Descrición do código de partida

1. Introdución

Como punto de partida para a realización do traballo ofrécese unha pequena aplicación completamente programada, que podedes utilizar para os desenvolvemento do voso traballo tutelado, substituíndo as clases do modelo, DAOs e tests polas vosas propias (as correspondentes ao dominio asignado).

A aplicación permite xestionar usuarios, sobre os que se almacena información en forma de entradas nun *log*. O modelo de datos inclúe dúas clases, *Usuario* e *EntradaLog*, entre as que se establece unha asociación "un a moitos" implementada como bidireccional

As clases do modelo están "mapeadas" utilizando anotacións estándar JPA. Para acceder aos servizos de persistencia utilízase tamén a API estándar JPA. Como implementación JPA utilízase o framework Hibernate.

2. Estrutura da aplicación.

A aplicación está implementada en forma de proxecto Maven, e inclúe polo tanto un ficheiro de configuración inicial *pom.xml*, verificado con **Maven 3.9.5**, que integra inicialmente as seguintes tecnoloxías:

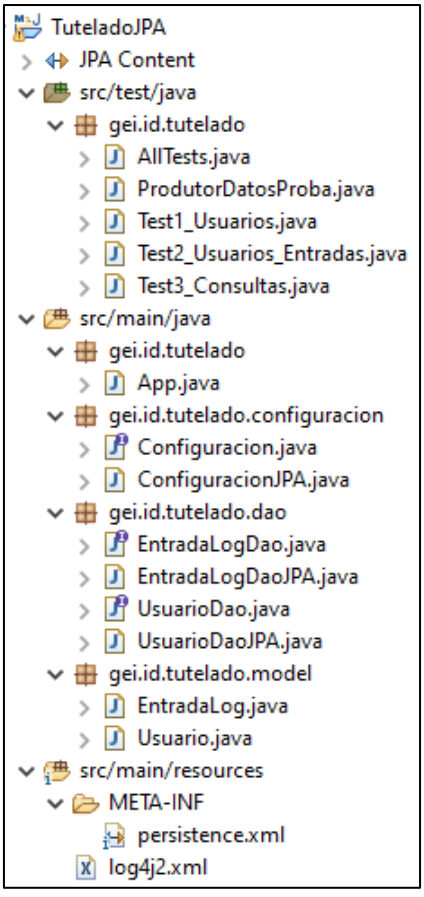

Figura 1:Estrutura e ficheiros da aplicación

- Java SE (version 11 e posteriores)
- Hibernate 5.4.21.Final
- PostgreSQL JDBC driver v4.2 (42.2.2), verificado contra servidor PostgreSQL 12.4
- JUnit 4.12
- Log4j 2.12.1

Para o proxecto Maven escolleuse o arquetipo *quickstart*. Todas as clases da aplicación están incluídas no paquete *gei.id.tutelado*, que á súa vez contén os seguintes paquetes:

- *gei.id.tutelado.model:* clases do modelo
- *gei.id.tutelado.dao:* clases DAO
- *gei.id.tutelado.configuracion*: clases para configurar os DAOs

3. Configuración

a) JPA + Hibernate

A configuración básica realízase no ficheiro ***META-INF/persistence.xml,*** no que se definen:

i) Unidade de persistencia (única) da aplicación - que leva por nome *TuteladoPU* - e clases que a integran.

```
<persistence-unit name="TuteladoPU" transaction-type="RESOURCE_LOCAL">
    <description>
    Unidade de persistencia (única) para traballo tutelado de ID
    </description>

    <!-- Persistence provider -->
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <!-- Entity classes -->
    <class>gei.id.tutelado.model.Usuario</class>
    <class>gei.id.tutelado.model.EntradaLog</class>

    <exclude-unlisted-classes>true</exclude-unlisted-classes>
```

Figura 2:Definición, en persistence.xml, das propiedades da unidade de persistencia.

Como se pode ver:

- Indícase que Hibernate será o provedor de persistencia
- Indícanse as clases que inclúen anotacións persistentes
- Indícase que SÓ as clases sinaladas deben ser examinadas por Hibernate, ignorando as demais. Isto permite (se o necesitamos) probar de forma illada o mapeo de certas clases do noso modelo (e non o de todas).

ii) Propiedades de configuración do acceso á base de datos (BD) relacional:

```
<properties>
    <property name="hibernate.connection.url" value="jdbc:postgresql://localhost:5432/tuteladoDB" />
    <property name="hibernate.connection.driver_class" value="org.postgresql.Driver" />
    <property name="hibernate.connection.username" value="hib" />
    <property name="hibernate.connection.password" value="hib123" />
```

Figura 3: Definición, en persistence.xml, da configuración de acceso á BD

AVISO: A BD e o usuario *deben ser creados previamente a man* en PostgreSQL para que a aplicación funcione. Podedes usar estes ou outros no voso TT (se son outros, modificade a configuración definida en *persistence.xml*).

iii) Propiedades de configuración específicas de Hibernate:

```
<property name="hibernate.show_sql" value="false" />
<property name="hibernate.format_sql" value="false" />
<property name="hibernate.hbm2ddl.auto" value="create" />
    <!--
        Valores para hbm2ddl.auto

        validate: validate the schema, makes no changes to the database.
        update: update the schema.
        create: creates the schema, destroying previous data.
        create-drop: drop the schema at the end of the session.

        Ollo! Lembrar facer EntityManagerFactory.close()

    -->
```

Figura 4:Definición, en persistence.xml, das propiedades específicas de Hibernate.

Como se pode ver, a opción *hibernate.hbm2ddl.auto* esta fixada a *"create".* Iso significa que Hibernate creará as táboas da BD "de cero" ao arrancar a aplicación, a partir das regras de mapeo definidas. As táboas permanecerán na BD ao rematar a execución (o cal nos permite comprobar que o mapeo funciona correctamente); e serán eliminadas (e creadas de novo) na seguinte execución.

b) Log4j

A nosa aplicación utiliza a libraría *Log4j* para producir mensaxes informativas durante a execución do código, coa posibilidade de filtralas en función do seu nivel de importancia/gravidade. As mensaxes rexístranse nun *logger* (que podemos ver como unha especie de "ficheiro de log" configurable).

No código de todos os tests da aplicación creamos unha instancia da clase *Logger,* asociada a un *logger* denominado *gei.id.tutelado*.

```
private Logger log = LogManager.getLogger("gei.id.tutelado");
```

Figura 5: Definición no código Java dunha instancia de Logger

Cada mensaxe rexistrada no *logger* debe ser etiquetada en función da súa importancia/relevancia/gravidade. As categorías son: FATAL, ERROR, WARN, INFO, DEBUG, TRACE.

Para rexistrar unha mensaxe no *logger* coa etiqueta apropiada, utilizamos os métodos de *Logger*: por exemplo, *Logger.info()* ou *Logger.error()*.

```
log.info("Probando gravacion de usuario con Nif duplicado ----------------------------------------------");
produtorDatos.u1.setNif(produtorDatos.u0.getNif());
```

Figura 6: Xeración de mensaxe con etiqueta INFO no código da aplicación

A configuración de log4j defínese no ficheiro **log4j2.xml**

```
<Loggers>
  <!--  Configuracion xeral -->
  <Root level="fatal">
    <AppenderRef ref="Console"/>
  </Root>
  <!--  O noso codigo -->
      <logger name="gei.id.tutelado" level="all"/>
  <!--  Hibernate -->
      <logger name="org.hibernate" level="error"/>
      <logger name="org.hibernate.SQL" level="debug"/>

</Loggers>
```

Figura 7: Definición, en log4j2.xml, das propiedades dos loggers involucrados na aplicación.

No dito ficheiro definimos:

- O lugar a onde deben enviarse (e quedar rexistradas) por defecto todas as mensaxes rexistradas a través dos nosos *loggers*. No noso caso, a consola.
- Un nivel de filtrado de *ERROR* para o *logger* principal incorporado "de serie" no código de Hibernate (*org.Hibernate*). Omítense así as mensaxes informativas normais lanzadas durante a execución (non se mostrarán na consola) e móstranse só as de erro.
- Un nivel de filtrado de *DEBUG* para o *logger* de SQL incorporado "de serie" no código de Hibernate (*org.Hibernate.SQL*). Este nivel permite a visualización na consola das mensaxes que informan das consultas SQL lanzadas por Hibernate contra a BD.
- O *logger* específico para a nosa aplicación (*gei.id.tutelado*), cun nivel de filtrado establecido como *ALL* (dese xeito móstranse na consola todas as mensaxes producidas no código da aplicación, teñan a etiqueta que teñan).

Os niveis de filtrado establecidos son os recomendados para executar a aplicación. Porén, podedes modificar en calquera momento o nivel de filtrado de ***cada*** *logger*, para aumentar/diminuír o nivel de detalle das mensaxes mostradas a través del.

4. *ConfiguracionJPA*

A clase *ConfiguracionJPA* é a encargade de crear (e destruír) a (única) instancia de *EntityManagerFactory* que usaremos durante a execución da aplicación / tests.

```
public class ConfiguracionJPA implements Configuracion {

    private EntityManagerFactory emf=null;

    @Override
    public void start() {
        emf = Persistence.createEntityManagerFactory("TuteladoPU");
    }

    @Override
    public Object get(String artifact) {
        if (artifact.equals("EMF")) {
            return emf;
        }
        throw new IllegalArgumentException();
    }

    @Override
    public void endUp() {
        emf.close();
    }
}
```

Figura 8: Código da clase ConfiguracionJPA

5. Aplicación

A aplicación está pensada para probar a creación automática da BD e a validez (ou non) das nosas regras de mapeo. Non fai ningún tipo de proba adicional: Non crea, recupera nin elimina obxectos da BD (implementaremos esas probas na forma de casos de proba JUnit)

O código da aplicación é moi simple:

a) Crea unha instancia de *ConfigurationJPA*
b) Invoca a *ConfigurationJPA.start()* para crear unha instancia de *EntityManagerFactory*
c) Invoca a *ConfigurationJPA.endUp()* para que a dita instancia de *EntityManagerFactory* sexa destruída.

```
public class App
{
    public static void main( String[] args )
    {
        Configuracion cfg;
        cfg = new ConfiguracionJPA();
        cfg.start();
        cfg.endUp();
    }
}
```

Figura 9: Código principal da aplicación

6. DAOS

Na aplicación foron definidas dúas interfaces DAO, unha por cada clase do modelo: *UsuarioDAO* e *EntradaLogDAO*. Nelas definimos todas as operacións de persistencia de forma xenérica, e sen ter en conta a forma de implementalas (JDBC directamente, JPA...).

As clases *UsuarioDAOJPA* e *EntradaLogDAOJPA* constitúen a implementación, baseada no uso de JPA, desas interfaces.

A implementación dos DAOs presupón o uso de *EntityManagers* xestionados por aplicación. Para poder facelo, os DAOS deben recibir, no momento de seren inicializados, unha instancia de *EntityManagerFactory* (creada previamente e compartida entre eles) que se lles facilita a través dun método denominado *setup()*.

```
private EntityManagerFactory emf;
private EntityManager em;

@Override
public void setup (Configuracion config) {
    this.emf = (EntityManagerFactory) config.get("EMF");
}
```

Figura 10: Paso de instancia de EntityManagerFactory a un DAO

A dita instancia de *EntityManagerFactory* permite que cada método do DAO xestione **o seu propio** *EntityManager* e **as súas propias** transaccións.

```
@Override
public Usuario almacena(Usuario user) {

    try {
        em = emf.createEntityManager();
        em.getTransaction().begin();

        em.persist(user);

        em.getTransaction().commit();
        em.close();

    } catch (Exception ex ) {
        if (em!=null && em.isOpen()) {
            if (em.getTransaction().isActive()) em.getTransaction().rollback();
            em.close();
            throw(ex);
        }
    }
    return user;
}
```

Figura 11: Exemplo de método de DAO con xestión propia do EntityManager

7. Casos de proba con *JUnit*

Para poder probar os DAOs, a aplicación conta con tres clases con casos de proba *JUnit*:

- *Test1_Usuarios*: Probas de operacións DAO aplicadas sobre usuarios sen entradas de log asociadas (para probar de forma illada o mapeo da clase *Usuario*).
- *Test2_UsuariosConEntradas:* Probas de operacións DAO aplicadas sobre usuarios e entradas de log en conxunto
- *Test3_Consultas*: Para probar todo tipo de consultas extra definidas a maiores das operacións CRUD básicas.

Os tres casos de proba (que contan con varios métodos separados) están incluídos nunha *suite* denominada *AllTests*. Inclúen anotacións que permiten que sexan executados vía `mvn install`.

7.1. Inicialización das clases de proba.

Cada clase de proba estará encargada de:

a) Crear unha instancia de *ConfigurationJPA* (e, por conseguinte, tamén unha instancia de *EntityManagerFactory*).
b) Crear unha instancia de **cada** clase DAO que sexa necesaria para realizar as probas, (pasándolles, vía *setup(),* a instancia de *EntityManagerFactory* creada previamente).
c) Crear unha instancia de *ProdutorDatosProba* (pasándolle, vía *Setup(),* a instancia de *EntityManagerFactory* creada previamente). A clase inclúe métodos para crear un conxunto inicial de datos (tanto en memoria como na BD) necesarios para executar cada proba individual; e tamén para eliminalos unha vez finalizada a dita proba.

A idea é crear unha **única** vez as ditas instancias, e compartilas entre **todos** os métodos/probas @Test da clase. E, unha vez rematadas todas as probas, eliminar todas esas instancias. Para conseguilo, cómpre seguir os seguintes pasos na codificación de cada clase de proba JUnit:

a) En *init()* (marcado con *@BeforeClass*):

- Obtemos unha instancia de *ConfiguracionJPA* (e polo tanto tamén de *EntityManagerFactory*) que queda almacenada na variable estática *cfg*.
- Obtemos instancias de cada DAO que precisemos utilizar, que quedan almacenadas en sendas variables estáticas (no noso caso, *usuDao* e *logDao*).
- Facilitamos a instancia de *ConfiguracionJPA* a cada instancia DAO a través do método *setup* (para proporcionarlle así acceso a instancia de *EntityManagerFactory* creada previamente).
- Obtemos unha instancia de *ProdutorDatosProba*. Esa clase inclúe métodos para producir (e gravar) instancias de clases do modelo que poderemos usar durante as probas.

```java
@BeforeClass
public static void init() throws Exception {
    cfg = new ConfiguracionJPA();
    cfg.start();

    usuDao = new UsuarioDaoJPA();
    logDao = new EntradaLogDaoJPA();
    usuDao.setup(cfg);
    logDao.setup(cfg);

    produtorDatos = new ProdutorDatosProba();
    produtorDatos.Setup(cfg);
}
```

Figura 12: Inicialización de clase JUnit

b) En *endclose*() (marcado con *@AfterClass*):

- Destruímos a instancia (**única**) de *EntityManagerFactory* creada para **todas** as probas da clase, invocando o método *endUp* da instancia de *ConfiguracionJPA*.

```java
@AfterClass
public static void endclose() throws Exception {
    cfg.endUp();
}
```

Figura 13: Finalización de clase JUnit

7.2. A clase *ProdutorDatosProba*

Cada instancia da clase *ProdutorDatosProba* permite crear instancias das clases do modelo, que logo poderemos utilizar nas nosas probas.

```java
public void creaUsuariosSoltos() {

    // Crea dous usuarios EN MEMORIA: u0, u1
    // SEN entradas de log

    this.u0 = new Usuario();
    this.u0.setNif("000A");
    this.u0.setNome("Usuario cero");
    this.u0.setDataAlta(LocalDate.now());

    this.u1 = new Usuario();
    this.u1.setNif("111B");
    this.u1.setNome("Usuaria un");
    this.u1.setDataAlta(LocalDate.now());

    this.listaxeU = new ArrayList<Usuario> ();
    this.listaxeU.add(0,u0);
    this.listaxeU.add(1,u1);

}
```

Figura 14: Método de ProdutorDatosProba que produce dúas instancias de Usuario para usar nas probas.

A clase inclúe tamén:

- Un método (*gravaUsuarios*) que grava na BD todos os usuarios creados (e as súas entradas de log, se as teñen), para cando iniciemos casos de proba que precisen a existencia de información previa recollida na BD.
- Un método (*limpaBD*) que elimina o contido de todas as táboas da BD.

7.3. Implementación das probas

A execución de cada proba marcada con @Test constara de varias fases:

1. Limpeza: Borramos completamente a BD para empezar a proba desde cero
2. Inicialización: Creamos en memoria (e na BD, se é preciso) os datos de partida necesarios para poder executar a proba.
3. Execución: Execución propiamente dita da proba, que inclúe a invocación de todos os métodos DAO necesarios, e a verificación do resultado.

1) Limpeza

Eliminamos o contido previo de todas as táboas da BD, para poder comezar desde cero, invocando a *ProdutorDatosProba.limpaBD()*.

Facémolo no método *setUp()* da clase JUnit (marcado con *@Before*) para non ter que repetilo en cada *@Test*

```
@Before
public void setUp() throws Exception {
    log.info("");
    log.info("Limpando BD ------------------------
    produtorDatos.limpaBD();
}

@After
public void tearDown() throws Exception {
}
```

Figura 15: Borrado da BD que se executa antes de cada método @Test

2) Inicialización

Invocamos aos métodos de *ProdutorDatosProba* necesarios para establecer a situación inicial de partida da proba. Neste caso xa temos que incluír código específico en **cada** metodo/proba *@Test* en función dos datos que precisemos.

```
@Test
public void t2_CRUD_TestRecupera() {

    Usuario u;

    log.info("");
    log.info("Configurando situación de partida do test ----------

    produtorDatos.creaUsuariosSoltos();
    produtorDatos.gravaUsuarios();
```

Figura 16: Inicio dunha proba, con rexistro de datos iniciais.

3) Execución

Tamén no código de cada método/proba @Test invocamos normalmente aos métodos dos DAOs, dos que (lembremos) xa contamos con instancias (compartidas) creadas previamente (Ver Figura 12: Inicialización de clase JUnit)

```
Assert.assertNull(produtorDatos.u0.getId());
usuDao.almacena(produtorDatos.u0);
Assert.assertNotNull(produtorDatos.u0.getId());
```

Figura 17: Invocación de métodos de DAO

8. Execución do código

a) Para probar o código nunha consola, situarse no directorio principal (o directorio no que está almacenado o ficheiro *pom.xml*) e executar:

```
$ mvn clean
$ mvn install
```

b) Para instalar o proxecto en Eclipse, executar:

```
$ mvn eclipse:clean
$ mvn eclipse:eclipse
```

(e entón importar o proxecto en Eclipse, como proxecto Maven)

c) Para entregar o código, executar:

```
$ mvn package -D maven.test.skip=true
```

(crea no directorio *target* un ficheiro comprimido "*TuteladoJPA-1.1-src.zip*")