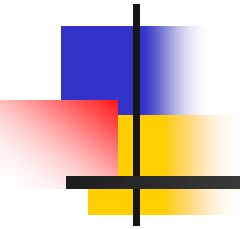
Tema 4: Pruebas de Integración de la Capa Modelo



Introducción

- Se ha utilizado JUnit 5 para implementar las pruebas de integración del servicio **`MovieService`**
- Son pruebas de integración, y no de unidad, porque prueban el correcto funcionamiento de **`MovieService`** (**`MovieServiceImpl`**), e indirectamente lo que éste utiliza internamente (los DAOs que acceden a la BD)
- Las pruebas de integración del servicio **`MovieService`** residen en **`MovieServiceTest`** y están implementadas al estilo tradicional
  - Convención de nombrado: **`XxxServiceTest`**
  - En general, para cada caso de uso se han diseñado varios casos de prueba, cada uno implementado en un método **`@Test`**
  - A veces resulta conveniente probar más de un caso de prueba en un único método **`@Test`**
- A modo de repaso de JUnit, a continuación se ilustra la implementación de tres casos de prueba para el caso de uso **`buyMovie`**
- Posteriormente se destacan aspectos específicos a la implementación de pruebas de servicios de la capa Modelo en el contexto de JUnit y JDBC

Repaso (1)

- JUnit 5 consta de 3 módulos
  - JUnit Platform: módulo core
  - **JUnit Jupiter**: nuevo modelo de programación de pruebas
  - JUnit Vintage: permite ejecutar pruebas escritas con JUnit 3 o JUnit 4 (compatibilidad hacia atrás).
  - Más información: https://junit.org/junit5/docs/current/user-guide/#overview-what-is-junit-5
- Con respecto a JUnit 4, JUnit Jupiter
  - Ofrece un modelo de programación similar, pero que aprovecha las características introducidas en Java 8
  - Usa `org.junit.jupiter.api` como paquete principal, en lugar de ~~`org.junit`~~
  - Incluye la clase `Assertions`, similar a la clase ~~`org.junit.Assert`~~, pero con métodos adicionales
  - `@Test`: no proporciona el parámetro ~~`expected`~~
    - En su lugar hay que usar `Assertions.assertThrows` dentro del caso de prueba

Repaso (2)

```
// ...
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
// ...

public class MovieServiceTest {

    private static MovieService movieService = null;
    private static SqlSaleDao saleDao = null;

    private Movie getValidMovie(String title) {
        return new Movie(title, (short) 85, "Movie description",
            19.95F));
    }

    private Movie getValidMovie() {
        return getValidMovie("Movie title");
    }
```

Repaso (3)

```
private Movie createMovie(Movie movie) {

    Movie addedMovie = null;
    try {
        addedMovie = movieService.addMovie(movie);
    } catch (InputValidationException e) {
        throw new RuntimeException(e);
    }
    return addedMovie;

}

private void removeMovie(Long movieId) {

    try {
        movieService.removeMovie(movieId);
    } catch (InstanceNotFoundException |
            MovieNotRemovableException e) {
        throw new RuntimeException(e);
    }

}
```

Repaso (4)

```
private void removeSale(Long saleId) {

    DataSource dataSource = DataSourceLocator
            .getDataSource(MOVIE_DATA_SOURCE);

    try (Connection connection = dataSource.getConnection()) {

        try {

            /* Prepare connection. */
            connection.setTransactionIsolation(
                Connection.TRANSACTION_SERIALIZABLE);
            connection.setAutoCommit(false);

            /* Do work. */
            saleDao.remove(connection, saleId);

            /* Commit. */
            connection.commit();
```

Repaso (5)

```
            } catch (InstanceNotFoundException e) {
                connection.commit();
                throw new RuntimeException(e);
            } catch (SQLException e) {
                connection.rollback();
                throw new RuntimeException(e);
            } catch (RuntimeException|Error e) {
                connection.rollback();
                throw e;
            }

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

    }
```

Repaso (6)

org.junit.jupiter.api.Test (JUnit 5) y **NO org.junit.Test** (JUnit 4)

```
@Test
public void testBuyMovieAndFindSale()
    throws InstanceNotFoundException, InputValidationException,
    SaleExpirationException {

    Movie movie = createMovie(getValidMovie());
    Sale sale = null;

    try {

        // Buy movie
        LocalDateTime beforeBuyDate =
            LocalDateTime.now().withNano(0);

        sale = movieService.buyMovie(movie.getMovieId(), USER_ID,
            VALID_CREDIT_CARD_NUMBER);

        LocalDateTime afterBuyDate =
            LocalDateTime.now().withNano(0);
```

Repaso (7)

```
// Find sale
Sale foundSale = movieService.findSale(sale.getSaleId());

// Check sale
assertEquals(sale, foundSale);
assertEquals(VALID_CREDIT_CARD_NUMBER,
    foundSale.getCreditCardNumber());
assertEquals(USER_ID, foundSale.getUserId());
assertEquals(movie.getMovieId(), foundSale.getMovieId());
assertTrue(movie.getPrice() == foundSale.getPrice());
assertTrue((foundSale.getExpirationDate().compareTo(
    beforeBuyDate.plusDays(SALE_EXPIRATION_DAYS)) >= 0) &&
    (foundSale.getExpirationDate().compareTo(
    afterBuyDate.plusDays(SALE_EXPIRATION_DAYS)) <= 0));
assertTrue((foundSale.getSaleDate().compareTo(
    beforeBuyDate) >= 0) &&
   (foundSale.getSaleDate().compareTo(afterBuyDate) <= 0));
assertTrue(foundSale.getMovieUrl().startsWith(
    BASE_URL + sale.getMovieId()));
```

Repaso (8)

```
} finally {
        // Clear database: remove sale (if created) and movie
        if (sale != null) {
            removeSale(sale.getSaleId());
        }
        removeMovie(movie.getMovieId());
    }
}
```

Repaso (9)

```
@Test
public void testBuyMovieWithInvalidCreditCard() {

    Movie movie = createMovie(getValidMovie());

    try {
        assertThrows(InputValidationException.class, () -> {
            Sale sale = movieService.buyMovie(movie.getMovieId(),
                USER_ID, INVALID_CREDIT_CARD_NUMBER);
            removeSale(sale.getSaleId());
        });

    } finally {
        // Clear database
        removeMovie(movie.getMovieId());
    }
}
```

De **`org.junit.jupiter.api.Assertions`** (JUnit 5) y **NO** de **`org.junit.Assert`** (JUnit 4)

Expresión lambda (segundo parámetro) que debería devolver la excepción especificada en el primer parámetro de **`assertThrows`**

Repaso (y 10)

```
    @Test
    public void testBuyNonExistentMovie() {

        assertThrows(InstanceNotFoundException.class, () -> {
            Sale sale = movieService.buyMovie(NON_EXISTENT_MOVIE_ID,
                USER_ID, VALID_CREDIT_CARD_NUMBER);
            removeSale(sale.getSaleId());
        });

    }

    // Casos de prueba para el resto de casos de uso...

} // class
```

Aspectos específicos

- Conceptos básicos de pruebas en Maven
- Independencia entre casos de prueba
- Uso de DAOs
- Uso de una BD específica para ejecutar las pruebas
- Igualdad de objetos
- Inicialización
- Configuración
- DataSourceLocator

Conceptos básicos de pruebas en Maven

- Con Maven el código de pruebas se ubica en el directorio **`src/test`** (que tiene una estructura de directorios similar a la del directorio **`src/main`**)
- En el módulo **`ws-movies-model`** el código de las clases de prueba está contenido en el paquete **`es.udc.ws.movies.test.model`** (y subpaquetes)
- Las pruebas se pueden ejecutar automáticamente desde Maven (a través del plugin **`surefire`**) haciendo que se ejecute la fase **`test`** (e.g. **`mvn test`**)
  - Maven genera un informe detallado en **`target/surefire-reports`**
  - Por defecto, se consideran clases de pruebas aquellas clases de **`src/test/java`** cuyo nombre empieza o termina por **`Test`**

Independencia entre casos de prueba (1)

- Para favorecer el mantenimiento del código es importante tender, en la medida de lo posible, a que cada caso de prueba
  - [1] Cree los datos que precise
  - [2] Invoque a la operación a probar
  - [3] Realice las comprobaciones necesarias
  - [4] Elimine los datos que crearon en el paso [1] y los que se pudieron generar en el paso [2]
- De esta manera, cada caso de prueba es independiente del resto de casos de prueba
  - La ejecución de un caso de prueba puede asumir que sólo existen en BD los datos que él mismo ha creado en [1]
  - Modificar, eliminar o añadir un caso de prueba no tiene incidencia en el resto de casos
  - Esta independencia favorece que distintas personas a lo largo del tiempo puedan modificar las clases de prueba de manera “ágil”

Independencia entre casos de prueba (y 2)

- Para evitar replicar código, los casos de prueba ilustrados anteriormente utilizan los métodos privados
  - **`createMovie`**, **`removeMovie`** y **`removeSale`** para encargarse de los aspectos [1] y [4]
    - Asumen que los parámetros son correctos (porque se usan para crear datos necesarios para la ejecución de los casos de prueba [1] o para eliminar datos creados tras su ejecución [4])
    - En consecuencia, capturan las excepciones checked y las relanzan como **`RuntimeException`** (lo que evita tener que tratar esas excepciones checked en los casos de prueba)
  - **`getValidMovie`** para crear cómodamente un objeto **`Movie`** con datos correctos

Uso de DAOs

- Como se comentó en la discusión del aspecto anterior, en general los casos de prueba necesitan crear los datos necesarios [1] antes de invocar al caso de uso a probar, y finalmente eliminar los datos creados [4]
- Para [1] y [4] se ha seguido el siguiente criterio
  - Si el servicio ofrece un método apropiado, se usa (e.g. **`createMovie`** y **`removeMovie`**)
  - En otro caso, se usa directamente el DAO correspondiente (e.g. **`removeSale`**)
- Cuando en un caso de prueba se realizan las comprobaciones necesarias [3], puede ser necesario hacer alguna consulta a la BD
  - Mismo convenio que el anterior

Uso de una BD específica para ejecutar las pruebas

- Es una buena práctica disponer de varias BDs con el mismo esquema, por ejemplo
  - [1] Una BD para probar la aplicación “como humano” en la fase de desarrollo
  - [2] Una BD para ejecutar las pruebas automatizadas
  - [3] Una BD para producción
- En los ejemplos de la asignatura se dispone de BDs para [1] (**`ws`**) y [2] (**`wstest`**)
- De esta manera, el probar la aplicación como “humano” no altera la BD de pruebas
  - Los casos de prueba no se encontrarán con “datos inesperados”

Igualdad de objetos (1)

- **`Assertions.assertEquals(Object, Object)`**
  - Compara utilizando el método **`equals`** (definido en **`Object`**)
- La implementación por defecto de **`equals`** en **`Object`** realiza una comparación por referencia
  - **`sale1.equals(sale2)`** es **`true`** si **`sale1`** y **`sale2`** son dos variables que apuntan al mismo objeto **`Sale`**
- Si para alguna clase se desea que la comparación sea por contenido, es necesario redefinir el método **`equals`**
  - Por ejemplo, si queremos que **`sale1.equals(sale2)`** sea **`true`** aunque **`sale1`** y **`sale2`** sean dos objetos distintos, pero con el mismo contenido (mismo id de venta, mismo id de película, mismo id de usuario, etc.), es preciso redefinir **`equals`**
  - La implementación de **`equals`** debe devolver **`true`** si los dos objetos son iguales campo a campo

Igualdad de objetos (y 2)

- Cuando se redefine **`equals`**, debe definirse **`hashCode`** de manera consistente
  - Por defecto, **`hashCode`** devuelve enteros diferentes para distintos objetos
  - Cuando se redefine **`equals`**, **`hashCode`** debe devolver el mismo entero para objetos que, aunque distintos, son iguales en contenido
  - Esto permite usar los objetos como clave en estructuras que indexan por clave (e.g. mapas)
- En **`testBuyMovieAndFindSale`**, para que sea posible comparar **`sale`** y **`foundSale`**, es preciso redefinir **`equals`** (y **`hashCode`** por consistencia), dado que son dos objetos distintos, y si la prueba es satisfactoria, con el mismo contenido
- En los ejemplos de la asignatura, se ha usado el IDE para generar ágilmente la implementación de **`equals`** y **`hashCode`** en **`Movie`** y **`Sale`**

Inicialización (1)

- A lo largo de los casos de prueba se utilizan las variables estáticas **`movieService`** y **`saleDao`**
- Estas variables se inicializan en un método anotado con **`@BeforeAll`**
  - Se ejecuta una única vez, antes de que se ejecuten los métodos anotados con **`@Test`**

```
@BeforeAll
public static void init() {

    // Algo más??? ...

    movieService = MovieServiceFactory.getService();
    saleDao = SqlSaleDaoFactory.getDao();

}
```

Inicialización (2)

- RECORDATORIO del tema 3
  - El constructor de la implementación de **`MovieService`** obtiene la referencia al **`DataSource`** de la siguiente manera

```
public MovieServiceImpl() {
    dataSource = DataSourceLocator.getDataSource(
        MOVIE_DATA_SOURCE);
    // ...
}
```

  - [1, aplicación Web /servicio Web] Cuando la capa Modelo se ejecuta dentro de un servidor de aplicaciones, **`getDataSource`** tiene que localizar el **`DataSource`** proporcionado por el servidor de aplicaciones
  - [2, pruebas capa Modelo] Cuando la capa Modelo se ejecuta desde las pruebas de integración, el desarrollador de las pruebas tiene que proporcionar un **`DataSource`** y el método **`getDataSource`** lo devolverá

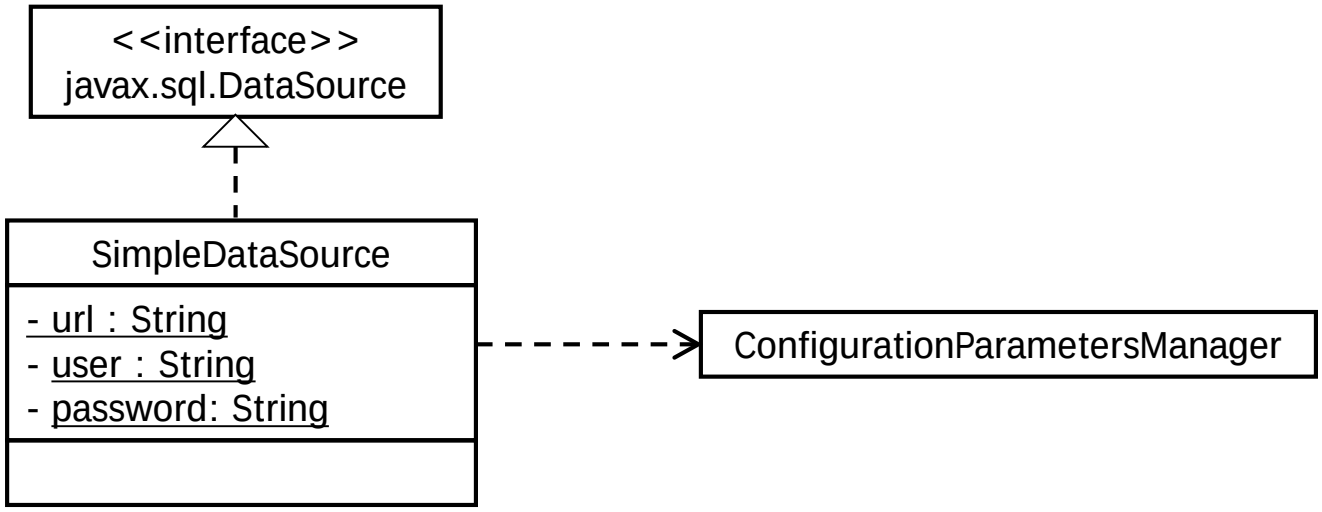
Inicialización (3)

- Para el escenario [2], **`DataSourceLocator`** proporciona el método **`addDataSource`**, que permite registrar un **`DataSource`** por nombre explícitamente

```
@BeforeAll
public static void init() {

    DataSource dataSource = new SimpleDataSource();
    DataSourceLocator.addDataSource(MOVIE_DATA_SOURCE,
        dataSource);

    movieService = MovieServiceFactory.getService();
    saleDao = SqlSaleDaoFactory.getDao();

}
```

Inicialización (4)

- **`SimpleDataSource`**
  - Es una implementación trivial de la interfaz **`DataSource`** proporcionada por el módulo **`ws-util`**
  - Implementa el método **`getConnection`** delegando en **`DriverManager.getConnection`** (para las pruebas no es necesario proporcionar pool de conexiones)
  - El resto de métodos lanzan **`SQLFeatureNotSupportedException`**
  - Delega en **`ConfigurationParametersManager`** para leer el nombre de la URL de conexión a la BD, el usuario y contraseña



Inicialización (5)

```
public class SimpleDataSource implements DataSource {

    private static final String URL_PARAMETER =
        "SimpleDataSource.url";
    private static final String USER_PARAMETER =
        "SimpleDataSource.user";
    private static final String PASSWORD_PARAMETER =
        "SimpleDataSource.password";

    private static String url;
    private static String user;
    private static String password;
```

Inicialización (6)

```
private synchronized void readConfiguration() {
    if (url == null) {
        try {

            /* Read configuration parameters. */
            url = ConfigurationParametersManager.getParameter(
                URL_PARAMETER);
            user = ConfigurationParametersManager.getParameter(
                USER_PARAMETER);
            password = ConfigurationParametersManager.getParameter(
                PASSWORD_PARAMETER);

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Inicialización (y 7)

```
    @Override
    public Connection getConnection() throws SQLException {
        readConfiguration();
        return DriverManager.getConnection(url, user, password);
    }

    << Resto de métodos =>
       throw new SQLFeatureNotSupportedException("Not implemented"); >>

}
```

Configuración (1)

- Con Maven la configuración de las pruebas reside en **`src/test/resources`**
- **`ws-movies/ws-movies-model/src/test/resources`** contiene el fichero **`ConfigurationParameters.properties`** con la siguiente configuración

```
SimpleDataSource.url=jdbc:mysql://localhost/wstest?...
SimpleDataSource.user=ws
SimpleDataSource.password=ws

MovieServiceFactory.className=es.udc.ws.movies.model.movieservice.MovieServiceImpl
SqlMovieDaoFactory.className=es.udc.ws.movies.model.movie.Jdbc3CcSqlMovieDao
SqlSaleDaoFactory.className=es.udc.ws.movies.model.sale.Jdbc3CcSqlSaleDao
```

Configuración (y 2)

- **`SimpleDataSource`** está configurado para usar la BD de pruebas (**`wstest`**)
- Existen dos ficheros **`ConfigurationParameters.properties`** en ws-**`movies-model`**
  - El “normal” [1] y el específico para ejecución de pruebas [2]
    - [1] **`src/main/resources/ConfigurationParameters.properties`**
    - [2] **`src/test/resources/ConfigurationParameters.properties`**
  - [2] añade con respecto a [1] la configuración de **`SimpleDataSource`**
  - Cuando se ejecuta las pruebas sobre un módulo Maven, el classpath está formado por (en este orden): **`target/test-classes`**, **`target/classes`** y los JARs de las dependencias
  - En consecuencia, cuando se ejecutan las pruebas, **`ConfigurationParametersManager`** encontrará primero a [2]

DataSourceLocator (1)

- Hemos visto que **`MovieServiceTest`** tiene que registrar un **`DataSource`** explícitamente antes de poder invocar al servicio de la capa Modelo (**`MovieServiceImpl`**)
  - En el método **`init`** (**`@BeforeAll`**) de **`MovieServiceTest`**

```
DataSource dataSource = new SimpleDataSource();
DataSourceLocator.addDataSource(MOVIE_DATA_SOURCE,
    dataSource);
```

  - En el constructor de **`MovieServiceImpl`**

```
dataSource = DataSourceLocator.getDataSource(
    MOVIE_DATA_SOURCE);
```

  - El valor de la constante **`MOVIE_DATA_SOURCE`** es **`ws-javaexamples-ds`**

DataSourceLocator (2)

- Sin embargo, cuando la capa Modelo se ejecuta dentro de un servidor de aplicaciones, es éste quien proporciona una implementación de un **`DataSource`** (típicamente con pool de conexiones)
- Esta será la situación cuando la capa Modelo se ejecute como parte de un servicio Web (o una aplicación Web)
- Una aplicación (aplicación Web o servicio Web) instalada en un servidor de aplicaciones Java puede obtener referencias a algunos objetos (e.g. DataSources) gestionados por el servidor de aplicaciones usando la API estándar JNDI (Java Naming and Directory Interface)

DataSourceLocator (3)

- JNDI (Java Naming and Directory Interface)
  - Es una API (**`javax.naming`**) que forma parte de la API básica de Java, para localizar y registrar objetos asociándoles nombres jerárquicos
  - Los servidores de aplicaciones exponen los objetos **`DataSource`** gestionados por el servidor de aplicaciones mediante la API de JNDI

DataSourceLocator (4)

- JNDI (cont)
  - Obtención de referencias a DataSource

```
InitialContext initialContext = new InitialContext();
DataSource dataSource = (DataSource) initialContext.lookup(
    "java:comp/env/jdbc/ws-javaexamples-ds");
```

  - **`InitialContext`** pertenece al paquete **`javax.naming`**
  - Los servidores de aplicaciones tienen la obligación de exponer los objetos **`DataSource`** en el contexto (“directorio”) **`java:comp/env`**
  - Cuando el administrador de un servidor de aplicaciones configura un **`DataSource`**, por convenio, se aconseja que lo registre bajo el subcontexto **`jdbc`**, de manera que el nombre completo queda como **`java:comp/env/jdbc/<<nombre simple del DataSource>>`**

DataSourceLocator (5)

- **`DataSourceLocator`** mantiene un mapa de DataSources indexado por nombre
- **`addDataSource(name, dataSource)`**
  - Añade el **`DataSource`** pasado como parámetro en el mapa
- **`getDataSource(name)`**
  - Comprueba si el **`DataSource`** solicitado está en el mapa
  - En caso afirmativo, lo devuelve
  - En otro caso (el **`DataSource`** no fue añadido con **`addDataSource`** => capa Modelo corre dentro de servidor aplicaciones)
    - Lo busca por JNDI
    - Lo inserta en el mapa

DataSourceLocator (6)

```
public class DataSourceLocator {

    public static final String JNDI_PREFIX = "java:comp/env/jdbc/";

    private static Map<String,DataSource> dataSources =
        Collections.synchronizedMap(new HashMap<String, DataSource>());

    private DataSourceLocator() {}

    public static void addDataSource(
        String name, DataSource dataSource) {

        dataSources.put(name, dataSource);

    }
```

Nombre “simple” (sin JNDI_PREFIX) del DataSource
(e.g. ws-javaexamples-ds)

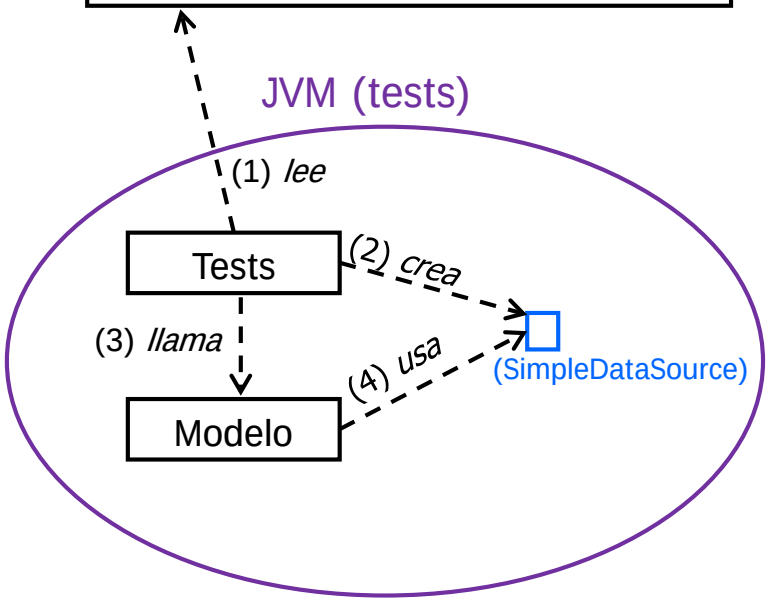
DataSourceLocator (y 7)

```
    public static DataSource getDataSource(String name)
        throws RuntimeException{

        DataSource dataSource = (DataSource) dataSources.get(name);

        if (dataSource == null) {

            try {
                InitialContext initialContext = new InitialContext();
                dataSource = (DataSource) initialContext.lookup(
                    JNDI_PREFIX + name);
                dataSources.put(name, dataSource);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }

        }
        return dataSource;
    }

} // class
```

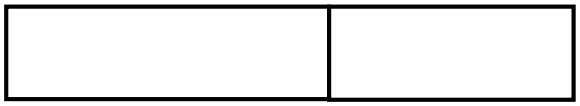
DataSourceLocator - Tests

ConfigurationParameters.properties
(src/test/resources)

```
SimpleDataSource.url=jdbc:mysql://localhost/wstest?...
SimpleDataSource.user=ws
SimpleDataSource.password=ws

...
```



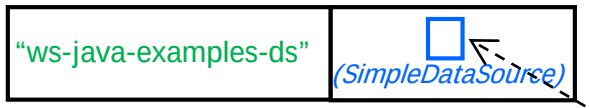
`Map <String,DataSource>`
`DataSourceLocator.datasources`



En el método `init` (`@BeforeAll`) de `MovieServiceTest`:

```
DataSource dataSource = new SimpleDataSource();
DataSourceLocator.addDataSource(MOVIE_DATA_SOURCE, dataSource);
```

`DataSourceLocator.datasources`

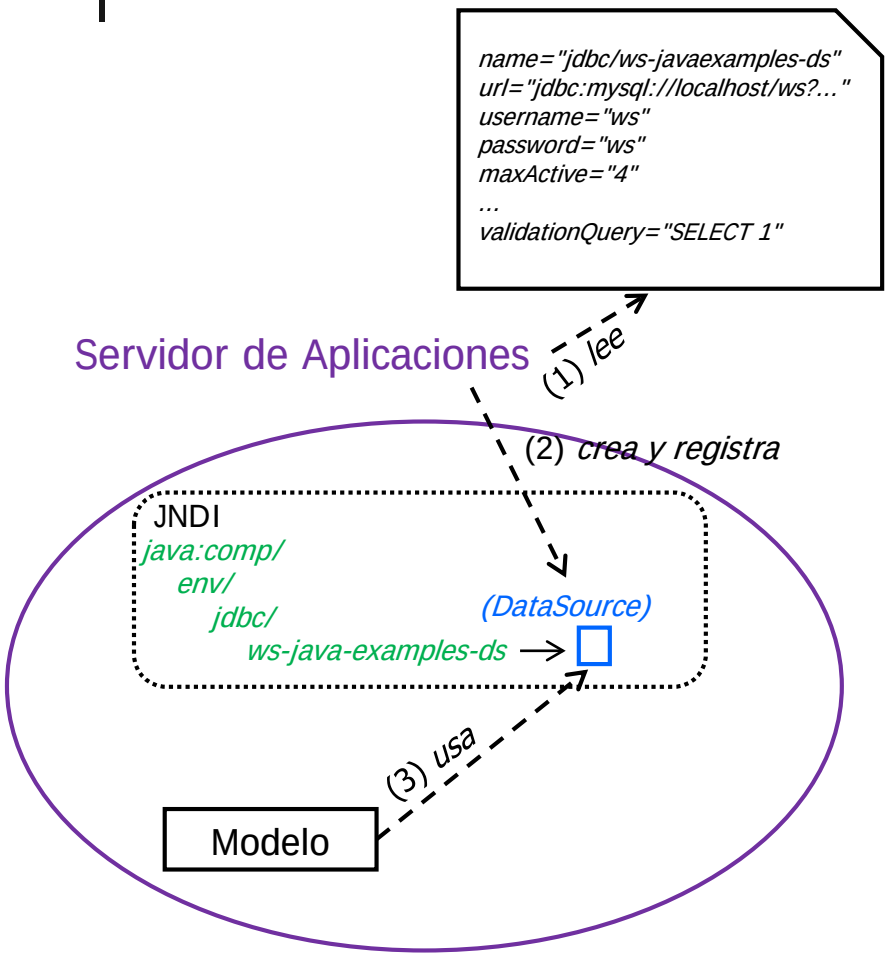



```
movieService = MovieServiceFactory.getService();
```

En el constructor de `MovieServiceImpl`:

```
dataSource = DataSourceLocator.getDataSource(MOVIE_DATA_SOURCE);
```

DataSourceLocator – Aplicación




```
Map <String,DataSource>
  DataSourceLocator.datasources
```

En el constructor de `MovieServiceImpl`:

```
dataSource = DataSourceLocator.getDataSource(MOVIE_DATA_SOURCE);
```

```
DataSourceLocator.datasources
```