Introducción a Spring JDBC y Unit Testing

- Continuando desde donde quedamos en la clase anterior, procedemos a implementar la capa de persistencia usando JDBC contra una base de datos PostgreSQL. Comenzamos, definiendo las dependencias sobre el pom padre:

```
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-jdbc</artifactId>
          <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
          <groupId>org.postgresql</groupId>
          <artifactId>postgresql</artifactId>
          <version>${org.postgresql.version}</version>
        </dependency>
```

la versión correspondiente:

```
    <org.postgresql.version>9.3-1102-jdbc41</org.postgres
ql.version>
```

y en el pom de persistence:

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

- Volvemos a correr `mvn eclipse:eclipse` y refrescamos el proyecto.
- Modificamos la implementación de `UserJdbcDao` para agregar un constructor que pueda en base a un `DataSource` (interfaz definida por JDBC) crear un `JdbcTemplate` (helper ofrecido por Spring para manejar JDBC) e implementar `findById` contra la misma.

```
@Repository
public class UserJdbcDao implements UserDao {
    private JdbcTemplate jdbcTemplate;

    private final static RowMapper<User> ROW_MAPPER = new
 RowMapper<User>() {

        @Override
        public User mapRow(ResultSet rs, int rowNum) thro
ws SQLException {
            return new User(rs.getString("username"), rs.
```

```
getInt("userid"));
        }
    };

    @Autowired
    public UserJdbcDao(final DataSource ds) {
        jdbcTemplate = new JdbcTemplate(ds);
    }

    @Override
    public User findById(final long id) {
        final List<User> list = jdbcTemplate.query("SELEC
T * FROM users WHERE userid = ?",
                ROW_MAPPER, id);
        if (list.isEmpty()) {
            return null;
        }
        return list.get(0);
    }
}
```

- El haber puesto `@Autowired` sobre un constructor le indica a Spring qué constructor usar cuando quiera crear instancias de este DAO. Obviamente, para poder llamar a este constructor, necesita disponier de una instancia apropiada de cada parámetro. Spring no conoce un `DataSource`, debemos por tanto configurar uno. Como en casos anteriores, vamos a hacer

esto con un `@Bean` en `WebConfig`

```
    @Bean
    public DataSource dataSource() {
        final SimpleDriverDataSource ds = new SimpleDrive
rDataSource();
        ds.setDriverClass(org.postgresql.Driver.class);
        ds.setUrl("jdbc:postgresql://localhost/paw");
        ds.setUsername("root");
        ds.setPassword("root");

        return ds;
    }
```

Donde `/paw` al final de la url indica que la base de datos se llama `paw` y el username y la password son aquellos establecidos en la creación de la base de datos PostgreSQL.

- La base de datos es creada desde línea de comandos usando `createdb paw -O root`, donde `paw` es el nombre de la base a crearse y `root` es el nombre de usuario owner de la misma. Por defecto en PostgreSQL los users y passwords corresponden a los del sistema operativo. Esta configuración se puede cambiar editando el archivo que en Ubuntu se encuentra en `/etc/postgresql/9.3/main/pg_hba.conf` y reiniciando el servicio.

- Levantamos Jetty y vemos como esto funciona. Insertamos a mano un par de registros en la base de datos usando el cliente de CLI (`psql -W paw root`).

- Intentamos ahora crear `User`s en la base de datos por código. En el `HelloWorldController` modificamos el método de recuperación y escribimos nuestro nuevo endpoint para creación:

```
    @RequestMapping("/")
    public ModelAndView index(@RequestParam(value = "user
Id", required = true) final int id) {
        final ModelAndView mav = new ModelAndView("index"
);
        mav.addObject("user", us.findById(id));

        return mav;
    }

    @RequestMapping("/create")
    public ModelAndView create(@RequestParam(value = "nam
e", required = true) final String username) {
        final User u = us.create(username);
        return new ModelAndView("redirect:/?userId=" + u.
getId());
    }
```

- Definimos el método create en `UserService`

```
    /**
     * Create a new user.
     *
     * @param username The name of the user.
     * @return The created user.
     */
    User create(String username);
```

- Hacemos la implementación concreta en `UserServiceImpl`

```
    @Override
    public User create(final String username) {
        return userDao.create(username);
    }
```

- Definimos el método `create` en el `UserDao`

```
    /**
     * Create a new user.
     *
     * @param username The name of the user.
     * @return The created user.
     */
    User create(String username);
```

- Y finalmente, implementamos el método `create` en

`UserJdbcDao`. Podríamos hacer esto con `jdbcTemplate` llamando a `execute` para el `INSERT` y luego recuperando el id autogenerado. Sin embargo, Spring nos ofrece un helper, `SimpleJdbcInsert`, que evita tener que lidiar con el armado de estas queries. Para esto agregamos el field en `UserJdbcDao`:

```
    private final SimpleJdbcInsert jdbcInsert;
```

Lo inicializamos en el constructor:

```
    jdbcInsert = new SimpleJdbcInsert(jdbcTemplate)
                .withTableName("users")
                .usingGeneratedKeyColumns("userid");
```

Y finalmente, escribimos el método `create`:

```
    @Override
    public User create(final String username) {
        final Map<String, Object> args = new HashMap<>();
        args.put("username", username); // la key es el n
ombre de la columna

        final Number userId = jdbcInsert.executeAndReturn
Key(args);

        return new User(username, userId.longValue());
```

```
    }
```

- Nos resta ahora asegurarnos de que al levantar la aplicación, se creen las tablas en forma automática. Para esto, modificamos nuevamente el constructor de `UserJdbcDao` para agregar:

```
    jdbcTemplate.execute("CREATE TABLE IF NOT EXISTS user
s ("
                        + "userid SERIAL PRIMARY KEY,"
                        + "username varchar(100)"
                        + ")");
```

- Habiendo hecho esto, procedemos a escribir un test unitario para nuestro DAO, que hasta acá carecía de estos. Como primer paso, vamos a necesitar una nueva dependencia de spring para tener ciertas utilidades; y vamos a querer usar HSQLDB para tener una base de datos en memoria, sobre la qu epodamos asegurar que correr los tests produzcan siempre lo mismos resultados.

Comenzamos por definir en el pom padre la dependencia sobre spring-test y HSQLDB:

```
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${org.springframework.version}</version>
```

```
        <scope>test</scope>
      </dependency>
      <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>${org.hsqldb.version}</version>
        <scope>test</scope>
      </dependency>
```

Donde definimos la versión de HSQLDB como un property:

```
    <org.hsqldb.version>2.3.1</org.hsqldb.version>
```

Y en el pom de persistence la declaramos:

```
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
    </dependency>
```

Corremos `mvn eclipse:eclipse` y refrescamos el proyecto en eclipse. Creamos en el módulo de persistencia, bajo `src/test/java` y

dentro del mismo paquete que `UserJdbcDao` nuestro `UserJdbcDaoTest`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = TestConfig.class)
public class UserJdbcDaoTest {

    private static final String PASSWORD = "Password";
    private static final String USERNAME = "Username";

    @Autowired
    private DataSource ds;

    @Autowired
    private UserJdbcDao userDao;

    private JdbcTemplate jdbcTemplate;

    @Before
    public void setUp() {
        jdbcTemplate = new JdbcTemplate(ds);
        JdbcTestUtils.deleteFromTables(jdbcTemplate, "use
rs");
    }

    @Test
    public void testCreate() {
        final User user = userDao.create(USERNAME, PASSWO
```

```
RD);

        assertNotNull(user);
        assertEquals(USERNAME, user.getUsername());
        assertEquals(PASSWORD, user.getPassword());

        assertEquals(1, JdbcTestUtils.countRowsInTable(jd
bcTemplate, "users"));
    }
}
```

`JdbcTestUtils` es una utiity class provista por spring que nos permite simplificar estos tests sin tener que escribir queries a mano en las tablas siendo modificadas; y sin tener que confiar en otros métodos de nuestro DAO para validar el escenario siendo testeado.

Donde el uso del runner de spring en lugar del default de JUnit va a permitirnos levantar un contexto de spring e inyectar variables con el uso de `@Autowired`. La clase `TestConfig` que se indica tiene configuración de contexto es una clase `@Configuration` que vamos a crear exclusivamente para testing. Por tanto, vamos a ponerla dentro del mismo paquete que este último test, con la única config requerida para testear nuestro dao, es decir, un `DataSource`:

```
@ComponentScan({ "ar.edu.itba.paw.persistence", })
@Configuration
public class TestConfig {
```

```
    @Bean
    public DataSource dataSource() {
        final SimpleDriverDataSource ds = new SimpleDrive
rDataSource();
        ds.setDriverClass(JDBCDriver.class);
        ds.setUrl("jdbc:hsqldb:mem:paw");
        ds.setUsername("ha");
        ds.setPassword("");

        return ds;
    }
}
```

Corremos el tests y encontramos nuestro primer problema. La sintaxis SQL del create table, aunque válida en PostgreSQL, no lo es en HSQLDB, donde en vez del tipo de datos `serial` se usa `identity` para indicar claves autogeneradas.

Decidimos entonces quitar del código la creación de las tablas. Esto además nos permite despreocuparnos de en qué orden se crearán los DAOs (y por tanto, se ejecutarían las queries) si estuviesen ahí dentro.

Removemos de `UserJdbcDao` la llamada a `execute` que crea la tabla, y creamos en el módulo persistence, bajo `src/test/resources` un archivo `schema.sql` con la query como contenido:

```
CREATE TABLE IF NOT EXISTS users (
    userid INTEGER IDENTITY PRIMARY KEY,
    username varchar(100)
);
```

Análogamente, bajo `src/main/resources/` creamos un `schema.sql` con la versioń equivalente para PostgreSQL

```
CREATE TABLE IF NOT EXISTS users (
    userid SERIAL PRIMARY KEY,
    username varchar(100)
);
```

Para nuestro test, es tan sencillo como anotar `UserJdbcDaoTest` con `@Sql("classpath:schema.sql")`. De esta forma, dicho archivo SQL será ejecutado antes de correr el test. El prefijo `classpath:` le indica a spring que debe buscar dicho recurso dentro de los source directories o dependencias. Dado que el archivo está en `src/test/resources` y bajo Maven este directorio es un source directory, lo encontrará sin necesidad de que nosotros estemos escribiendo paths absolutos o relativos (que puede o no funcionar en función de desde donde se ejecuten los tests y donde se dejan los `.class` generados).

Para nuestra aplicación web, es ligeramente más complicado. Vamos a modificar el `WebConfig` para indicarle como prepopular nuestra base

de datos. Para esto vamos a agregar un par de beans:

```
    @Value("classpath:schema.sql")
    private Resource schemaSql;
```

```
    @Bean
    public DataSourceInitializer dataSourceInitializer(fi
nal DataSource ds) {
        final DataSourceInitializer dsi = new DataSourceI
nitializer();
        dsi.setDataSource(ds);
        dsi.setDatabasePopulator(databasePopulator());
        return dsi;
    }

    private DatabasePopulator databasePopulator() {
        final ResourceDatabasePopulator dbp = new Resourc
eDatabasePopulator();
        dbp.addScript(schemaSql);
        return dbp;
    }
```

Corremos la aplicación y vemos que ahora sí, todo funciona como esperamos, y los tests lo mismo.