



# Spring

## Sesión 2: Acceso a datos



# Puntos a tratar

- Por qué usar Spring para acceso a datos
- Problemas típicos de JDBC
- JDBC con Spring
- JPA con Spring
- Transaccionalidad declarativa



# Por qué usar el acceso a datos de Spring

- Spring no nos obliga a usar su módulo DAO, es decisión nuestra aprovecharlo o usar el API que queramos directamente
- **Simplifica el código** de los DAOs en APIs tediosos como JDBC
  - Gracias a los templates
  - Permite usar también directamente el API si lo preferimos
- Ofrece una **rica jerarquía de excepciones** independiente del API de acceso a datos



# Código típico JDBC

```
public UsuarioTO login(String login, String password) throws DAOException {
    Connection con=null;
    try {
        con = ds.getConnection();
        PreparedStatement ps = con.prepareStatement(SQL);
        ps.setString(1, login);
        ps.setString(2, password);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            UsuarioTO uto = new UsuarioTO();
            uto.setLogin(rs.getString("login"));
            uto.setPassword(rs.getString("password"));
            uto.setFechaNac(rs.getDate("fechaNac"));
            return uto;
        }
        else
            return null;
    } catch (SQLException sqle) {
        throw new DAOException(sqle);
    }
}
```

Código “útil”  
Infraestructura



# Código típico JDBC (II)

```
finally {  
    if (con!=null) {  
        try {  
            con.close();  
        }  
        catch(SQLException sqle2) {  
            throw new DAOException(sqle2);  
        }  
    }  
}
```

**Código “útil”  
Infraestructura**



# Solución 1: Templates

- El código anterior es demasiado largo. Si siempre hay que abrir una conexión al principio y cerrarla al final ¿por qué lo tenemos que hacer explícitamente?
  - **Template:** ya tiene implementadas las partes que se hacen siempre. Solo tenemos que implementar lo que cambia
  - Hay *templates* para JDBC, JPA, Hibernate, myBatis,... Nuestro código no va a ser independiente del API, pero sí más corto que si usamos el API directamente



## Problema 2: Excepciones JDBC

- En JDBC, hay muy pocas. **SQLException** se usa para todo
  - APIs como Hibernate definen muchas más, pero son exclusivas del API
- Si son comprobadas, hay que poner **try/catch** o **throws**
  - Si fuerzas a la gente a poner catch, puedes conseguir que acabe poniendo catch vacíos (lo sé, vosotros no lo haríais nunca, yo tengo que confesar que sí)
  - Además, normalmente un DAO poco puede hacer por gestionar/arreglar una excepción (bueno, pues que la lance para arriba)
  - Pero si casi siempre la tiene que lanzar ¿no es también un poco tedioso poner siempre **throws**?



## Solución 2: Excepciones DAO en Spring

- De modo transparente, capturar la excepción del API que se está usando y transformarla en una propia de Spring, independiente del API
  - Hay muchas: **DataAccessResourceFailureException** (no hay conexión con la BD), **DataIntegrityViolationException** (clave duplicada),...
  - Al poner **@Repository**, se activa esta traslación automática
- Hacer todas las excepciones de acceso a datos no comprobadas
  - Todas las anteriores heredan de **DataAccessException**, que hereda de **RuntimeException**





# JDBC en Spring con JdbcTemplate

- En Spring hay muchos más templates para JDBC, pero este aprovecha Java 5, simplificando mucho el código
- Práctica habitual
  - Nuestro DAO guarda una referencia al template
  - Cuando nos inyecten el DataSource instanciamos el template

```
<jee:jndi-lookup id="ds" jndi-name="jdbc/MiDataSource"
                resource-ref="true"/>
```

*(en el XML con los beans)*

```
@Repository("JDBC")
public class UsuariosDAOJDBC {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource ds) {
        this.jdbcTemplate = new JdbcTemplate(ds);
    }
}
```



# SELECT con JdbcTemplate

- Normalmente cada registro del ResultSet va a acabar siendo un TO. El encargado de hacer esto en Spring es el **RowMapper**, cuyo interfaz debemos implementar

```
public class UsuarioTOMapper implements RowMapper<UsuarioTO> {  
  
    public UsuarioTO mapRow(ResultSet rs, int numRows) throws SQLException {  
        UsuarioTO uto = new UsuarioTO();  
        uto.setLogin(rs.getString("login"));  
        uto.setPassword(rs.getString("password"));  
        uto.setFechaNac(rs.getDate("fechaNac"));  
        return uto;  
    }  
}
```



## SELECT con JdbcTemplate (II)

- Para hacer el SELECT
  - Si esperamos un solo objeto, queryForObject()

```
private static final String LOGIN_SQL = "select * from usuarios " +
    "where login=? and password=?";

public UsuarioTO login(String login, String password) {
    //El row mapper de antes
    UsuarioTOMapper miMapper = new UsuarioTOMapper();
    try {
        return this.jdbcTemplate.queryForObject(LOGIN_SQL, miMapper,
            login, password);
    }
    //¡Que conste que la capturo porque quiero! (es no comprobada)
    catch(EmptyResultDataAccessException erdae) {
        return null;
    }
}
```



# SELECT con JdbcTemplate (III)

- Para hacer el SELECT
  - Si esperamos varios objetos, **query()**

```
private static final String LIST_SQL = "select * from usuarios " +
    "where localidad=?";

public List<UsuarioTO> listarPorLocalidad(String localidad) {
    //El row mapper de antes
    UsuarioTOMapper miMapper = new UsuarioTOMapper();
    try {
        return this.jdbcTemplate.query(LIST_SQL, miMapper, localidad);
    }
    //;Que conste que la capturo porque quiero! (es no comprobada)
    catch(EmptyResultDataAccessException erdae) {
        return null;
    }
}
```



# UPDATE con JdbcTemplate

```
private static final String REGISTRAR_SQL = "insert into usuarios" +  
      (login, password, fechaNac) values (?, ?, ?);  
  
public void registrar(UsuarioTO uto) {  
    this.jdbcTemplate.update(REGISTRAR_SQL, uto.getLogin(),  
        uto.getPassword(), uto.getFechaNac());  
}
```



# JPA: opciones de configuración

- JPA gestionado por la aplicación
- JPA gestionado por el contenedor
  - Directamente, para servidores de aplicaciones, que tendrán soporte “nativo”
  - También es posible para servidores web, como Tomcat, el soporte lo proporciona Spring
- En cualquiera de los casos, podemos usar anotaciones para inyectar el EM. Spring se ocupa de los detalles



# Necesitamos un bean que “fabrique” EMs

```
<beans xmlns="http://www.springframework.org/schema/beans"
..."> (faltan “cabeceras”, ver apuntes para código completo)
  <context:component-scan base-package="es.ua.jtech"/>
  <jee:jndi-lookup id="miDS" jndi-name="jdbc/MiDataSource" resource-ref="true"/>
  <bean id="miEMF"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="miDS"/>
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="showSql" value="true"/>
        <property name="generateDdl" value="true"/>
        <property name="databasePlatform" value="org.hibernate.dialect.HSQLDialect"/>
      </bean>
    </property>
  </bean>
</beans>
```



# JPA: opciones para el código

- Hay dos opciones
  - Usar **JPA\_Template**: al ser JPA un API bastante conciso, no ganaremos mucho en código
  - Usar el API JPA directamente. Esta segunda opción es la que recomienda la propia documentación de Spring. No atamos nuestro código al API de Spring y este será portable directamente a cualquier servidor de aplicaciones





# Código JPA “estándar”

- Lo único que tiene de especial es que funciona en Tomcat y similares

```
@Repository("JPA")
public class UsuariosDAOJPA implements IUsuariosDAO {
    @PersistenceContext
    EntityManager em;

    public UsuarioTO login(String login, String password) {
        UsuarioTO uto = em.find(UsuarioTO.class, login);
        if (uto!=null && password.equals(uto.getPassword()))
            return uto;
        else
            return null;
    }
}
```



# Transaccionalidad declarativa

- Normalmente se gestiona desde la capa de negocio, aunque está íntimamente ligada al acceso a datos
- Lo primero que necesitamos en Spring es un “Transaction Manager”. Hay varias implementaciones, dependiendo del API usado por los DAOs

```
<jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring" resource-ref="true" />

<!-- Elegimos un tipo de "Transaction Manager" (aquí para JDBC) -->
<bean id="miTxManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="miDataSource"/>
</bean>

<!-- Decimos que para este Transaction Manager vamos a usar anotaciones -->
<tx:annotation-driven transaction-manager="miTxManager"/>
```



# La anotación @Transactional

- Colocada delante de un método, lo hace transaccional. Delante de la clase hace que TODOS los métodos lo sean
- El comportamiento por defecto es **rollback automático** ante excepción no comprobada (recordemos que `DataAccessException` lo es)

```
@Service
public class GestorUsuarios {
    @Autowired
    private UsuariosDAO udao;

    @Transactional
    public void registrar(UsuarioTO uto) {
        udao.registrarEnBD(uto);
        udao.registrarEnListasDeCorreo(uto);
    }
}
```



# Configurar @Transactional

- Admite una serie de atributos
  - **rollbackFor**: clases que causarán *rollback*
  - **norollbackFor**
  - **propagation**: propagación de la transacción, como en EJB
  - **timeout**: tiempo de espera
  - ...

```
@Service
public class GestorUsuarios {
    @Autowired
    private UsuariosDAO udao;

    @Transactional(rollbackFor=AltaPublicidadException.class)
    public void registrar(UsuarioTO uto) {
        udao.registrarEnBD(uto);
        udao.registrarEnListasDeCorreo(uto);
    }
}
```



# Transacciones y JDBC directo

- Si usamos directamente JDBC, estamos abriendo y cerrando conexiones en cada método de cada DAO (generalmente). ¡¡Si la conexión se cierra, adiós a la posibilidad de *rollback*!!
- La clase DataSourceUtils nos permite abrir conexiones y “liberarlas” sin cerrarlas en realidad

```
//El DataSource se habría resuelto por inyección de dependencias  
@Autowired  
private DataSource ds;
```

```
...  
Connection con = DataSourceUtils.getConnection(ds)
```

Abrir conexión

```
DataSourceUtils.releaseConnection(con, ds);
```

Liberar conexión



# ¿Preguntas...?