ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

UNIVERSIDAD D CÓRDOBA

Programación web

Prácticas

Semana 3: Acceso a base de datos con JDBC (I)

Aurora Ramírez Quesada (aramirez@uco.es)

Departamento de Ciencia de la Computación e Inteligencia Artificial

Universidad de Córdoba

Índice de contenido

Acceso a base de datos con JDBC
Introducción

- **JDBC** (*Java Database Conectivity*) es la API Java para acceso a bases de datos SQL
- Las clases necesarias para manejar la conexión (a bajo nivel) están definidas en el paquete `java.sql`

Clase Java	Descripción
DriverManager	Carga el driver de la base de datos
Connection	Establece la conexión con la base de datos
Statement	Ejecuta sentencias SQL (principalmente para consultas)
PreparedStatement	Ejecuta sentencias SQL en base de datos, especialmente cuando la sentencia requiere más de una ejecución (p.ej. Transacciones o invocaciones reiteradas) o se ejecuta con parámetros
ResultSet	Guarda el resultado de una consulta

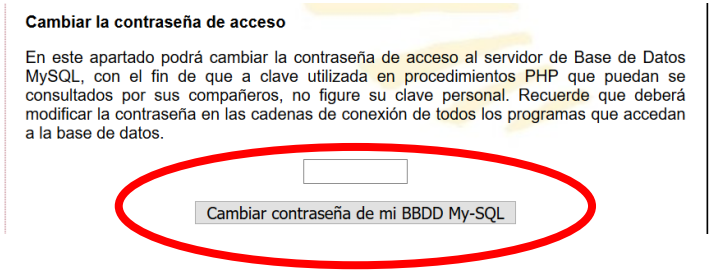
https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

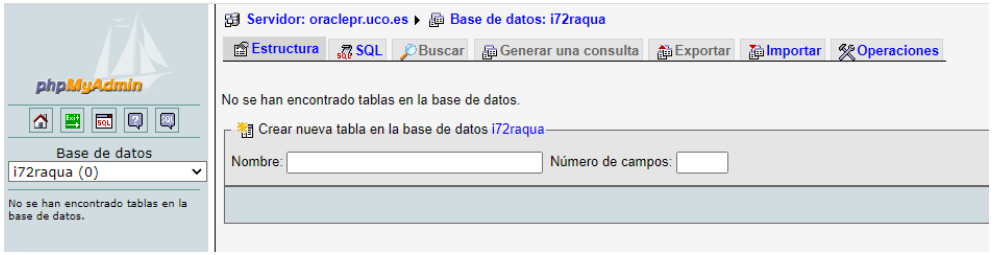
Acceso a base de datos con JDBC
Configuración de la base de datos UCO

- Se utilizará el servidor MySQL de la UCO para la persistencia de los datos
- Todos los estudiantes deben estar registrados en: http://oraclepr.uco.es/abd/
  - Si no apareces en el listado, contacta con la profesora de prácticas
  - Aunque se utilizará una cuenta por grupo, es conveniente que todos tengan acceso para ejecutar los ejemplos o hacer pruebas
  - Accede con tus credenciales UCO para crear la base de datos, que llevará el *login* como nombre

Acceso a base de datos con JDBC
Configuración de la base de datos UCO

- Cambia la clave de la base de datos para poder compartirla con el resto del grupo
- Para acceder desde fuera de la UCO será necesario tener activa la **VPN**
- El portal *phpMyAdmin* se utiliza para la gestión de la base de datos





Acceso a base de datos con JDBC

Configuración de la base de datos UCO

Esta configuración es válida para la versión **MySQL 5** instalada en el servidor UCO

- Desde *phpMyAdmin* se puede cargar un script para la creación del esquema de la base de datos y rellenarla con tuplas:
  - El nombre de las tablas y atributos deben ser claros y descriptivos
  - Incluye tuplas de ejemplo – creíbles y profesionales
- Para el ejemplo: *resources/scripts/create_bd.sql*
- El fichero *pom.xml* debe actualizarse para incluir la dependencia al driver JDBC
- Los datos de conexión deberán indicarse en el fichero *application.properties*:

Usuario:	<Usuario UCO>
Contraseña:	<Clave creada>
Servidor BBDD:	http://oraclepr.uco.es/
Puerto:	3306
Nombre BBDD:	<Usuario UCO>

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

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.49</version>
</dependency>
```

application.properties

src > main > resources > application.properties

```
1  spring.application.name=demo
2  spring.datasource.driver-class-name=com.mysql.jdbc.Driver
3  spring.datasource.url=jdbc:mysql://oraclepr.uco.es:3306/i72raqua
4  spring.datasource.username=i72raqua
5  spring.datasource.password=pw2021
```

Acceso a base de datos con JDBC
JDBC en Spring

- Spring proporciona diferentes grados de soporte para el acceso con JDBC
  - Utilizando las clases del paquete `java.sql`: solución de bajo nivel, requiere conocimiento de las clases base y gestión de la configuración a nivel de código
  - Utilizando la clase `JdbcTemplate`: Abstrae aspectos de bajo nivel, manteniendo la flexibilidad necesaria para hacer consultas de cualquier tipo
  - Utilizando el componente **Spring Data JDBC**: Automatiza las consultas a tablas mediante anotaciones, requiriendo el uso de interfaces específicas y una metodología de nombrado concreta
- Para el desarrollo de las prácticas, se hará uso de la solución intermedia:
  - Aprendizaje del flujo de comunicación para operaciones **CRUD** (***C**reate-**R**ead-**U**pdate-**D**elete*)
  - Permite la combinación con el patrón "*Repository*" para encapsular el acceso a datos
  - Gestión más sencilla de las consultas y de los resultados que utilizando las clases Java "puras"
  - Evitamos más dependencias y tener que ceñirnos a una nomenclatura concreta (libertad de diseño)

Acceso a base de datos con JDBC
Clase JdbcTemplate

- Soporta cualquier tipo de operación JDBC, aliviando errores comunes
- Ejecuta tanto consultas (*queries*) como operaciones de actualización (*updates*)
- Proporciona un acceso sencillo al conjunto de resultados, iterando sobre `ResultSet`
- Captura las excepciones de tipo `SQLException`, tratándolas como excepciones Spring
- Para poder utilizarla, hay que añadir la dependencia en *`pom.xml`*: `spring-boot-starter-jdbc`
- En la clase, se importa desde el paquete: `org.springframework.jdbc.core.JdbcTemplate`
- Para que funcione correctamente, los repositorios deberán recibirla en el constructor (inyección de dependencias)
- Spring gestionará automáticamente su creación, en base a la configuración de la base de datos indicada en *`application.properties`*

Especificación API:
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html

Acceso a base de datos con JDBC
Clase JdbcTemplate

- Para realizar una **consulta**: `List<T> query(String sql, RowMapper<T> rowMapperObject)`

1. Ejecuta la consulta recibida en el parámetro `sql`
2. Invoca a un objeto "extractor" (a implementar) para mapear el objeto genérico `ResultSet` a un objeto Java (clase de dominio)
3. Invoca a `mapRow` tantas veces como filas haya devuelto `ResultSet`
4. Devuelve una lista de objeto "`T`" (`Student` en este caso) donde cada elemento corresponde a una tupla de la tabla

```java
public List<Student> findAllStudents(){
    try{
        String query = "select id, name, surname, birthdate, type from Student;";
        List<Student> result = jdbcTemplate.query(query, new RowMapper<Student>(){
            public Student mapRow(ResultSet rs, int rowNumber) throws SQLException{
                return new Student(
                    rs.getInt(columnLabel:"id"),
                    rs.getString(columnLabel:"name"),
                    rs.getString(columnLabel:"surname"),
                    Date.valueOf(rs.getString(columnLabel:"birthdate")).toLocalDate(),
                    StudentType.valueOf(rs.getString(columnLabel:"type")));
            };
        });
        return result;
    }catch(DataAccessException exception){
        System.err.println(x:"Unable to find students");
        exception.printStackTrace();
        return null;
    }
}
```

Acceso a base de datos con JDBC
Clase JdbcTemplate

- **Consulta parametrizada**: `T query(String sql, ResultExtractor extractorMethod, Object… args)`
  1. Ejecuta la consulta escrita en el parámetro `sql`
  2. Los parámetros de la consulta se indican en secuencia (`args`)
  3. Ejecuta el método de extracción para recuperar la tupla accediendo al objeto `ResultSet`

```java
public Student findStudentById(int id){
    try{
        String query = "SELECT id, name, surname, birthDate, type FROM Student WHERE id=?;";
        Student result = jdbcTemplate.query(query, this::mapRowToStudent, id);
        if (result != null)
            return result;
        else
            return null;
    }catch(DataAccessException exception){
        System.err.println("Unable to find student with id=" + id);
        exception.printStackTrace();
        return null;
    }
}
```

```java
private Student mapRowToStudent(ResultSet row){
    try{
        if(row.first()){
            int id = row.getInt(columnLabel:"id");
            String name = row.getString(columnLabel:"name");
            String surname = row.getString(columnLabel:"surname");
            Date birthDateInTable = Date.valueOf(row.getString(columnLabel:"birthdate"));
            StudentType type = StudentType.valueOf(row.getString(columnLabel:"type"));

            Student student = new Student(id, name, surname, birthDateInTable.toLocalDate(), type);
            return student;
        }
        else{
            return null;
        }

    } catch (SQLException exception) {
        System.err.println(x:"Unable to retrieve results from the database");
        exception.printStackTrace();
        return null;
    }
}
```

> **!** En el ejemplo se asume que solo se devuelve una tupla, puesto que la consulta es sobre la clave primaria (id)

Acceso a base de datos con JDBC
Clase JdbcTemplate

- **Consulta parametrizada**: `T query(String sql, ResultExtractor extractorMethod, Object… args)`
  - Cuando la consulta parametrizada devuelve varias tuplas:
    1. Si se van a devolver todas las tuplas sin mayor comprobación, se puede usar el método extractor como en la consulta sin parametrizar
    2. Si se quiere analizar el resultado, se tendrá que iterar sobre el objeto `ResultSet` con el método `next()`

1

```java
public List<Student> findStudentsByType(StudentType type){
    try{
        String query = "SELECT id, name, surname, birthDate FROM Student WHERE type=?;";
        You, 1 second ago | 1 author (You)
        List<Student> result = jdbcTemplate.query(query, new RowMapper<Student>(){
        public Student mapRow(ResultSet rs, int rowNumber) throws SQLException{
            return new Student(
                rs.getInt(columnLabel:"id"),
                rs.getString(columnLabel:"name"),
                rs.getString(columnLabel:"surname"),
                Date.valueOf(rs.getString(columnLabel:"birthdate")).toLocalDate(),
                type);
        };
        }, type.toString());
        return result;

    }catch(DataAccessException exception){
        System.err.println("Unable to find student with type=" + type.toString());
        exception.printStackTrace();
        return null;
    }
}
```

2

```java
public int getNumberStudentsByType(StudentType type){
    String query = "SELECT id, name, surname, birthDate FROM Student WHERE type=?;";
    try{
        int result = jdbcTemplate.query(query, this::countRowsStudentType, type.toString());
        return result;
    }catch(DataAccessException exception){
        System.err.println("Unable to find student with type=" + type.toString());
        exception.printStackTrace();
        return -1;
    }
}

private int countRowsStudentType(ResultSet row){
    try{
        int numberOfStudents = 0;
        while(row.next()){
            numberOfStudents++;
        }
        return numberOfStudents;
    } catch (SQLException exception) {
        System.err.println(x:"Unable to retrieve results from the database");
        exception.printStackTrace();
        return -1;
    }
}
```

Acceso a base de datos con JDBC
Clase JdbcTemplate

- Para realizar una **actualización**: `int update(String sql, Object… args)`
  1. Ejecuta la actualización escrita en el parámetro `sql`
  2. Los parámetros de la consulta se indican en secuencia (`args`)
  3. Devuelve el número de tuplas afectadas por la actualización

```
public boolean addStudent(Student student){
    try{
        String query = "INSERT INTO Student (id, name, surname, birthDate, type) VALUES (?, ?, ?, ?, ?);";
        int result = jdbcTemplate.update(query,
        student.getId(),
        student.getName(),
        student.getSurname(),
        student.getBirthDate().toString(),
        student.getType().toString());
        if (result>0)
            return true;
        else
            return false;
        } catch(DataAccessException exception){
            System.err.println(x:"Unable to insert student in the database");
            exception.printStackTrace();
        return false;
    }
}
```

Podemos preferir devolver un valor booleano para que el controlador sepa si la operación se ha realizado con éxito (si el cambio se ha reflejado en la base de datos o no)

Acceso a base de datos con JDBC
Clase JdbcTemplate

- Control de excepciones:
  - Los errores de acceso a base de datos son bastante frecuentes: consultas mal escritas, errores en el número y orden de parámetros, fallos en la conexión
  - Es conveniente incluir tratamiento de excepciones (bloques `try/catch`) para capturar errores, que no se propaguen hacia el controlador, y no interrumpir la ejecución abruptamente

- Decisión de diseño a seguir:
  - Los métodos reciben la consulta en formato `String`, dejando expuesta la estructura de la base de datos y dificultando la mantenibilidad
  - Haremos uso de un **fichero de propiedades** y la clase Java `Properties` para especificar las consultas como un recurso externo y poder reutilizarlas

Especificación API: https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Properties.html

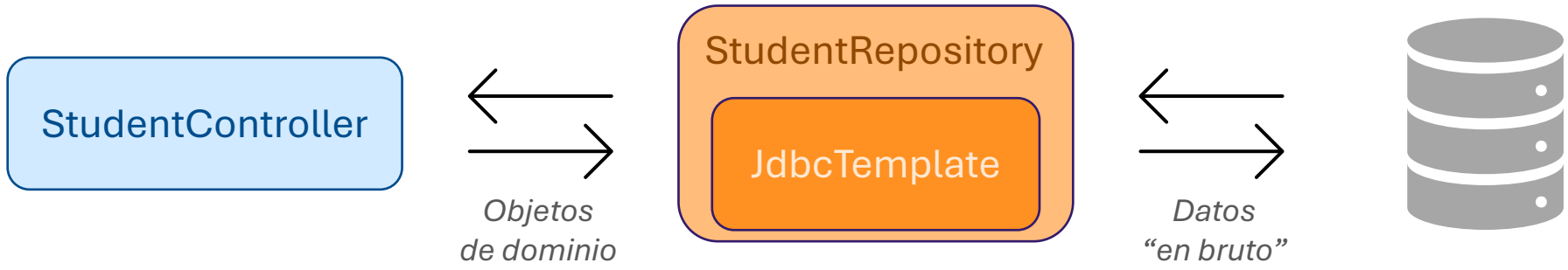
Acceso a base de datos con JDBC
Concepto de Repositorio

- Patrón de diseño software que encapsula y oculta la lógica de acceso a datos y los detalles sobre el almacenamiento de esos datos
  - Actúa de intermediario entre el componente que solicita datos (en nuestro caso, el controlador) y el sistema de almacenamiento de datos (en nuestro caso, la base de datos)
  - Transforma los datos "en bruto" a objetos de dominio, permitiendo que cambios futuros en la tecnología de almacenamiento no impacten en el funcionamiento de la aplicación
  - Concepto similar a los *Data Access Object* (DAO), con sutiles diferencias:
    - Los DAO tradicionalmente se asocian 1:1 a cada entidad de la base de datos, y necesitan conocer más detalles de su estructura
    - Los repositorios se especializan en un conjunto de conceptos de dominio relacionados
    - Los repositorios tienen un diseño orientado a interfaces, muchos *frameworks* ofrecen interfaces para repositorios donde se predefinen métodos comunes: `findAll()`, `findById()`, `save()`

https://java-design-patterns.com/patterns/repository/#programmatic-example-of-repository-pattern-in-java

Acceso a base de datos con JDBC
Concepto de Repositorio

- Patrón de diseño software que encapsula y oculta la lógica de acceso a datos y los detalles sobre el almacenamiento de esos datos
  - El repositorio debe recibir en el constructor un objeto `JdbcTemplate`
  - Se utiliza la anotación **@Repository** para su configuración automática
  - Puesto que no vamos a utilizar Spring Data JDBC, no es necesario extender la interfaz `Repository`: Esta interfaz parametrizable, junto con el uso de otras anotaciones, permite automatizar la creación de consultas simples (**no lo usaremos**)
  - Los repositorios definirán los métodos de acceso a datos para que sean invocados desde los controladores



Acceso a base de datos con JDBC
Ejemplo

- Implementación completa del repositorio `StudentRepository`:
  1. Declaración e inicialización de la clase: Anotación y constructor parametrizado (`JdbcTemplate`)
  2. Creación del objeto `Properties` para acceder al fichero con las consultas SQL

StudentRepository.java

src > main > java > es > uco > pw > demo > model > StudentRepository.java > StudentRepository

```
18
19  @Repository
20  public class StudentRepository {
21
22      private JdbcTemplate jdbcTemplate;
23      private Properties sqlQueries;
24      private String sqlQueriesFileName;
25
26      public StudentRepository(JdbcTemplate jdbcTemplate){
27          this.jdbcTemplate = jdbcTemplate;
28      }
29
30      public void setSQLQueriesFileName(String sqlQueriesFileName){
31          this.sqlQueriesFileName = sqlQueriesFileName;
32          createProperties();
33      }
```

```
119     private void createProperties(){
120         sqlQueries = new Properties();
121         try {
122             BufferedReader reader;
123             File f = new File(sqlQueriesFileName);
124             reader = new BufferedReader(new FileReader(f));
125             sqlQueries.load(reader);
126         } catch (IOException e) {
127             System.err.println(x:"Error creating properties object for SQL queries");
128             e.printStackTrace();
129         }
130     }
131 }
```

Acceso a base de datos con JDBC
Ejemplo

- Implementación completa del repositorio `StudentRepository`:
  3. Acceso a fichero de propiedades para obtener la consulta SQL en cada método
  4. Incorporación de tratamiento de excepciones (consulta no encontrada, resultado no válido, etc.)

```java
public List<Student> findAllStudents(){
    try{
        String query = sqlQueries.getProperty(key:"select-findAllStudents");
        if(query != null){
            List<Student> result = jdbcTemplate.query(query, new RowMapper<Student>(){
            public Student mapRow(ResultSet rs, int rowNumber) throws SQLException{
                return new Student(
                    rs.getInt(columnLabel:"id"),
                    rs.getString(columnLabel:"name"),
                    rs.getString(columnLabel:"surname"),
                    Date.valueOf(rs.getString(columnLabel:"birthdate")).toLocalDate(),
                    StudentType.valueOf(rs.getString(columnLabel:"type")));
                };
            });
            return result;
        }
        else
            return null;
    }catch(DataAccessException exception){
        System.err.println(x:"Unable to find students");
        exception.printStackTrace();
        return null;
    }
}
```

```java
public boolean addStudent(Student student){
    try{
        String query = sqlQueries.getProperty(key:"insert-addStudent");
        if(query != null){
            int result = jdbcTemplate.update(query,
            student.getId(),
            student.getName(),
            student.getSurname(),
            student.getBirthDate().toString(),
            student.getType().toString());
            if (result>0)
                return true;
            else
                return false;
        }
        else
            return false;
        } catch(DataAccessException exception){
            System.err.println(x:"Unable to insert student in the database");
            exception.printStackTrace();
        return false;
    }
}
```

! Durante el desarrollo, es conveniente imprimir por consola la traza de error con `exception.printStackTrace()`

Acceso a base de datos con JDBC
Ejemplo

- Probando el acceso a base de datos desde el controlador `HomeController`:

1. Asociar el repositorio al controlador en el constructor
2. Configurar el repositorio con el fichero de consultas SQL

```java
@Controller
public class HomeController {

    StudentRepository studentRepository;

    public HomeController(StudentRepository studentRepository){
        this.studentRepository = studentRepository;
        String sqlQueriesFileName = "./src/main/resources/db/sql.properties";
        this.studentRepository.setSQLQueriesFileName(sqlQueriesFileName);
    }
```

! Como **no** tenemos definido el **flujo MVC completo**, aquí simplemente invocamos a los métodos y mostramos los resultados por consola al acceder a la página home

```java
@GetMapping("/")
public String home() {

    // Testing student repository and database access

    // Find all students
    List<Student> listOfStudents = studentRepository.findAllStudents();
    if(listOfStudents!=null)
        System.err.println("[HomeController] Number of students in the database: " + listOfStudents.size());

    // Add student
    LocalDate date = LocalDate.of(year:2002, month:7, dayOfMonth:27);
    Student student = new Student(id:7, name:"Gloria", surname:"Garcia", date, StudentType.PARTIAL_TIME);
    boolean success = studentRepository.addStudent(student);
    System.out.println("[HomeController] Student added to the database: " + success);

    // Find student by ID
    int id = 7;
    Student insertedStudent = studentRepository.findStudentById(id);
    if (insertedStudent!=null){
        System.out.println("[HomeController] Student with id=" + id + " found:");
        System.out.println(
            "\tName: " + insertedStudent.getName() +
            "\n\tSurname: " + insertedStudent.getSurname() +
            "\n\tBirth date: " + insertedStudent.getBirthDate().toString() +
            "\n\tType: " + insertedStudent.getType().toString());
    }

    // Find students by type
    List<Student> erasmusStudents = studentRepository.findStudentsByType(StudentType.ERASMUS);
    if(erasmusStudents!=null){
        System.out.println(x:"List of Erasmus students:");
        erasmusStudents.forEach((s) -> System.out.println("\tName: " + s.getName() + " Surname: " + s.getSurname()));
    }

    // Count students by type
    int numberOfPartialStudents = studentRepository.getNumberStudentsByType(StudentType.PARTIAL_TIME);
    System.out.println("Number of partial-time students: " + numberOfPartialStudents);

    return new String(original:"home");
}
```

Objetivos de la semana

1. Crea tu base de datos personal en el servidor UCO
2. Replica el ejemplo explicado en clase, cambiando la configuración para usar tu base de datos
   - Versión completa disponible en el repositorio Github: https://github.com/aramirez-uco/pw-examples
3. Implementa el script de creación de la base de datos para el proyecto de prácticas, incluyendo datos realistas en las tablas
4. Implementa los repositorios para acceso a datos: solo para tablas individuales (sin considerar relaciones) y siguiendo las recomendaciones de diseño
5. Consulta la bibliografía recomendada:
   - Capítulo 3: "*Working with Data*" del libro "*Spring in Action*" (6th ed.)
   - MySQL:
     - https://dev.mysql.com/doc/refman/5.7/en/
     - https://www.w3schools.com/mysql/
   - Spring MVC: https://spring.io/guides/gs/relational-data-access