**JDBC Template**

* Lucrul cu JDBC API e cam anevoios, caci necesita crearea de Connection, Statement, ResultSet etc. pentru un simplu query
* Spring ofera o implementare mai buna pentru JDBC, numita JDBC Template
* Ea tot foloseste Statement. ResultSet, doar ca intern
* Pentru a crea un DataSource pentru H2 data base si un JdbcTemplate

@Bean  
public DataSource dataSource(){  
 return new EmbeddedDatabaseBuilder()  
 .addScript("script.sql")  
 .setType(EmbeddedDatabaseType.*H2*)  
 .build();  
}  
@Bean  
public JdbcTemplate jdbcTemplate(){  
 return new JdbcTemplate(dataSource());  
}

**Metode JdbcTemplate**

**Update() e pentru a insera, sterge date**

* int update(String query) - folosit pentru a insera, actualiza si sterge date in BD, adica pentru DML

jdbcTemplate.update("INSERT INTO Teacher VALUES(1,'Mititiuc','Eduard')");

* int update(String query, Object arg1, Object Arg2 .... Object ArgN) - ca update, dar foloseste PreparedStatement folosind argumentele oferite de noi

jdbcTemplate.update("INSERT INTO Teacher VALUES(?, ?, ?)",1,"Mititiuc","Eduard");

* void execute(String query) - este pentru DDL query, asa cum nu ia nici un argument si nu returneaza nimic

**query() este pentru a selecta lista de date**

* T query(String sql, ResultSetExtractor)
* List<T> query(query, RowMapper)
* @GetMapping("/")  
  public List<Teacher> teacher(){  
   jdbcTemplate.execute("INSERT INTO Teacher VALUES(1,'Mititiuc','Eduard')");  
    
   return jdbcTemplate.query("SELECT \* FROM Teacher WHERE id=?",new TeacherRowMapper());  
  }

**queryForObject sunt deja pentru a returna date**

* T queryForObject(String sql, RowMapper Object) - are nevoie de un object de tip RowMapper, sau care extinde. De ex, ca sa obtinem un intreg Teacher, trebuie de creat un TeacherRowMapper care sa implementeze RowMapper, asa:

@Override  
public Teacher mapRow(ResultSet rs, int rowNum) throws SQLException {  
 Teacher teacher = new Teacher();  
 teacher.setId(rs.getInt("id"));  
 teacher.setFirstName(rs.getString("first\_name"));  
 teacher.setLastName(rs.getString("last\_name"));  
  
 return teacher;  
}

@GetMapping("/")  
public Teacher index(){  
 jdbcTemplate.update("INSERT INTO Teacher VALUES(?, ?, ?)",1,"Mititiuc","Eduard");  
 return jdbcTemplate.queryForObject("SELECT \* FROM Teacher WHERE id = 1", new TeacherRowMapper());  
}

* T queryForObject(String sql, RowMapper Object, Object args1,.., Object args2,...,Object argsN)

**Atentie!RowMapper object, nu cu .class!!**

* T queryForObject(String sql, Class<T> returnType) - executa query si returneaza un obiect de tip T,. **Suporta doar o coloana sa fie selectata!**
* T queryForObject(String sql, Class<T> returnType, Object arg1, ..., Object argN) - creaza un sql query si leaga argumentele de query si returneaza un obiect de tip T. **Tot returneaza doar o coloana!**
* T queryForObject(String sql, Object[] args, int[] argsType, Class<T> returnType) - creaza un sql query si leaga argumentele de tipul specificat de query si returneaza un obiect de tip T

**NamedParameterJdbcTemplate**

* JdbcTemplate are un neajuns, si anume ca trebuie sa inlocuim ? cu argumentele noastre
* NamedParameterJdbcTemplate rezolva problema data, si ne permite sa folosim named parametrs asa :nume

Dependenta ca la JdbcTemplate

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.3.16</version>

</dependency>

* Vom folosi deja un HashMap ce va avea ca key numele de dupa : si argumentul nostru sau un MapSqlParameterSource
* Metode(tot cam aceleasi, doar ca mai iau si un Map sau un MapSqlParameterSource):

- update(query, MapSqlParameterSource)

- update(query, Map)

- queryForObject(sql,Map,RowMapper Object)

- queryForObject(sql,Map,Object.class) - pentru a returna o valoare simpla

@GetMapping("/")  
public Teacher index(){  
 MapSqlParameterSource mapSqlParameterSource1 = new MapSqlParameterSource();  
 mapSqlParameterSource1.addValue("id",2);  
 mapSqlParameterSource1.addValue("first\_name","Goncear");  
 mapSqlParameterSource1.addValue("last\_name","Alex");  
  
 MapSqlParameterSource mapSqlParameterSource2 = new MapSqlParameterSource();  
 mapSqlParameterSource2.addValue("id",2);  
  
  
 jdbcTemplate.update("INSERT INTO Teacher VALUES(:id,:first\_name,:last\_name)",mapSqlParameterSource1);  
  
 return jdbcTemplate.queryForObject("SELECT \* FROM Teacher WHERE id=:id",mapSqlParameterSource2,new TeacherRowMapper());  
  
}

Sau

@GetMapping("/")  
public Teacher index(){  
 MapSqlParameterSource mapSqlParameterSource1 = new MapSqlParameterSource();  
 mapSqlParameterSource1.addValue("id",2);  
 mapSqlParameterSource1.addValue("first\_name","Goncear");  
 mapSqlParameterSource1.addValue("last\_name","Alex");  
  
 Map<String, Integer> map = new HashMap<>();  
 map.put("id",2);  
  
  
 jdbcTemplate.update("INSERT INTO Teacher VALUES(:id,:first\_name,:last\_name)",mapSqlParameterSource1);  
  
 return jdbcTemplate.queryForObject("SELECT \* FROM Teacher WHERE id=:id",map,new TeacherRowMapper());  
  
}

GetAll: List<T> query(string, RowMapper Obj)

@GetMapping("/")  
public List<Teacher> index(){  
 MapSqlParameterSource mapSqlParameterSource1 = new MapSqlParameterSource();  
 mapSqlParameterSource1.addValue("id",2);  
 mapSqlParameterSource1.addValue("first\_name","Goncear");  
 mapSqlParameterSource1.addValue("last\_name","Alex");  
  
 Map<String, Integer> map = new HashMap<>();  
 map.put("id",2);  
  
  
 jdbcTemplate.update("INSERT INTO Teacher VALUES(:id,:first\_name,:last\_name)",mapSqlParameterSource1);  
  
 return jdbcTemplate.query("SELECT \* FROM Teacher",new TeacherRowMapper());  
  
}

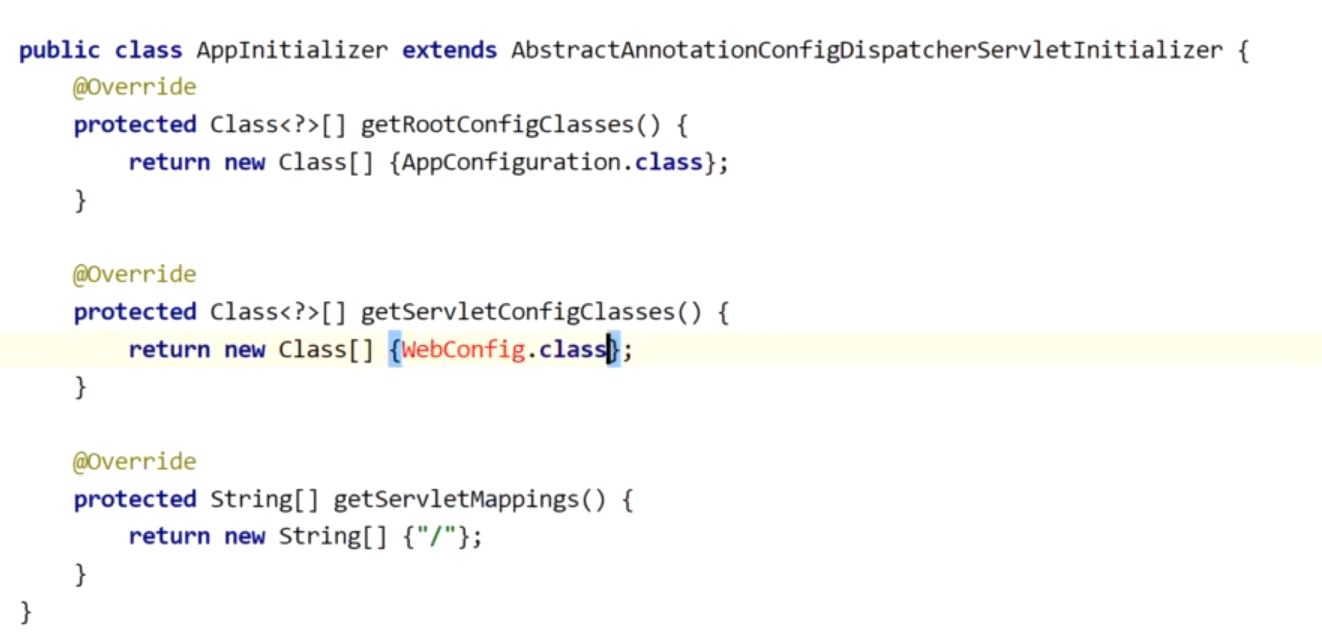
* Daca nu avem nevoie de vreun Map, scriem null;

**Cum aflam key pus de baza de date**

Pentru asta folosim un KeyHolder ca argument

KeyHolder keyHolder = new GeneratedKeyHolder();  
jdbcTemplate.update("INSERT INTO Teacher VALUES(:id,:first\_name,:last\_name)",mapSqlParameterSource1,keyHolder);  
System.*out*.println(keyHolder.getKey());

**DispatcherServlet**



la getRootConfigClasses punem @Configuration unde se gasesc beanurile care reprezinta bussiness logicul, sau alte beanuri

la getServletConfigClasses punem acel @Configuration unde se gasesc anotatiile si beanurile pentru server, ca @EnableWebMvc etc. sau orice tine de web

**XML Format in loc de JSON**

@GetMapping(value = "/",produces = MediaType.*APPLICATION\_XML\_VALUE*)

<dependency>  
 <groupId>com.fasterxml.jackson.dataformat</groupId>  
 <artifactId>jackson-dataformat-xml</artifactId>  
</dependency>

* E bine ca la @ExceptionHandler sa punem ca argument si exceptia ce o poate capta:
* @ExceptionHandler(ArrayIndexOutOfBoundsException.class)

**AOP**

* Cand cream un @Aspect, la rularea contextului, Spring gaseste toate clasele @Aspect din ele si scaneaza toate anotatiile. Apoi scaneaza toate clasele ce sunt @Component sau alte tipuri, si pentru ele creaza proxy-uri
* In proxy creat, se va modifica comportamentul metodelor,
* Poincut trebuie sa fie cat mai particulare, de altfel se vor crea prea multe proxyuri la runtime si aplicatia va frana sau se va deschide foarte greu
* Metodele apelate in alte metode nu vor fi interceptate de proxy.

De ex:

void test(){

func();

}

func() { }

La apelarea lui test(), aspectele nu vor merge asupra lui func(), din cauza ca test() nu s-a executat din proxy object