Internet Applications Design and Implementation

2015 - 2016 - 1st edition

(5 - Data Sources And The Web)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

JDBC

JDBC

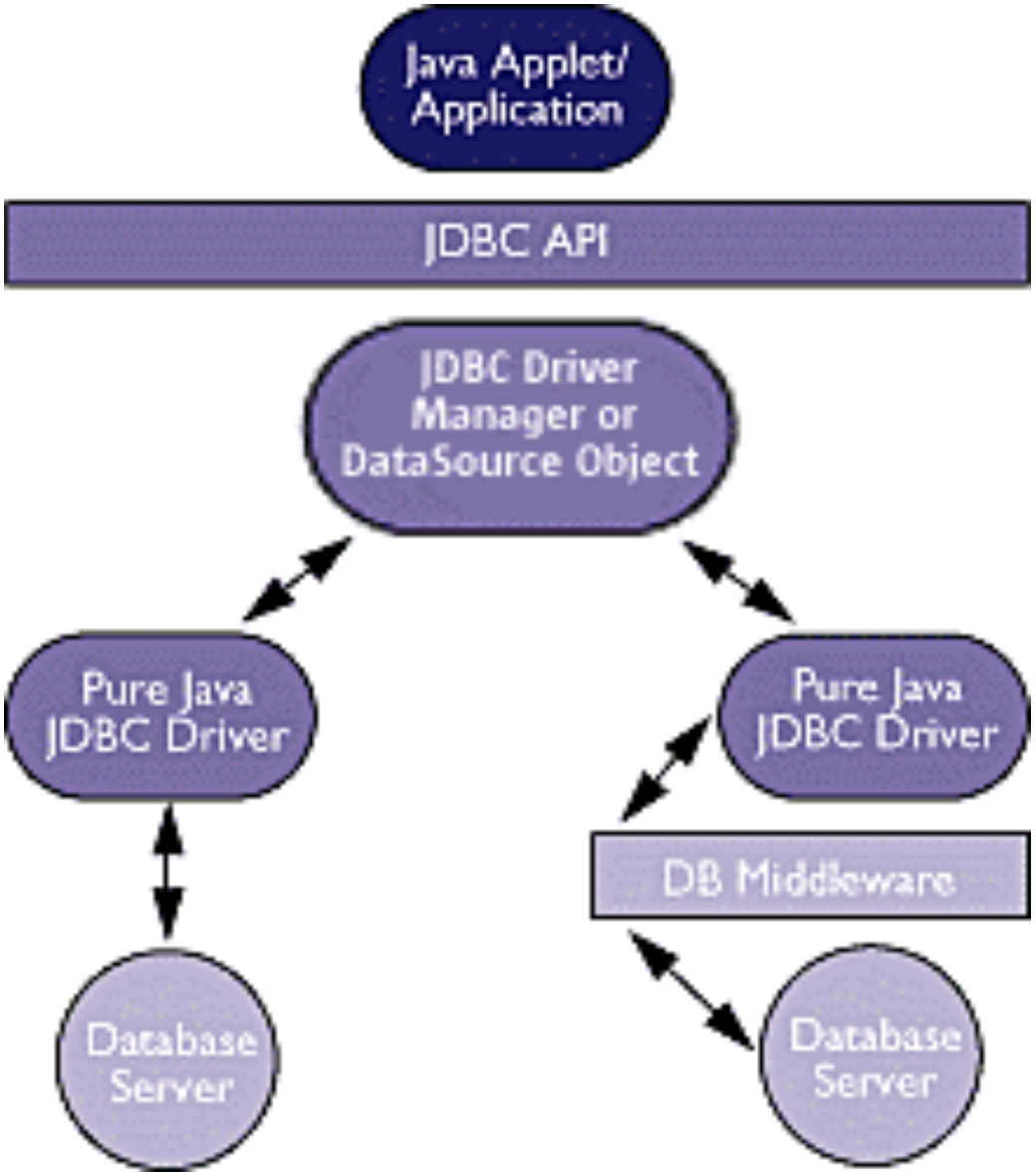
- Java Database Connectivity (JDBC)
- API for database-independent connectivity between Java code and data tabular sources
  - Includes relational databases, spreadsheets, or CSV files
- *Write Once, Run Anywhere*
- http://www.oracle.com/technetwork/java/overview-141217.html

JDBC API Overview

- The JDBC API makes it possible to do 3 things:
  1. Establish a connection with a database or access any tabular **data source**
  2. Send **SQL statements**
  3. Process the **results**

JDBC Architecture

- API for application writers
- (Low -level) API for driver writers



JDBC Technology Core features (the java.sql package)

Feature		Benefit
Result set enhancements		Ease of programming
- Scrollable result set		Ability to move a result set's cursor to a specific row. This feature is used by GUI tools and for programmatic updating
- Updatable result set		Ability to use Java programming language commands rather than SQL
New data types support		Performance improvement (ability to manipulate large objects such as BLOB and CLOB without bringing them to the client from the DB server)
Batch updates		Performance improvement (sending multiple updates to the DB for processing as a batch can be much more efficient than sending update statements separately)
Savepoints		Ability to roll transactions back to where a savepoint is set

JDBC Optional Package features (the `javax.sql` package)

Feature		Benefit
JNDI support		Ease of deployment (gives JDBC driver independence, makes JDBC applications easier to manage)
Connection pooling		Performance improvement (a connection pool is a cache of database connections that is maintained in memory, so that the connections may be reused)
Distributed transactions		Important for implementing a distributed transaction processing system
JavaBeans (`RowSet` objects)		Send data across a network to thin clients, such as web browsers, laptops, PDAs, and so on Access any tabular data source, even spreadsheets or flat files Make results sets scrollable or updatable when the JDBC driver does not support scrollability and updatability
Reference to JDBC Rowset		Encapsulate a driver as a JavaBeans component for use in a GUI
Statement pooling		Performance improvement (by pooling statements as well as pooling connections)

JDBC & Java Examples

```
public void connectToAndQueryDatabase(String username,
                                       String password)

    Connection con = DriverManager.getConnection(
                        "jdbc:myDriver:myDatabase",
                        username,
                        password);

    Statement stmt = con.createStatement();

    ResultSet rs =
            stmt.executeQuery("SELECT a, b, c FROM Table1");

    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
}
```

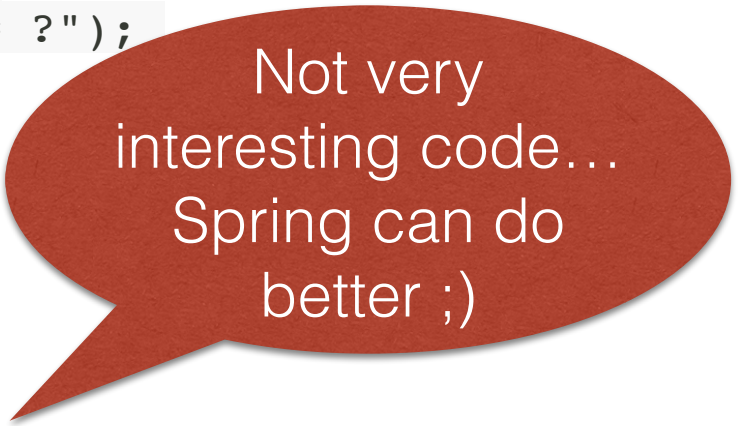
Data source

SQL statement

Results

JDBC & Java Examples

```
public static void UpdateCarNum(int carNo, int empNo) throws SQLException
{

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = DriverManager.getConnection(
                        "jdbc:default:connection");

            pstmt = con.prepareStatement(
                        "UPDATE EMPLOYEES " +
                        "SET CAR_NUMBER = ? " +
                        "WHERE EMPLOYEE_NUMBER = ?");

            pstmt.setInt(1, carNo);
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
        }
        finally {
            if (pstmt != null) pstmt.close();
        }
    }
```



JDBC & Java Examples

```
public void createTable() throws SQLException {
    String createString =
        "create table " + dbName +
        "(SUP_ID integer NOT NULL, " +
        "SUP_NAME varchar(40) NOT NULL, " +
        "STREET varchar(40) NOT NULL, " +
        "CITY varchar(20) NOT NULL, " +
        "STATE char(2) NOT NULL, " +
        "ZIP char(5), " +
        "PRIMARY KEY (SUP_ID))";

    Statement stmt = null;

    try {
        stmt = con.createStatement();
        stmt.executeUpdate(createString);
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) stmt.close();
    }
}
```

JDBC & Java Examples

```
public void populateTable() throws SQLException {

    Statement stmt = null;
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(
            "insert into " + dbName +
            "values(49, 'Superior Coffee', " +
            "'1 Party Place', " +
            "'Mendocino', 'CA', '95460')");

        stmt.executeUpdate(
            "insert into " + dbName +
            "values(101, 'Acme, Inc.', " +
            "'99 Market Street', " +
            "'Groundsville', 'CA', ‘95199')");

      …
```

Different Data Sources - Same Code

```
Connection con1 = DriverManager.getConnection(
                          "jdbc:mysql://localhost:3306/",
                          username,
                          password);

Statement stmt = con1.createStatement();

ResultSet rs =
      stmt.executeQuery("SELECT a, b, c FROM Table1");
```

```
Connection con2 = DriverManager.getConnection(
                       "jdbc:xls:file:test.xlsx");

Statement stmt = con2.createStatement();

ResultSet rs =
      stmt.executeQuery("SELECT a, b, c FROM Table1");
```

Statement Interface

boolean execute(String sql)
Executes the given SQL statement, which may return multiple results.

int[] executeBatch()
Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts.

ResultSet executeQuery(String sql)
Executes the given SQL statement, which returns a single ResultSet object.

int executeUpdate(String sql)
Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement.

ResultSet getResultSet()
Retrieves the current result as a ResultSet object.

`https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html`

ResultSet Interface

boolean first()
Moves the cursor to the first row in this ResultSet object.

boolean getBoolean(String columnLabel)
Retrieves the value of the designated column in the current row of this ResultSet object as a boolean in the Java programming language.

<T> T getObject(int columnIndex, Class<T> type)
Retrieves the value of the designated column in the current row of this ResultSet object and will convert from the SQL type of the column to the requested Java data type, if the conversion is supported.

int getRow()
Retrieves the current row number.

boolean next()
Moves the cursor forward one row from its current position.

boolean previous()
Moves the cursor to the previous row in this ResultSet object.

void updateBoolean(String columnLabel, boolean x)
Updates the designated column with a boolean value.

`https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html`

ORM

Object-Relational Mapping (ORM)

- In many situations there is a need for persistent storage
- In many cases within an object-oriented system
- And using a relational database
- Thus, an *object-relational mapping* is necessary

Object-Relational Impedance Mismatch

- Identity
  - Object: a == b and a.equals(b)
  - Relational: primary key based
- Inheritance
  - Object: natural relation
  - Relational: does not exist
- Accessing data
  - Object: through the object interface
  - Relational: select queries (with joins)
- Associations
  - 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 Persistance API (JPA)

JPA provides a POJO persistence model for ORM

```
@Entity
public class Customer {

    private int id;
    private String name;
    private Collection<Order> orders;

   // no-args constructor necessary
    public Costumer () {}

   // primary key required
    @Id  // property access is used
    public int getId() {
        return id;
    }

   // gets and sets required
    public void setId(int id) {
        this.id = id;
    }
    …
```

```
    …
    public String getName() {
        return name;
    }

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

// also OneToOne, ManyToOne, and ManyToMany
    @OneToMany(cascade=ALL,
               mappedBy="customer")
    public Collection<Order>
                            getOrders() {
        return orders;
    }

    public void setOrders(
           Collection<Order> newValue)
           {
        this.orders = newValue;
    }
}
```

Java Persistance API (JPA)

```
@Entity
@Table(name="ORDER_TABLE")
public class Order {

    private int id;
    private String address;
    private Customer customer;

    @Id
    @Column(name="ORDER_ID")
    public int getId() {
        return id;
    }

    public void setId(int id){
        this.id = id;
    }
  …
```

```
  …

    @Column(name="SHIPPING_ADDRESS")
    public String getAddress() {
        return address;
    }
    public void setAddress(
                    String address) {
        this.address = address;
    }

    @ManyToOne()
  // foreign key: column used for join
    @JoinColumn(name="CUSTOMER_ID")
    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(
                    Customer customer) {
        this.customer = customer;
    }
}
```

Hibernate

Hibernate

- An ORM framework

- JPA provider
  - Implements the JPA specification
  - Also has its own API

OpenJPA

JDO™
Java™ Data Objects

CakePHP

eclipselink

Microsoft® .NET
Entity Framework

Spring &
Data (Bases/Sources)
Abstractions

JDBC & Spring

- Spring Boot provides support for JDBC (dah)
- As in any JDBC setting, it is necessary to define a DataSource (DB, CSV file, etc.)
- Spring Boot offers the JdbcTemplate class to assist programmers

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
```

```java
@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 POM 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

public interface CrudRepository<T,ID extends Serializable>

http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

An Example

```
@Entity

@NamedQuery(name = "User.findByTheUsersName",
            query = "from User u where u.username = ?1”)

public class User extends AbstractPersistable<Long> {

   private static final long serialVersionUID = -2952735933715107252L;

   @Column(unique = true)
   private String username;

   private String firstname;

   private String lastname;

  // required by the JPA specification
  // protected as this is its only purpose
   protected User() {}
…
```

CRUD Repository

```
public interface SimpleUserRepository extends CrudRepository<User, Long> {
   /**
    * Find the user with the given username. This method will be
translated into a query using the {@link javax.persistence.NamedQuery}
annotation at the {@link User} class.
    *
    * @param lastname
    * @return
    */
   User findByTheUsersName(String username);

   /**
    * Find all users with the given lastname. This method will be
translated into a query by constructing it directly from the method name
as there is no other query declared.
    *
    * @param lastname
    * @return
    */
   List<User> findByLastname(String lastname);
```

CRUD Repository - continued

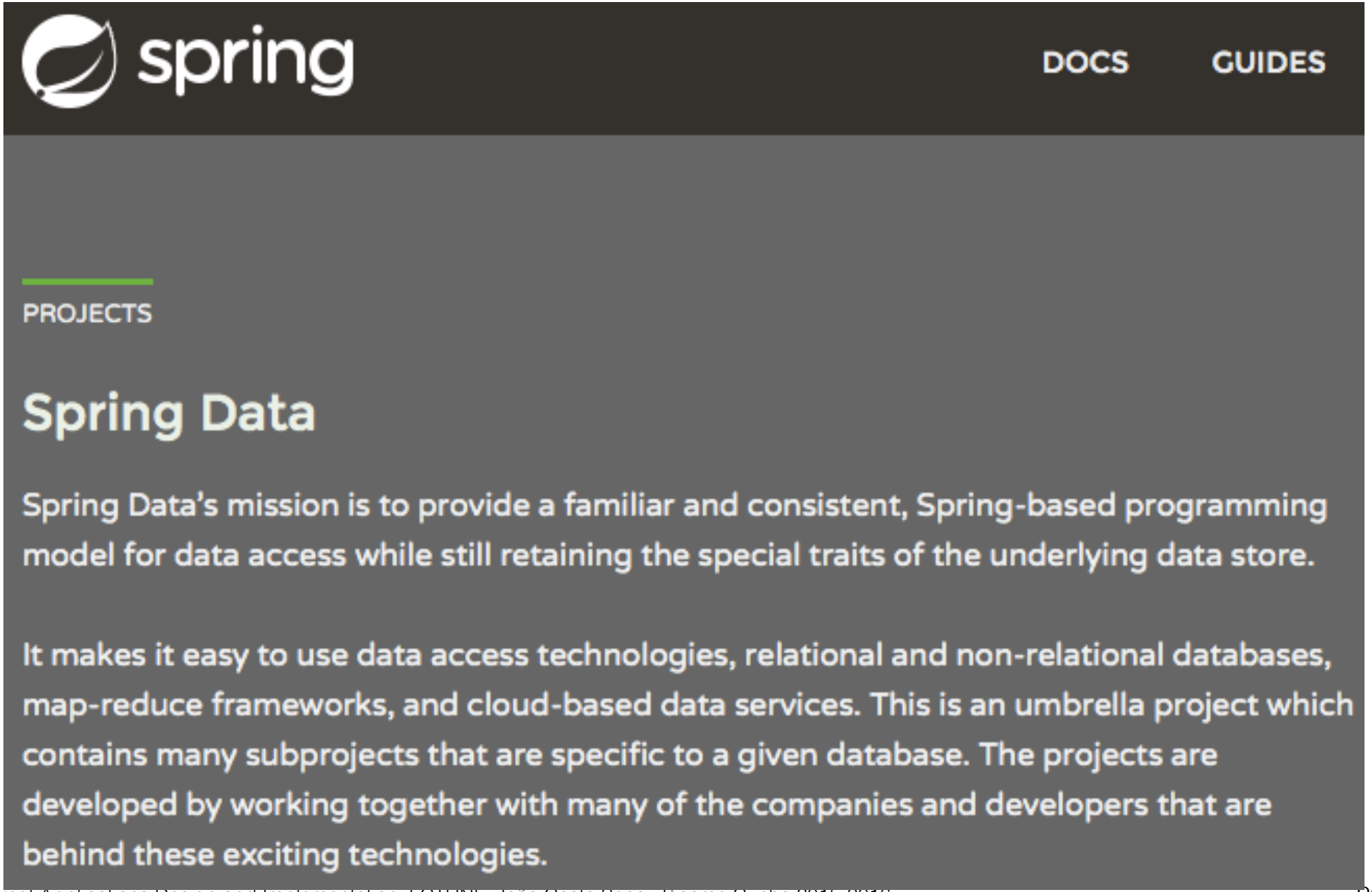
```
    …
    /**
     * Returns all users with the given firstname. This method will be translated
into a query using the one declared in the {@link Query} annotation declared one.
     *
     * @param firstname
     * @return
     */
    @Query("select u from User u where u.firstname = ?")
    List<User> findByFirstname(String firstname);

    /**
     * Returns all users with the given name as first- or lastname. Makes use of the
{@link Param} annotation to use named parameters in queries. This makes the query to
method relation much more refactoring safe as the order of the method parameters is
completely irrelevant.
     * @param name
     * @return
     */
    @Query("select u from User u where u.firstname = :name or u.lastname = :name")
    List<User> findByFirstnameOrLastname(@Param("name") String name);
}
```

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or
Is,Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = 1?
Between	findByStartDateBetween	… where x.startDate between 1? and ?
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
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1
Containing	findByFirstnameContaining	… where x.firstname like ?1
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by
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) =

More?

http://docs.spring.io/spring-data/jpa/docs/1.9.0.RELEASE/reference/html/

Even more??


spring
DOCS
GUIDES
PROJECTS
Spring Data
Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.
It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies.

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 - Main Modules

- 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.

Spring Data - 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");
  }
}
```