain base classes providing basic properties
- Support for transparent auditing (created, last changed)
- Possibility to integrate custom repository code
- Easy Spring integration via JavaConfig and custom XML namespaces
- Advanced integration with Spring MVC controllers
- Experimental support for cross-store persistence

Spring Data

- Spring Data Commons - Core Spring concepts underpinning every Spring Data project.
- Spring Data JPA - Makes it easy to implement JPA-based repositories.
- Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.
- Spring Data Redis - Provides easy configuration and access to Redis from Spring applications.
- Spring Data Solr - Spring Data module for Apache Solr.
- Spring Data Gemfire - Provides easy configuration and access to GemFire from Spring applications.
- Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.

Community Modules

- Spring Data Cassandra - Spring Data module for Apache Cassandra.
- Spring Data Couchbase - Spring Data module for Couchbase.
- Spring Data DynamoDB - Spring Data module for DynamoDB.
- Spring Data Elasticsearch - Spring Data module for Elasticsearch.
- Spring Data Neo4j - Spring based, object-graph support and repositories for Neo4j.

MongoDB Example

```
public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    MongoOperations mongoOps = new MongoTemplate(new Mongo(), "database");
    mongoOps.insert(new Person("Joe", 34));

    log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

    mongoOps.dropCollection("person");
  }
}
```

Other database representations

- Objectify: gives easy and full access to the Google Cloud Datastore
- Dynamic / Reflexion based
- Alternative to JPA interface

```
@Entity
class Car {
    @Id String vin; // Can be Long, long, or String
    String color;
}

ofy().save().entity(new Car("123123", "red")).now();
Car c = ofy().load().type(Car.class).id("123123").now();
ofy().delete().entity(c);
```

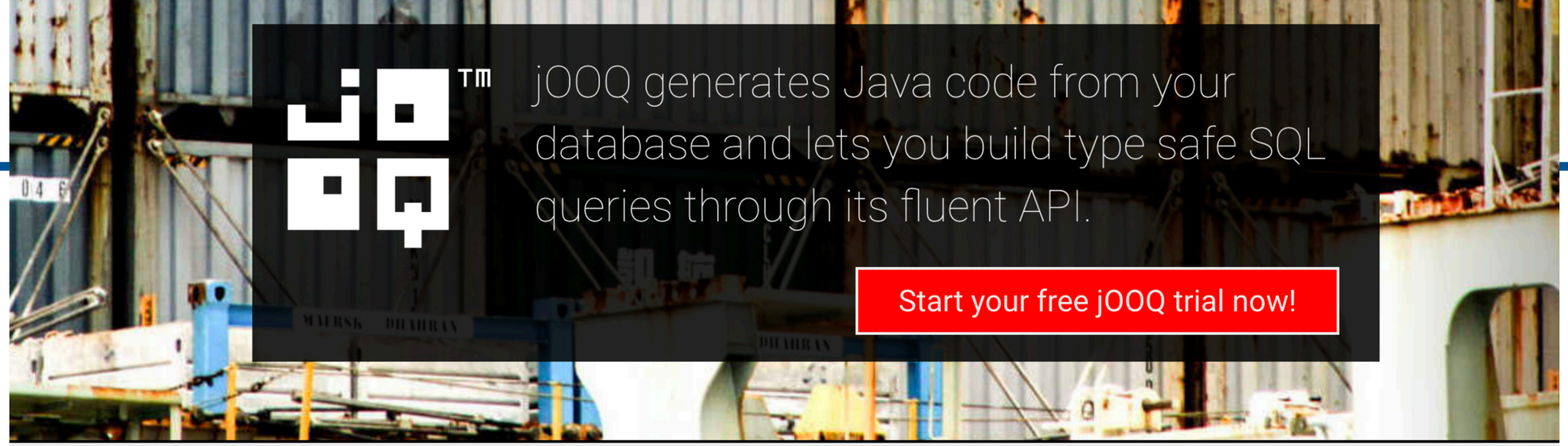
https://github.com/objectify/objectify

Other database representations

- Objectify: gives easy and full access to the Google Cloud Datastore
- Dynamic / Reflexion based
- Alternative to JPA interface

- Objectify surfaces **all native datastore features**, including batch operations, queries, transactions, asynchronous operations, and partial indexes.
- Objectify provides **type-safe key and query classes** using Java generics.
- Objectify provides a **human-friendly query interface**.
- Objectify can automatically **cache your data in memcache** for improved read performance.
- Objectify can store polymorphic entities and perform **true polymorphic queries**.
- Objectify provides a simple, **easy-to-understand transaction model**.
- Objectify provides built-in facilities to **help migrate schema changes** forward.
- Objectify provides **thorough documentation** of concepts as well as use cases.
- Objectify has an **extensive test suite** to prevent regressions.

https://github.com/objectify/objectify



Great Reasons for Using jOOQ

Our customers spend most time on *their* business-logic.
Because jOOQ takes care of all their Java/SQL infrastructure problems.

https://www.jooq.org/

Database First

Tired of ORMs driving your database model?

Whether you design a new application or integrate with your legacy, your database holds your most important asset: your data.

jOOQ is SQL-centric. Your database comes "first".

Typesafe SQL

Fed up with detecting SQL syntax errors in production?

SQL is a highly expressive and type safe language with a rich syntax. jOOQ models SQL as an internal DSL and uses the Java compiler to compile your SQL syntax, metadata and data types.

Code Generation

Bored with renaming table and column names in your Java code?

jOOQ generates Java classes from your database metadata. Your Java compiler will tell you when your code is out of sync with your schema.

Active Records

Annoyed by the amount of SQL you write for CRUD?

jOOQ lets you perform CRUD and POJO mapping directly on Active Records, which are also generated from the code generator.

Internet Applications Design and Implementation 2020 - 2021

(Lecture 4 - Part 5 - JPA Associations)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)

(with previous participations of Jácome Cunha (jacome@fct.unl.pt) and João Leitão (jc.leitao@fct.unl.pt))

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
       @ManyToOne
        val kind:Category
)

@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @OneToMany
        val books:Set<Book>
)
```

```
interface BookRepository : CrudRepository<Book,Long>
interface CategoryRepository: CrudRepository<Category,Long>
…
val fantasy = Category(0, “Fantasy", emptySet<Book>())
categories.save(fantasy)
val lor = Book(0,"Lord of the Rings", fantasy)
books.save(lor)
```

```
call next value for hibernate_sequence
insert into category (name, id) values (?, ?)
binding parameter [1] as [VARCHAR] - [Fantasy]
binding parameter [2] as [BIGINT] - [1]
call next value for hibernate_sequence
insert into book (kind_id, name, id) values (?, ?, ?)
binding parameter [1] as [BIGINT] - [1]
binding parameter [2] as [VARCHAR] - [Lord of the Rings]
binding parameter [3] as [BIGINT] - [2]
```

JPA Associations

```java
@Entity
@Table(name = "book_category")
public class BookCategory {
    private int id;
    private String name;
    private Set<Book> books;

    public BookCategory(){ … }

    public BookCategory(String name) {…}

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { … }

    public void setId(int id) { … }

    public String getName() {…}

    public void setName(String name) {…}

    @OneToMany(mappedBy = "bookCategory",
               cascade = CascadeType.ALL)
    public Set<Book> getBooks() { … }

    public void setBooks(Set<Book> books) { … }
}
```

```java
@Entity
public class Book{
    private int id;
    private String name;
    private BookCategory bookCategory;

    public Book() {}

    public Book(String name) { this.name = name;}

    public Book(String name, BookCategory bookCategory) { … }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { … }

    public void setId(int id) { … }

    public String getName() { … }

    public void setName(String name) { … }

    @ManyToOne
    @JoinColumn(name = "book_category_id")
    public BookCategory getBookCategory() { … }

    public void setBookCategory(BookCategory bookCategory) {…}
}
```

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @ManyToOne
        val kind:Category
)

@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @OneToMany
        val books:Set<Book>
)
```

```
interface BookRepository : CrudRepository<Book,Long>
interface CategoryRepository: CrudRepository<Category,Long>
…
val book = books.findById(lor.id)
logger.info(book.get().kind.toString())
```

```
select book0_.id as id1_0_0_,
       book0_.kind_id as kind_id3_0_0_,
       book0_.name as name2_0_0_,
       category1_.id as id1_1_1_,
       category1_.name as name2_1_1_
from book book0_
left outer join category category1_ on
   book0_.kind_id=category1_.id where book0_.id=?
```

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @ManyToOne
        val kind:Category
)

@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @OneToMany
        val books:Set<Book>
)
```

```
interface BookRepository : CrudRepository<Book,Long>
interface CategoryRepository: CrudRepository<Category,Long>
…
val kind = categories.findById(fantasy.id)
```

```
select
   category0_.id as id1_1_0_,
   category0_.name as name2_1_0_
from category category0_
where category0_.id=?
```

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,

        @OneToMany
        val kind:Category
)

@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @ManyToOne
        val books:Set<Book>
)
```

```
interface BookRepository : CrudRepository<Book,Long>
interface CategoryRepository: CrudRepository<Category,Long>
…
val kind = categories.findById(fantasy.id)
logger.info(kind.toString())
```

```
failed to lazily initialize a collection of role:
com.demo.Category.books
```

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @ManyToOne
        val kind:Category
)

@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        val name:String,
        @OneToMany(fetch = FetchType.EAGER)
        val books:Set<Book>
)
```

```
interface BookRepository : CrudRepository<Book,Long>
interface CategoryRepository: CrudRepository<Category,Long>
…
val kind = categories.findById(fantasy.id)
logger.info(kind.toString())
```

```
Optional[Category(id=1, name=Fantastic, books=[])]
```

```
select
category0_.id as id1_1_0_,
category0_.name as name2_1_0_,
books1_.category_id as category1_2_1_,
book2_.id as books_id2_2_1_,
book2_.id as id1_0_2_,
book2_.kind_id as kind_id3_0_2_,
book2_.name as name2_0_2_,
category3_.id as id1_1_3_,
category3_.name as name2_1_3_
from category category0_
left outer join category_books books1_ on category0_.id=books1_.category_
left outer join book book2_ on books1_.books_id=book2_.id
left outer join category category3_ on book2_.kind_id=category3_.id
where category0_.id=?
```

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        @Column(nullable = false)
        val name:String,
        @ManyToOne
        val kind:Category
)
@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        @Column(nullable = false)
        val name:String,
        @OneToMany(cascade = arrayOf(CascadeType.ALL), mappedBy = "kind", fetch = FetchType.EAGER)
        var books:List<Book>
)
```

```
val fantasy = Category(0, "Fantasy", emptyList<Book>())
val lor = Book(0,"Lord of the Rings", fantasy)
val silm = Book(0,"Silmarillion", fantasy)
fantasy.books = listOf(lor,silm)
categories.save(fantasy)
```

```
insert into category (name, id) values (?, ?)
binding parameter [1] as [VARCHAR] - [Fantasy]
binding parameter [2] as [BIGINT] - [1]
insert into book (kind_id, name, id) values (?, ?, ?)
binding parameter [1] as [BIGINT] - [1]
binding parameter [2] as [VARCHAR] - [Lord of the Rings]
binding parameter [3] as [BIGINT] - [2]
insert into book (kind_id, name, id) values (?, ?, ?)
binding parameter [1] as [BIGINT] - [1]
binding parameter [2] as [VARCHAR] - [Silmarillion]
binding parameter [3] as [BIGINT] - [3]
```

```
Category(id=1, name=Fantasy, books=[{ Lord of the Rings, Fantasy }, { Silmarillion, Fantasy }])
```

JPA Associations

```
@Entity
data class Book(
        @Id
        @GeneratedValue
        val id:Long,
        @Column(nullable = false)
        val name:String,
        @ManyToOne
        val kind:Category
)
@Entity
data class Category(
        @Id
        @GeneratedValue
        val id:Long,
        @Column(nullable = false)
        val name:String,
        @OneToMany(cascade = arrayOf(CascadeType.ALL), mappedBy = "kind", fetch = FetchType.EAGER)
        var books:List<Book>
)
```

```
val kind = categories.findById(fantasy.id)
logger.info(kind.toString())
logger.info(kind.get().books.size.toString())
for (b in kind.get().books) {
    logger.info(b.toString())
    logger.info(b.kind.toString())
}
```

```
select
    category0_.id as id1_1_0_,
    category0_.name as name2_1_0_,
    books1_.kind_id as kind_id3_0_1_,
    books1_.id as id1_0_1_,
    books1_.id as id1_0_2_,
    books1_.kind_id as kind_id3_0_2_,
    books1_.name as name2_0_2_
from category category0_
left outer join book books1_ on
    category0_.id=books1_.kind_id where category0_.id=?
```

```
Category(id=1, name=Fantasy, books=[{ Lord of the Rings, Fantasy }, { Silmarillion, Fantasy }])
```

Eager and Lazy

- Evaluation modes - transferring objects to memory

```
@OneToMany(mappedBy = "course", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
private Set<Enrollment> enrollments;
```

- Eager: transfers all objects related to the root

```
@OneToMany(mappedBy = "course", cascade = CascadeType.ALL, fetch = FetchType.Lazy)
private Set<Enrollment> enrollments;
```

- Lazy: transfers object when needed
  - Needs a transactional environment

```
@Component
public class ProfessorService {

    @Autowired
    CourseRepository courseRepository;

    @Autowired
    ProfessorRepository professors;

    @Transactional
    public void addCourses(String name, String... courses) {
        Professor p = professors.findByName(name);

        Set<Course> cs = p.getCourses();
        for(String c: courses)
            cs.add(courseRepository.findByName(c).get(0));
        professors.save(p);
    }
```

Many To Many

```java
@Entity
public class Course {

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

    private String name;

    private int credits;

    @OneToMany(mappedBy = "course",
               cascade = CascadeType.ALL,
               fetch = FetchType.EAGER)
    private Set<Enrollment> enrollments;

    @ManyToMany(mappedBy = "courses")
    private Set<Professor> professors;
}
```

```java
@Entity
public class Professor {

    public Professor(String name) {
        this.setName(name);
    }

    public Professor() {}

    @javax.persistence.Id
     @GeneratedValue(strategy=GenerationType.AUTO)
    private long Id;

    private String name;

    @ManyToMany
    private Set<Course> courses;
}
```

creates *professor_courses* table in a bidirectional relation

https://en.wikibooks.org/wiki/Java_Persistence/ManyToMany

http://www.objectdb.com/java/jpa/getting/started

https://hellokoding.com/jpa-many-to-many-extra-columns-relationship-mapping-example-with-spring-boot-maven-and-mysql/

Internet Applications Design and Implementation 2020 - 2021

(Lecture 4 - Part 6 -Optimization of data fetching)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)

(with previous participations of Jácome Cunha (jacome@fct.unl.pt) and João Leitão (jc.leitao@fct.unl.pt))

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

N+1 query problem

```
@Entity
@Table(name = "book_category")
public class BookCategory {
    private int id;
    private String name;
    private Set<Book> books;

    public BookCategory(){ … }

    public BookCategory(String name) {…}

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { … }

    public void setId(int id) { … }

    public String getName() {…}

    public void setNam

    @OneToMany(mappedBy
               cascade
    public Set<Book> getBooks() { … }

    public void setBoo
}
```

```
@Entity
public class Book{
    private int id;
    private String name;
    private BookCategory bookCategory;

    public Book() {}

    public Book(String name) { this.name = name;}

    public Book(String name, BookCategory bookCategory) { … }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { … }

    public void setId(int id) { … }

    public String getName() { … }

    @JoinColumn(name = "book_category_id")
    public BookCategory getBookCategory() { … }
```

```
@Repository
public interface BookCategoryRepository extends JpaRepository<BookCategory, Integer>{}
```

```
for( BookCategory c: categoryRepo.findAll() )
  for( Book b: c.books )
    …
```

```
select * from book_category;
select * from book where bookCategory = ?
…
select * from book where bookCategory = ?
```

N+1 query problem

```
@Entity
@Table(name = "book_category")
public class BookCategory {
    private int id;
    private String name;
    private Set<Book> books;

    public BookCategory(){ … }

    public BookCategory(String name) {…}

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { … }

    public void setId(int id) { … }

    public String getName() {…}

    public void setName(String name) { }

    public Set<Book> getBooks() { … }

}
```

```
@Repository
public interface BookCategoryRepository extends JpaRepository<BookCategory, Integer>{}
```

```
for( Book b : BookRepo.findAll() )
  System.out.println( b.toString() + b.bookCategory.toString())
```

```
@Entity
public class Book{
    private int id;
    private String name;
    private BookCategory bookCategory;

    public Book() {}

    public Book(String name) { this.name = name;}

    public Book(String name, BookCategory bookCategory) { … }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { … }

    public void setId(int id) { … }

    public String getName() { … }

    @JoinColumn(name = "book_category_id")
    public BookCategory getBookCategory() { … }

    setBookCategory(BookCategory bookCategory) {…}
```

```
select * from book;
select * from bookCategory where bookCategory = ?
…
select * from bookCategory where bookCategory = ?
```

Prefetching

- Instead of using global fetching strategies, one can define how objects/ collections related to one particular entity are loaded to memory

```
fun appointmentsOfPet(id: Long): List<AppointmentDAO> {
    val pet = pets.findById(id)
                  .orElseThrow { NotFoundException("There is no Pet with Id $id") }
    return pet.appointments
}
```

Two queries

```
interface PetRepository : JpaRepository<PetDAO, Long> {
    @Query("select p from PetDAO p inner join fetch p.appointments where p.id = :id")
    fun findByIdWithAppointment(id:Long) : Optional<PetDAO>
}
```

```
fun appointmentsOfPet(id: Long): List<AppointmentDAO> {
    val pet = pets.findByIdWithAppointment(id)
                  .orElseThrow { NotFoundException("There is no Pet with Id $id") }
    return pet.appointments
}
```

Appointment's collections is fetched and loaded in one single query

Custom queries for efficient execution

- ORM relations can produce a large number of queries… which can be optimized by means of a single query being dispatched to the DB

```
@Query(" select s from Student s "+
     "   inner join fetch s.courses en "+
     "   inner join en.course c "+
     " where c.name = :name")
List<Student> searchByCourse(@Param("name") String name);
```

- Optimization may work by summarising data

```
@Query("select new ciai.model.StudentSummary(s.name,s.age) from Student s")
List<StudentSummary> summaryStudents();
```

Java Persistence Query Language (JPQL ⊆ HQL)

- Simple custom queries

```
@Query("select s from Student s where s.name like CONCAT(?,'%')")
List<Student> search(String name);
```

- Complex custom queries

```
@Query(" select s from Student s "+
       "   inner join s.courses en "+
       "   inner join en.course c "+
       " where c.name = :name")
List<Student> searchByCourse(@Param("name") String name);
```

http://www.objectdb.com/java/jpa/getting/started