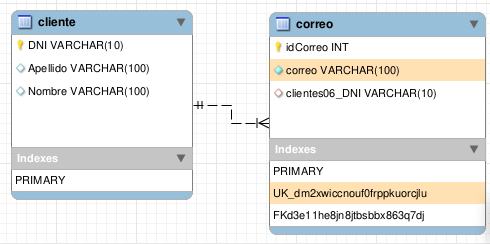
Crear una aplicación JPA, Hibernate y DAO con una vista de terminal. Usar la gestión de dependencias con Maven y el motor de base de datos MySQL.

Requerimiento: Crear una base de datos con las tablas llamadas Cliente y Correo, agregar las funcionalidades GRUD a cada tabla, la relación entre tablas es: un Cliente puede tener muchos correos.

Esquema de la base de datos



Datos de acceso a la base de datos para un entorno de desarrollo y para un entorno de producción.

<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">

<persistence-unit name="DBCliente\_Produccion" transaction-type="RESOURCE\_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

<properties>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/dbclienteproduccion?serverTimezone=UTC"/>

<property name="javax.persistence.jdbc.user" value="carlosproduccion"/>

<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>

<property name="javax.persistence.jdbc.password" value="12345678"/>

<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>

<property name="javax.persistence.schema-generation.database.action" value="none"/>

</properties>

</persistence-unit>

<persistence-unit name="DBCliente\_Desarrollo" transaction-type="RESOURCE\_LOCAL">

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

<properties>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/dbclientedesarrollo?zeroDateTimeBehavior=CONVERT\_TO\_NULL"/>

<property name="javax.persistence.jdbc.user" value="carlosdesarrollo"/>

<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>

<property name="javax.persistence.jdbc.password" value="12345678"/>

<property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>

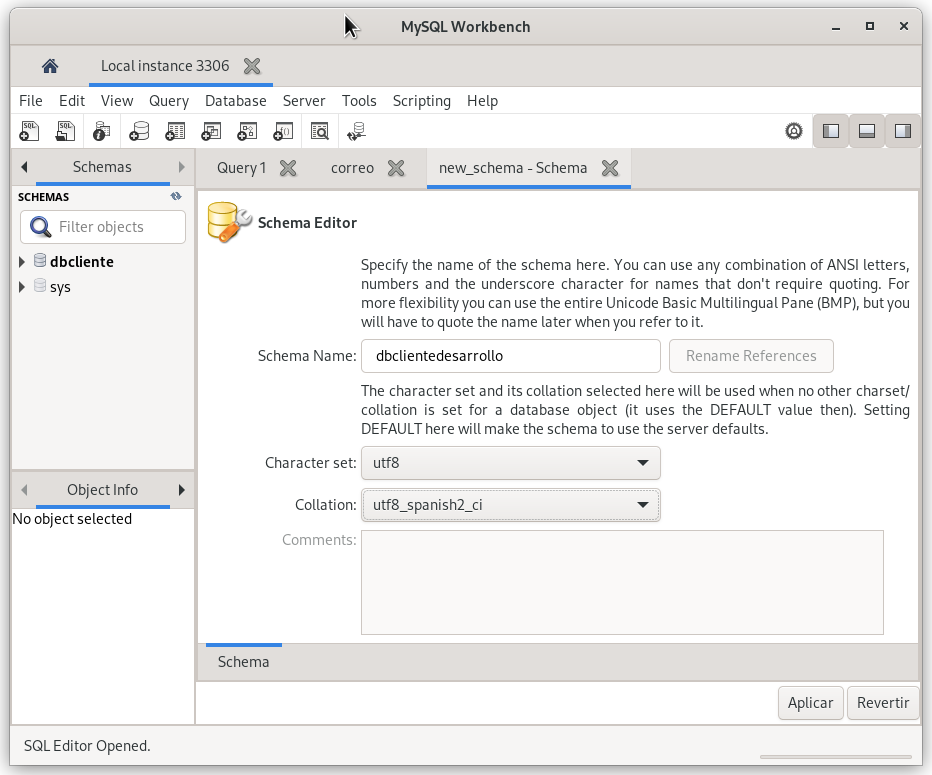
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>

</properties>

</persistence-unit>

</persistence>

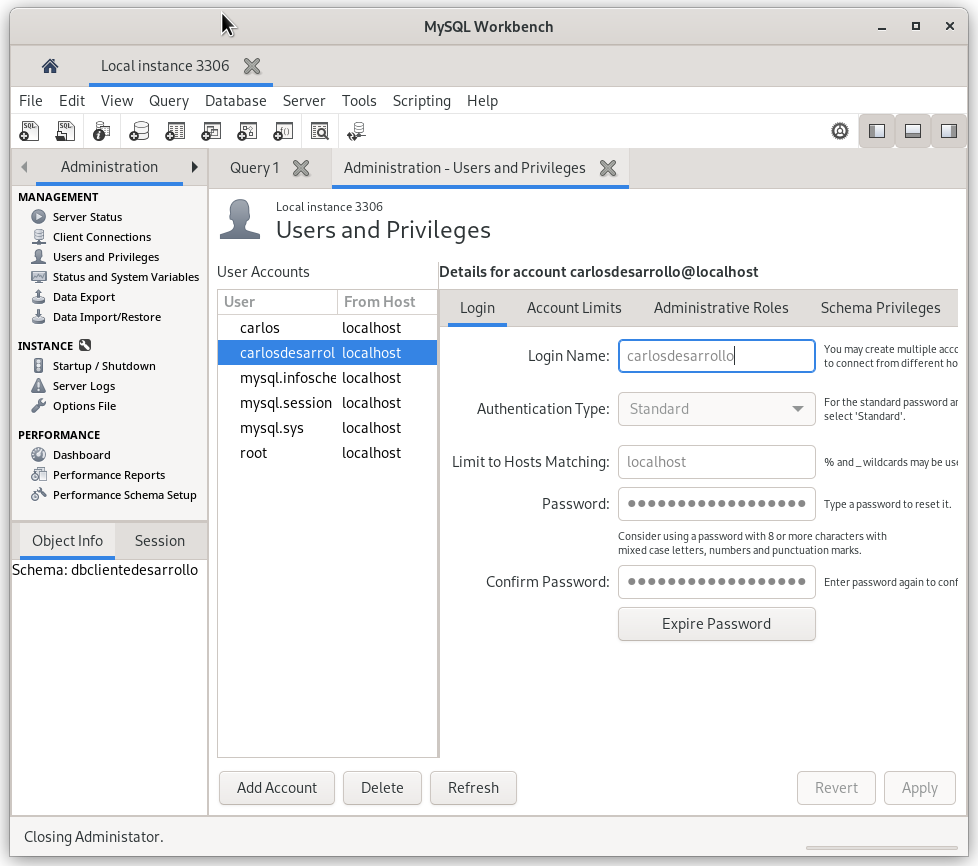
Paso N.º 01 – Crear la base de datos

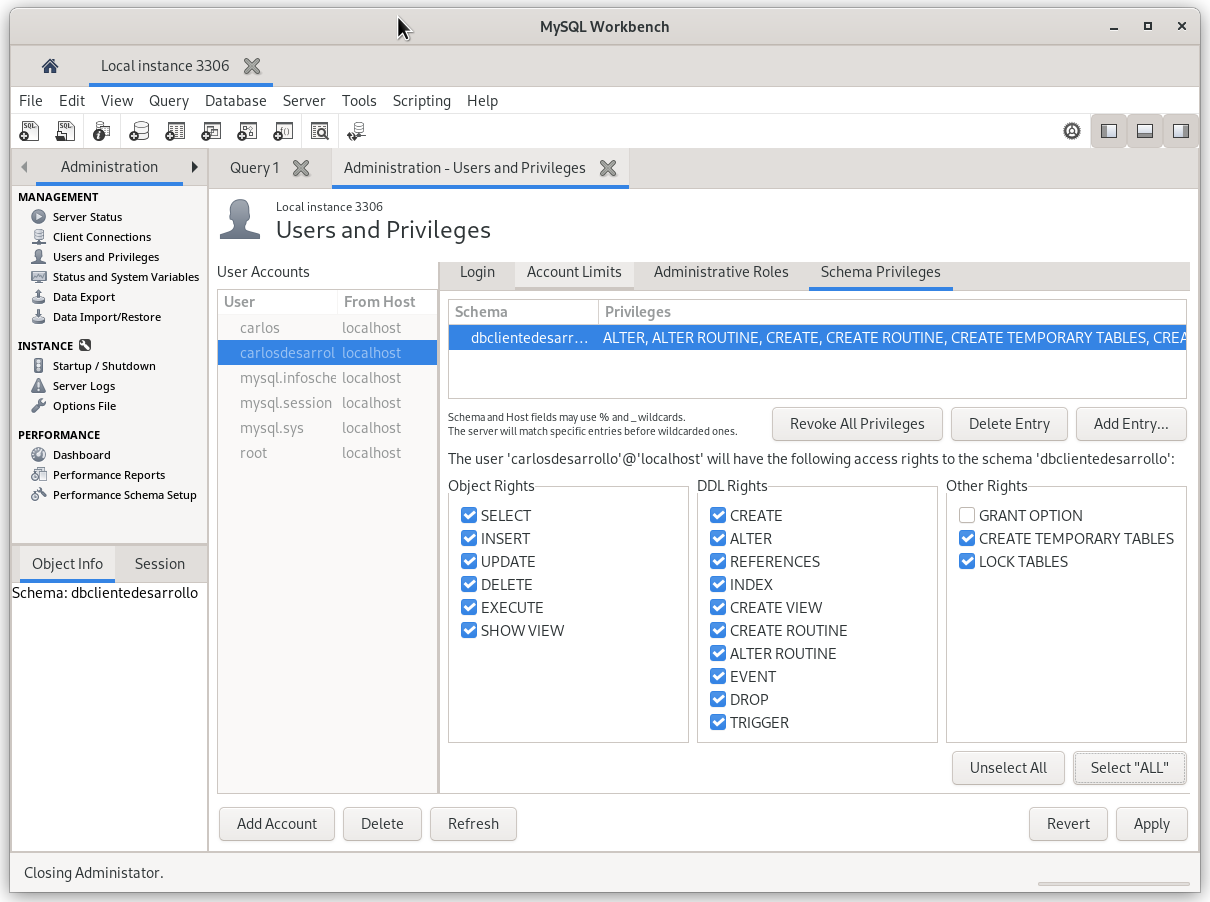


Paso N.º 02 – Crear el usuario de desarrollo y asignarle la nueva base de desarrollo.

Tecnologías usadas para este proyecto

JPA (Java Persistence API): Es una API de Java que proporciona una interfaz para el manejo de persistencia de datos en aplicaciones Java. JPA simplifica el acceso y la manipulación de datos en una base de datos relacional utilizando entidades Java y consultas orientadas a objetos.





DAO (Data Access Object): Es un patrón de diseño de software que se utiliza para separar la lógica de acceso a datos de la lógica de negocio en una aplicación. El objetivo del patrón DAO es proporcionar una capa de abstracción entre la aplicación y la fuente de datos, lo que facilita la gestión y manipulación de datos.

Hibernate: Es un framework de mapeo objeto-relacional (ORM) que se utiliza junto con JPA para simplificar la interacción con la base de datos. Hibernate proporciona una implementación de JPA y ofrece características adicionales, como el mapeo de objetos a tablas de base de datos y la gestión automática de consultas.

MySQL: Es un sistema de gestión de bases de datos relacional muy utilizado. En los ejemplos, se mencionó la dependencia de MySQL Connector/J, que es el controlador JDBC para interactuar con bases de datos MySQL desde una aplicación Java.

Maven: Maven es una herramienta de gestión de dependencias para proyectos en la construcción de un producto software. Puede ser utilizada en el desarrollo de software en Java. Facilita la construcción de software reutilizando artefactos de software de terceros o propios, gestión de dependencias y automatización de tareas en un proyecto. En los ejemplos mencionados, se utilizó el plugin Maven Jar para generar el archivo JAR con las dependencias incluidas.

Diagrama de clases



Arquitectura o estructura de las capas DAO en Java:

Capa de Entidades (Clase entidades: Correos06 y Clientes06):

En esta capa se definen las clases que representan las entidades de negocio, como por ejemplo: Clientes, Pedidos, Productos, etc. Cada clase de entidad mapea directamente a una tabla en la base de datos y contiene los atributos y métodos relacionados con dicha entidad. Para mapear las clases entidad y las tablas se utilizan las anotaciones JPA.

Estas son algunas de las anotaciones JPA usadas para mapear la clase Clientes06 a tabbla llamada cliente del la bases de datos MySQL:

* @Entity
* @Table(name = "cliente" )
* @Column(name="DNI", nullable=false)
* @Id
* @Column(name="Nombre")
* @Column(name="Apellido")

Capa de Acceso a Datos (Acceso a los datos - DAO):

En esta capa se definen las interfaces y las implementaciones de los DAO (Data Access Objects).

Los DAO son responsables de interactuar con la base de datos y realizar operaciones de lectura/escritura sobre las entidades. Cada entidad tendría su correspondiente interfaz DAO y su implementación concreta.

Esta capa quedaría representada por las clases: DAOGRUD03, AbstractDAO05 y la clase de la API Java EntityManagerUtil04

Capa de Servicios (Servicios o funciones de negocio):

En esta capa se definen las clases de servicios que actúan como intermediarios entre las capas superiores (como la capa de presentación) y la capa de acceso a datos.

Los servicios encapsulan la lógica de negocio y utilizan los DAO para realizar operaciones en la base de datos. Aquí se definen métodos que permiten realizar operaciones como crear, leer, actualizar y eliminar entidades.

Esta capa quedaría representada por las clases controladoras ClienteDAO07 y CorreoDAO07

Capa de Presentación o vista (Vista):

En esta capa se encuentran las clases que interactúan directamente con el usuario y las clases controladoras que a su vez acceden a las capas de datos.

Puede ser una interfaz de Terminal, una interfaz de usuario (GUI) o una API REST en el caso de las aplicaciones Web o de celulares.

Las clases de esta capa utilizan los servicios expuestos por las clases controladoras para realizar las operaciones de negocio y presentar los resultados al usuario.

Existen otros patrones de diseño como es el MVC. El patrón MVC es una forma de organizar y estructurar una aplicación en tres componentes principales:

Modelo (Model): Representa los datos y la lógica de negocio de la aplicación.

En el ejemplo anterior, las entidades y los DAO se pueden considerar como el Modelo, ya que representan las entidades de negocio y contienen la lógica para acceder y manipular los datos.

Vista (View): Es responsable de la presentación de los datos al usuario.

En el ejemplo, la capa de presentación (Presentation Layer) se puede considerar como la Vista, ya que contiene las clases que interactúan directamente con el usuario y presentan los resultados de las operaciones.

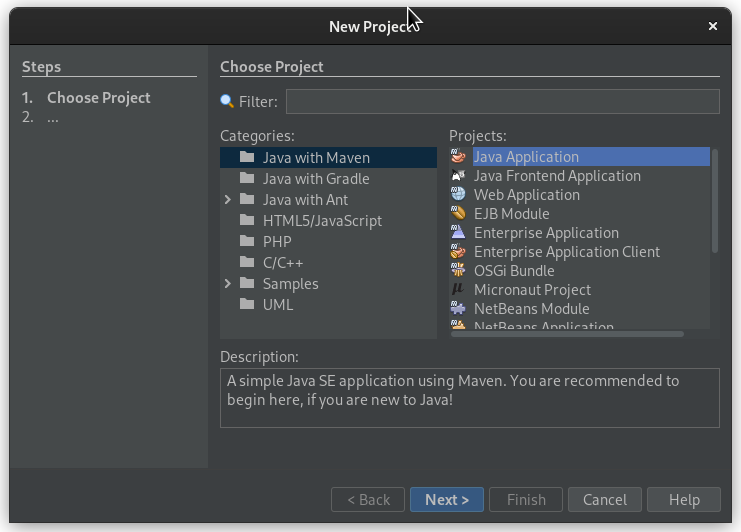
Controlador (Controller): Actúa como intermediario entre el Modelo y la Vista. En el ejemplo, los servicios (Service Layer) podrían considerarse como el Controlador, ya que encapsulan la lógica de negocio y utilizan el Modelo (entidades y DAO) para realizar las operaciones. En este enfoque, el Controlador se encarga de recibir las solicitudes del usuario desde la Vista, interactúa con el Modelo para realizar las operaciones necesarias y luego actualiza la Vista con los resultados correspondientes. Esta separación de responsabilidades permite una mejor organización del código y facilita el mantenimiento y la escalabilidad de la aplicación. Muchos Framework usan la arquitectura o patrón de diseño MVC para organizar las aplicaciones.

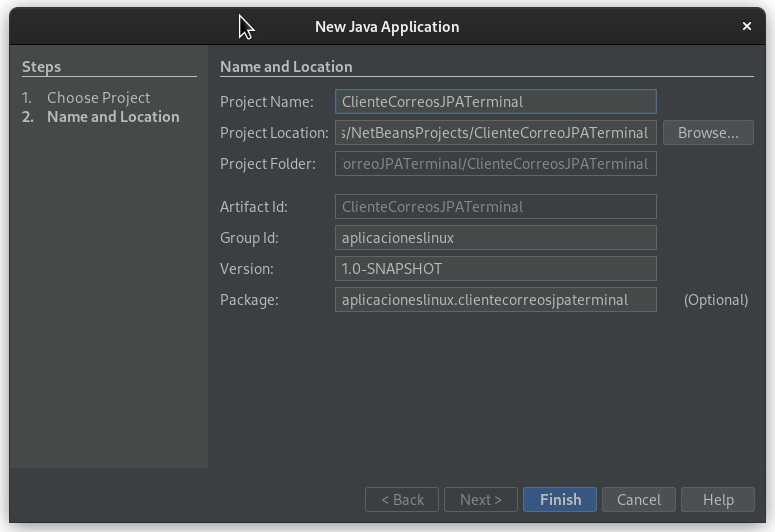
Las arquitecturas de aplicaciones por capas o por el patrón MVC se utiliza normalmente sin importar si es una aplicación monolito o aplicación basada en micro-servicios. Es muy importante aprender a diseñar aplicaciones por capas o aprender a utilizar un Framework que implemente el patrón de diseño MVC.

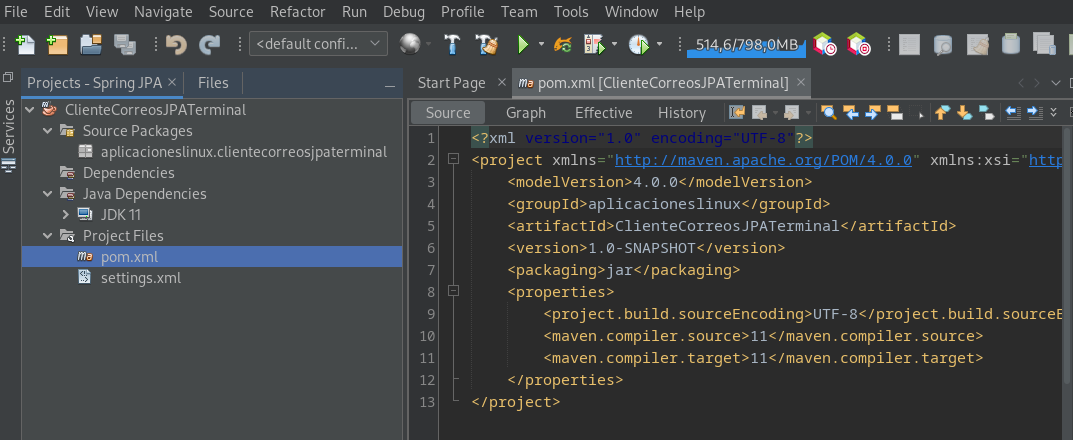
Este es un ejemplo de una aplicación diseñada o de arquitectura por capas. Las arquitecturas por capas son fáciles de escalar, agregar nuevas funcionalidades, agregar nuevos requerimientos de usuario, remplazar la capa de vista por distintas tecnologías de FrontEnd, remplazar la capa de datos por distintas tecnologías de BackEnd.

Paso N.º 01 – Crear el proyecto POM.xml

Crear un nuevo proyecto en NetBeans







El archivo POM.xml creado es muy básico, hay que agregar las dependencia faltantes y la configuración de las estructuras de paquetes para los test y para localizar la clase que contiene el método main().

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>aplicacioneslinux</groupId>

<artifactId>ClienteCorreosJPATerminal</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>11</maven.compiler.source>

<maven.compiler.target>11</maven.compiler.target>

</properties>

</project>

Este es el POM.xml que incluye las ordenes básicas para compilar, crear el archivo jar y encontrar el método main() para ejecutar. Esta configuración en el archivo pom.xml es usada para crear el archivo de manifiesto en el directorio META-INF.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>AplicacionesLinux</groupId>

<artifactId>ClienteCorreosJPATerminal</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>11</maven.compiler.source>

<maven.compiler.target>11</maven.compiler.target>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<release>11</release>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.2.0</version>

<configuration>

<archive>

<manifest>

<addClasspath>true</addClasspath>

<classpathPrefix>lib/</classpathPrefix>

<mainClass>aplicacioneslinux.clientecorreosjpaterminal.MainClass</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

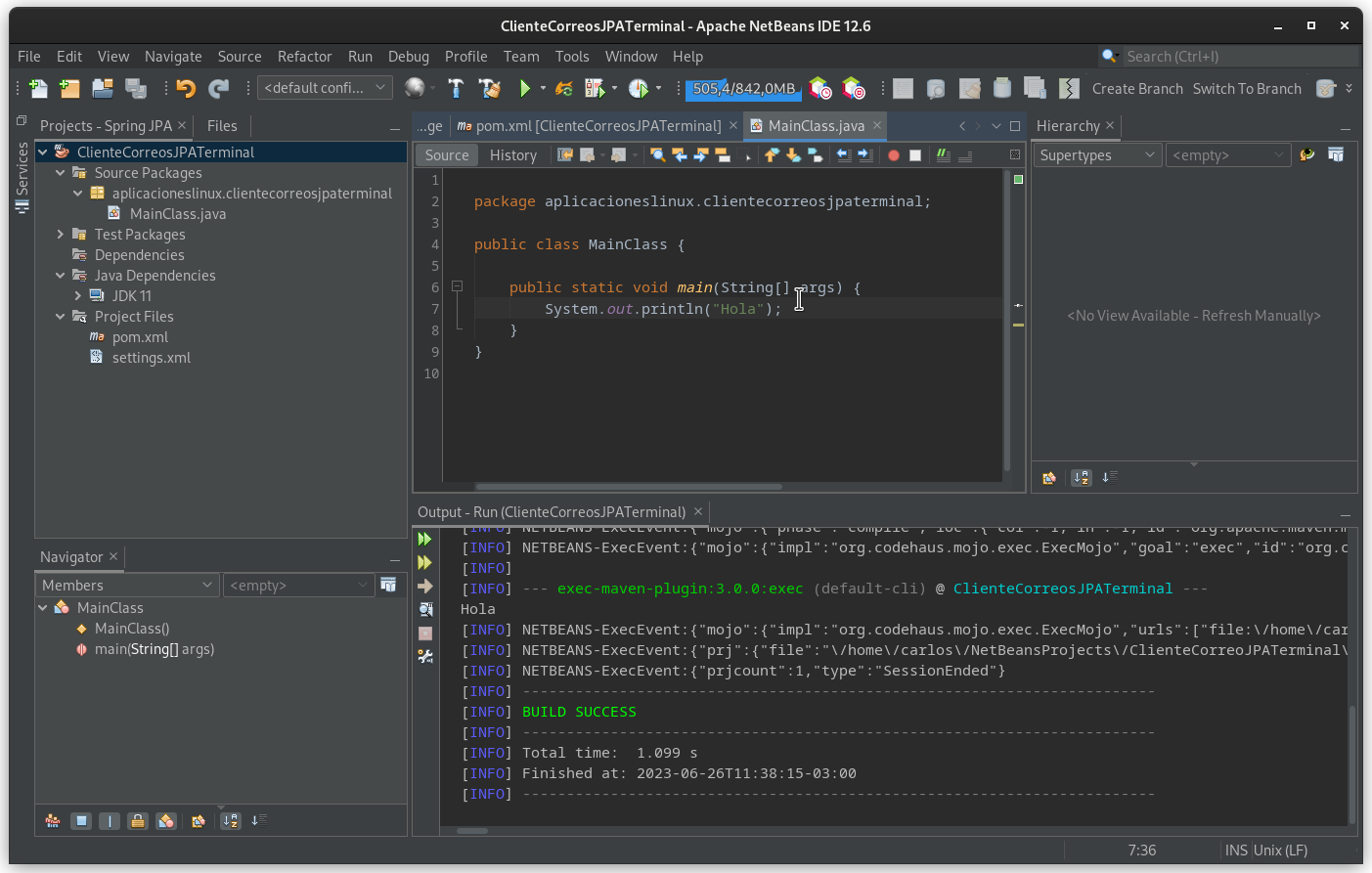
</plugins>

</build>

</project>

Agregar una clase con el método main() según se ha especificado en el archivo POM.xml y correr el programa para verificar que compila.

<mainClass>aplicacioneslinux.clientecorreosjpaterminal.MainClass</mainClass>



Realizar un clean and build para generar el archivo jar. Ejecutar el archivo jar y verificar que está todo bien.

Configuración adicional pero no obligatoria en el archivo POM.xml

El archivo POM.xml debe ubicarse en el directorio raíz del proyecto, el directorio raiz es conocido normalmente como el “resource directory”. Dicho directorio debe configurarse en las propiedades del proyecto en el IDE y en el archivo POM.xml.

Estas configuraciones en el archivo pom.xml no son necesarias, pero en algunas ocasiones los IDEs requieren se sean configuradas.

El siguiente fragmento del archivo POM.xml muestra la configuración de los directorios más habituales para configurar los directorios más inportantes de un proyecto:

* Directorio raíz del proyecto <directory>${basedir}</directory>
* Directorio de los fuentes el proyecto <sourceDirectory>src/main/java</sourceDirectory>
* Directorio de salida del proyecto <outputDirectory>target/classes/</outputDirectory>

…

<build>

<sourceDirectory>src/main/java</sourceDirectory>

<outputDirectory>target/classes/</outputDirectory>

<resources>

<resource>

<directory>${basedir}</directory>

<includes>

<include>\*</include>

</includes>

<filtering>true</filtering>

</resource>

</resources>

</build>

…

La variable ${basedir} en el código se refiere al directorio base del proyecto, es decir, el directorio en el que se encuentra el archivo pom.xml. Maven reemplazará automáticamente ${basedir} con la ruta absoluta al directorio base del proyecto durante el proceso de compilación.

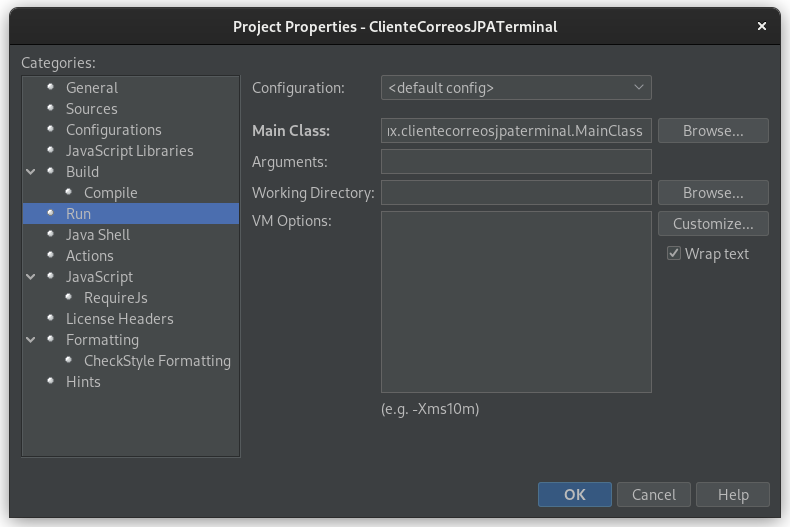
En este caso, la línea <directory>${basedir}</directory> indica que se utilizará el directorio base del proyecto como directorio de recursos para los archivos a incluir en la construcción. El atributo <filtering>true</filtering> indica que los recursos se filtrarán durante la construcción, lo que significa que se reemplazarán las variables con sus valores correspondientes.

Explicación de cada elemento en el código:

* <build>: Define la configuración de construcción del proyecto.
* <sourceDirectory>src/main/java</sourceDirectory>: Especifica el directorio fuente principal donde se encuentran los archivos fuente Java del proyecto.
* <outputDirectory>target/classes/</outputDirectory>: Indica el directorio de salida donde se colocarán los archivos compilados (clases) después de la construcción.
* <resources>: Define los recursos (archivos no Java) que se incluirán en la construcción.
* <resource>: Representa un recurso específico que se incluirá en la construcción.
* <directory>${basedir}</directory>: Indica el directorio base del proyecto como directorio de recursos para este recurso en particular.
* <includes><include>\*</include></includes>: Especifica que se deben incluir todos los archivos en el directorio de recursos.
* <filtering>true</filtering>: Habilita el filtrado de recursos, lo que permitirá la sustitución de variables.

En resumen, este código configura la estructura de directorios y los recursos para la construcción de un proyecto Maven, utilizando el directorio base del proyecto como directorio de recursos del proyecto y filtrando o recorriendo los archivos en el directorio base del proyecto para reemplazar las variables definidas en ellos durante la construcción.

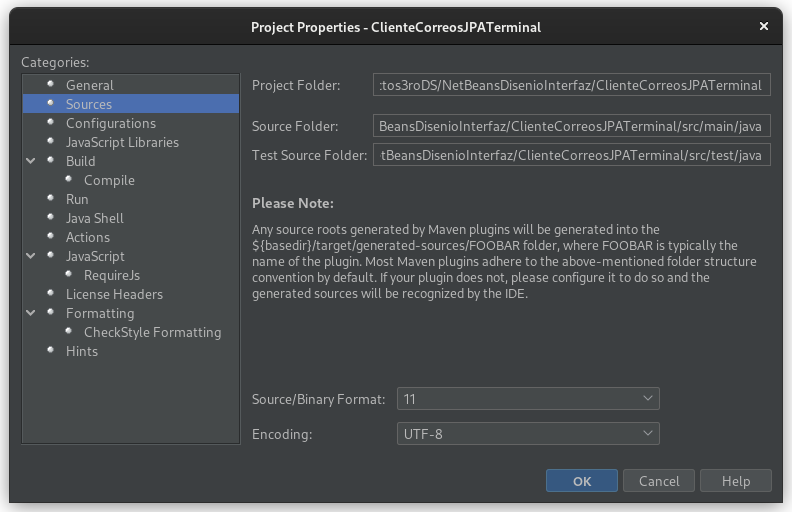
Para ejecutar la aplicación desde el proyecto creado por Netbeans hay que indicar en las propiedades del proyecto el camino a la clase que tiene el método main().



Entonces hay dos lugares donde se debe configurar el lugar donde se ha programado el método men().

1. En el archivo pom.xml (Para compilar y generar el archivo \*.jar con el comando build)
2. En las propiedades del proyecto (para ejecutar el proyecto con el comando run)

En la imagen podemos ver la configuración de las propiedades del proyecto en NetBeans.



Source Folder: .../ClienteCorreosJPATerminal/src/main/java

../ClienteCorreosJPATerminal/... es el directorio raiz del proyecto, también es conocido como directorio base.

.../src/main/java/… es el directorio para los archivos fuentes y para otros paquetes con archivos fuentes o recursos para compilar o incluir en el archivo JAR.

Fin del Paso N.º 01

Paso N.º 02 – Crear el archivo persistence.xml

<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence

http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">

<persistence-unit name="DBCliente" transaction-type="RESOURCE\_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

<properties>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/dbcliente?serverTimezone=UTC"/>

<property name="javax.persistence.jdbc.user" value="carlos"/>

<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>

<property name="javax.persistence.jdbc.password" value="12345678"/>

<property name="javax.persistence.schema-generation.database.action" value="update"/>

<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect" />

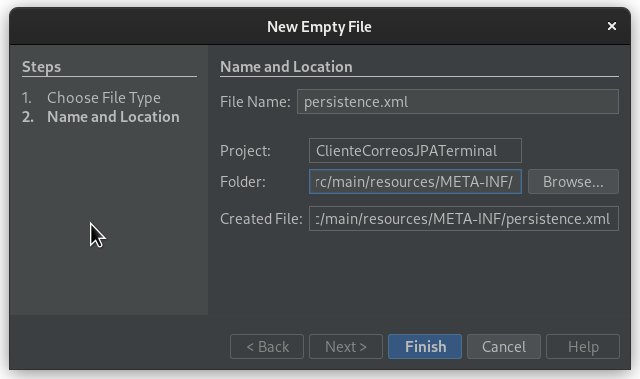
</properties>

</persistence-unit>

</persistence>

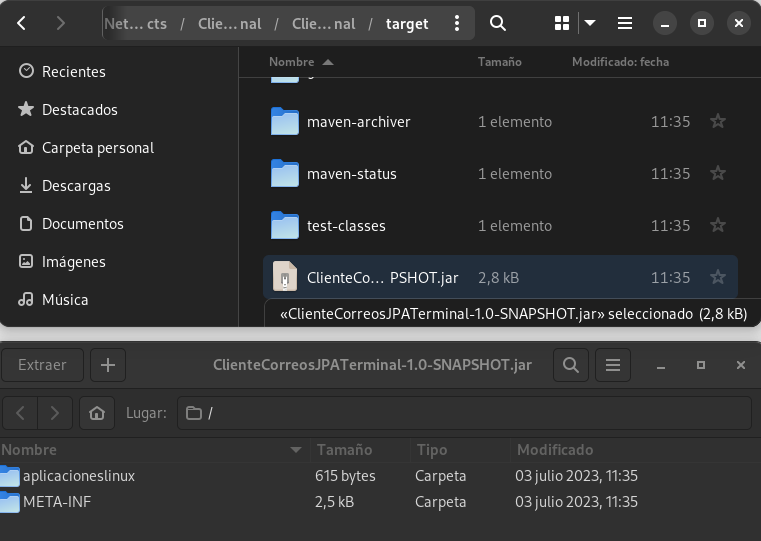
El archivo de persistencia de los datos usando JPA que mantendrá el nombre de la unidad de persistencia y los datos de la conexión al motor de la base de datos MySQL. Se debe crear en el directorio META-INF

En la imagen podemos ver la interfaz gráfica del NetBeans para crear el archivo de persistencia

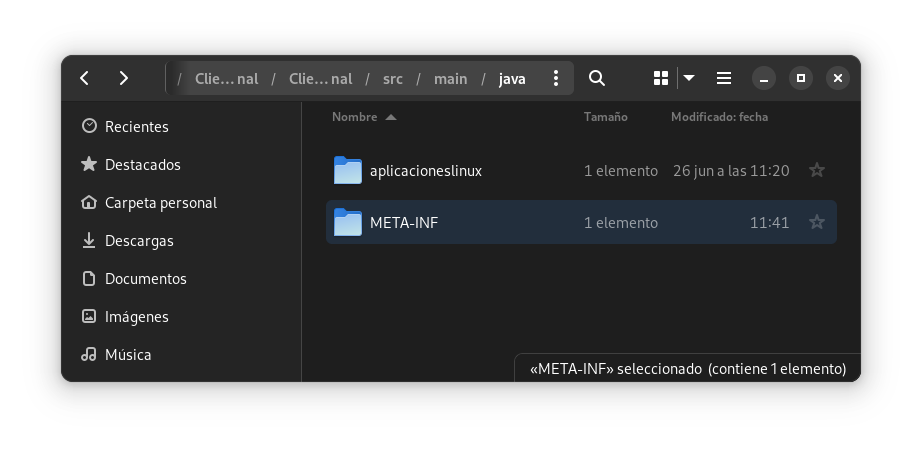


Técnica para saber donde crear la carpeta META-INF

Para saber donde crear el directorio META-INF hay que abrir el archivo \*.jar generado con clean and build y verificar el nombre de la carpeta creada como el directorio raíz para los archivos fuentes de la aplicación.



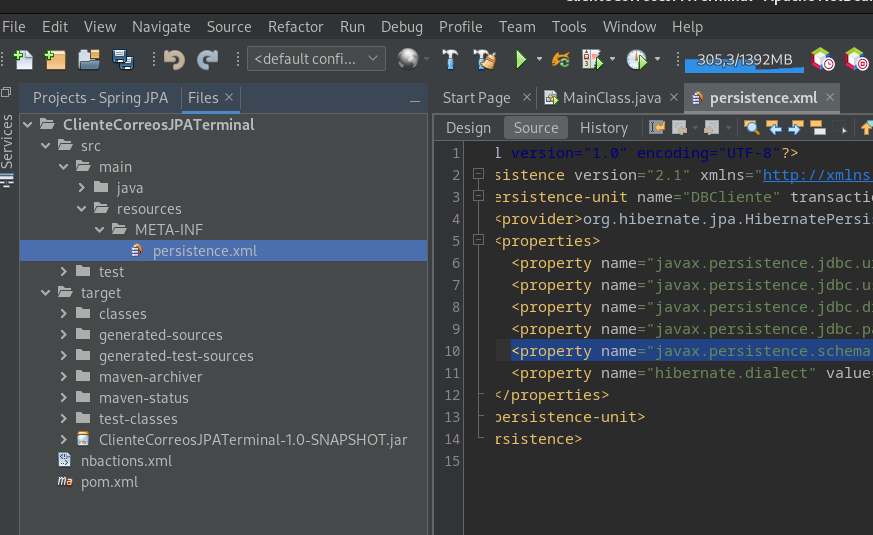
En la siguiente imagen podemos ver que la carpeta META-INF fue creada junto a la carpeta aplicacioneslinux que a su vez están contenidas en la carpeta llamada .../main/. La carpeta llamada java es la carpeta raíz para todos los archivos que se deben compilar e incluir dentro del archivo \*.jar y entregar al usuario de la aplicación.



En conclusión, hay dos directorios raíz para el proyecto.

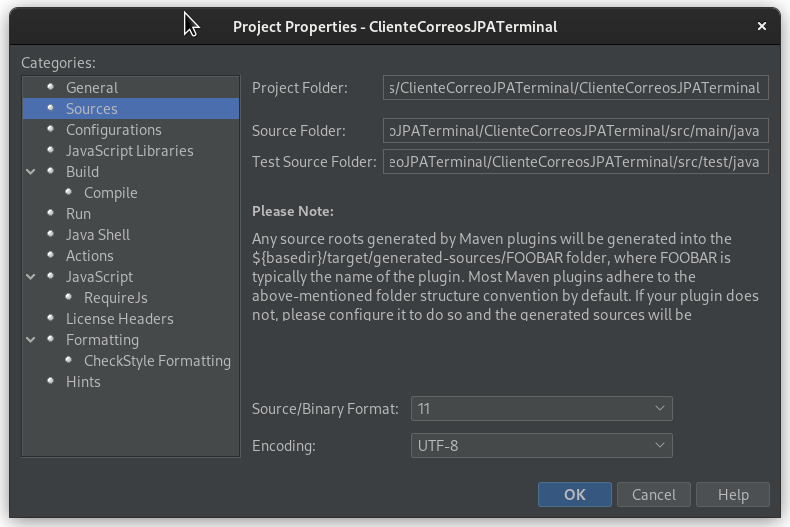
1. El directorio raíz para los recursos del proyecto (el contenido no se incluye en el archivo \*.jar, por ejemplo el archivo POM.xml)
2. El directorio raíz llamado .../main/para los paquetes y fuentes del proyecto (todos los archivos incluidos en esta carpeta se compilará y será incluido dentro del archivo \*.jar, por ejemplo la configuración a la conexión MySQL y otras configuraciones, archivos de dependencias y archivos de librerías de terceros)

La siguiente imagen muestra la ubicación del archivo persistence.xml. El mencionado archivo se localiza en el directorio META-INF.



El directorio META-INF debe localizarse en algún directorio que esté indicado como “source directory” en las propiedades del proyecto en el IDE y en el archivo POM.xml. En la siguiente imagen podemos ver que el primer recurso es el directorio llamado java.

Configuración en las propiedades del proyecto en NetBeans



Configuración en el archivo POM.xml

…

<sourceDirectory>src/main/</sourceDirectory>

…

En conclusión, los archivos fuentes y recursos para compilar e incluirlos en el archivo \*.jar deben configurarse en dos lugares.

1. En propiedades del proyecto en el IDE NetBeans
2. En el archivo POM.xml (Es opcional cuando se usa la estructura por defecto)

Este es una estructura de carpetas por defecto para un proyecto Maven

**- Carpeta del proyecto (raíz del proyecto)**

- src

**- main (raíz de los recursos)**

- java

- (paquetes y fuentes Java)

- resources

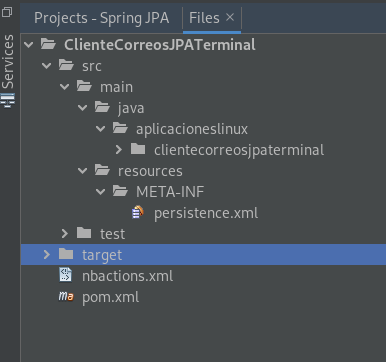
- META-INF

- persistence.xml

- test

- target

- pom.xml



Todas las carpetas que están dentro del main formarán parte del archivo \*.jar.

El resto de las carpetas son recursos del proyecto.

Probar que todo funciona correctamente

Tarea de verificación y control.

1. Hacer Run para ejecutar la aplicación, no debería haber errores.
2. Hacer Build para generar el archivo \*.JAR y verificar que no hay errores.
3. Ejecutar el archivo \*.jar

Salida por la terminal

:~/.../ClienteCorreosJPATerminal/target$ ls

classes generated-sources maven-archiver test-classes

ClienteCorreosJPATerminal-1.0-SNAPSHOT.jar generated-test-sources maven-status

:~/.../ClienteCorreosJPATerminal/target$ java -jar ClienteCorreosJPATerminal-1.0-SNAPSHOT.jar

Hola ...

Paso N.º 03 – Crear la clase del tipo interfaz DAOCRUD03

package modelodatos;

import java.util.List;

public interface DAOGRUD03<T> {

//Métodos abstractos GRUD create, read, update, delete

void guardar(T t); //Create //Crear

List<T> listarTodo();//Read //Leer

void editar(T t); //Update //Actualizar

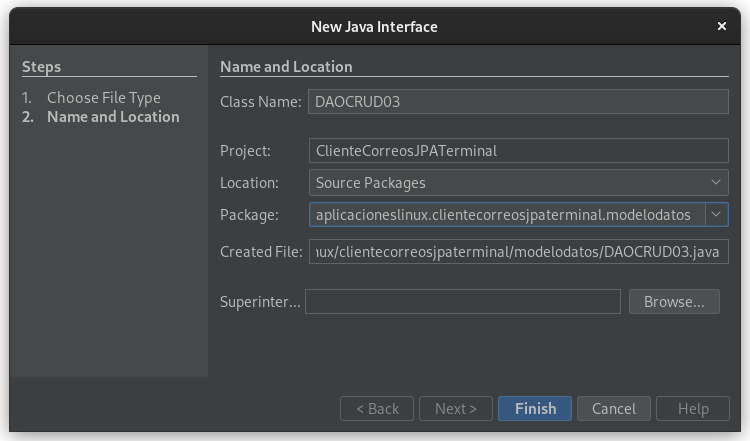
void borrar(T t); //Delete //Borrar

///////////////////////////////////////

//Otros métodos relacionados con los requerimientos del sistema o reglas de negocios

///////////////////////////////////////

}



Paso N.º 04 – Crear la clase EntityManagerUtil04

package modelodatos;

import javax.persistence.EntityManager;

import javax.persistence.EntityManagerFactory;

import javax.persistence.Persistence;

public class EntityManagerUtil04 {

public EntityManagerUtil04() {

super();

}

private static EntityManager manager = null;

public static EntityManager getEntityManager() {

if(manager==null) {

EntityManagerFactory factory = Persistence.createEntityManagerFactory("DBCliente");

manager = factory.createEntityManager();

}

return manager;

}

//public static void main(String[] args) {

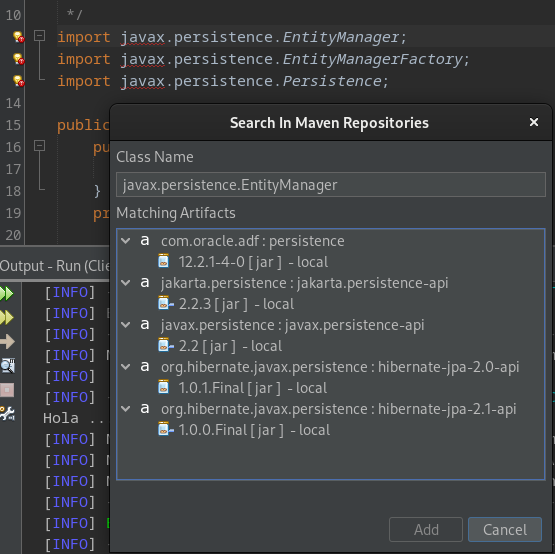
// EntityManager manager = EntityManagerUtil04.getEntityManager();

// System.out.println("Se ha creado un objeto EntityManager ==> " + manager.getClass().getCanonicalName());

//}

}

En caso de error en el paquete javax, buscar la dependencia correspondiente en los repositorios Maven.



La forma correcta de salvar el error en el paquete llamado javax es instalar las dependencias agregando el siguiente código en el archivo POM.xml.

<dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.6.8.Final</version>

</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.28</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-agroal -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-agroal</artifactId>

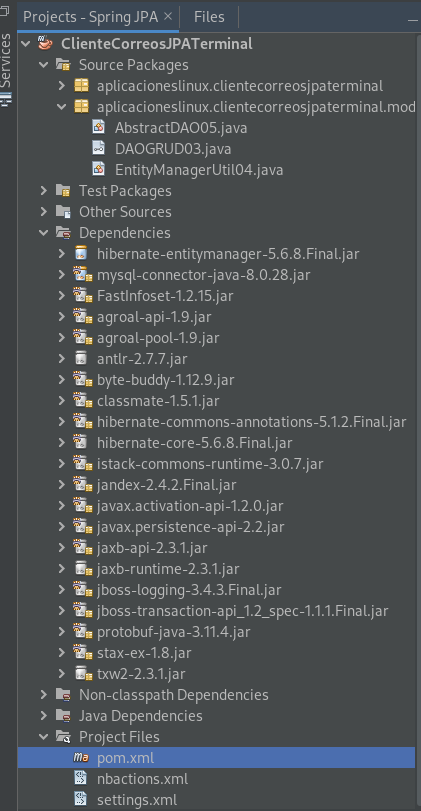
<version>5.6.4.Final</version>

<type>pom</type>

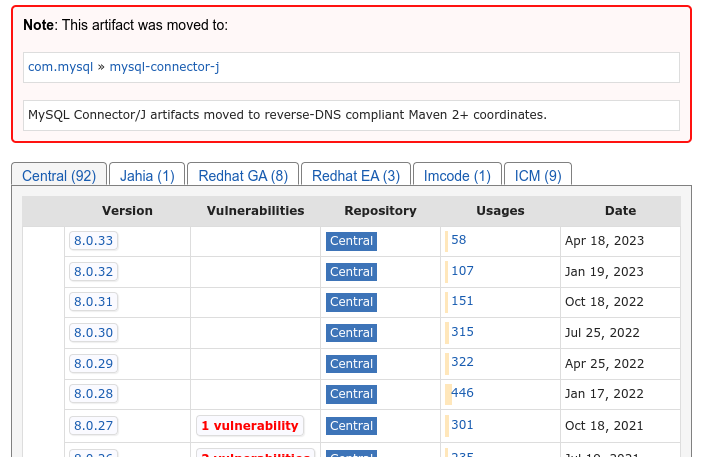
</dependency>

</dependencies>

Verificamos la carpeta de archivos de dependencia donde se han descargado los archivos desde los repositorios maven



Verificar en la página de: <https://mvnrepository.com/artifact/mysql/mysql-connector-java> que las versiones descargadas no tienen vulnerabilidades o estén muy desactualizados.



Maven está informando que el conector se ha movido al grupo llamado com.mysql con otro nombre de artefacto. También está informando que la versión 8.0.28 no tiene vulnerabilidades conocidas. De igual forma hay que revisar todos los demás artefactos incluidos en el archivo POM.xml

Paso N.º 05 – Crear la clase AbstractDAO05

La clase AbstractDAO05<T> implemente la interfaz abstracta DAOGRUD03<T> y debe implementar todos sus métodos abstractos, en caso que renuncie a implementar uno de los métodos de la clase DAOGRUD03<T> entonces la clase AbstractDAO05<T> deberá declararse como abstracta y dejar que las clases controladoras se responsabilicen de escribir el código faltante.

Esta es la forma correcta de trabajar en un equipo de desarrollo de un producto software, el equipo seguramente a elegido alguna metodología Ágil para trabajar. Cada vez que un requerimiento ha cambiado o surjan nuevos requerimientos simplemente se seguirán los los pasos enumerados a continuación.

1. Agregar el nuevo requerimiento a una clase interfaz en la forma de un método abstracto. (Equipo de análisis funcional, diseño)
2. Implementar el requerimiento en la clase abstracta que implemente la interfaz. (equipo de programación)
3. Crear el controlador para refinar la implementación y ofrecer la nueva funcionalidad como un servicio a otras capas del sistema, por ejemplo a la capa de la vista del sistema. Los controladores normalmente se relacionan con las vistas FrontEnd, y con las entidades BackEnd. (Equipo de desarrollo)
4. Hacer pruebas, integrara desplegar. (Equipo DevOps)
5. Otras capas del sistema consumen la nuevas funcionalidades. (Equipo de desarrollo)

package modelodatos;

import java.util.List;

import java.util.function.Consumer;

import javax.persistence.EntityManager;

import javax.persistence.EntityTransaction;

import javax.persistence.Query;

public class AbstractDAO05<T> implements DAOGRUD03<T> {

private EntityManager entityManager = EntityManagerUtil04.getEntityManager();

private Class<T> entidadTipoT; //Cliente{} o Correo{} o OtraClase{} o ...

public Class<T> getEntidadTipoT() { //Pregunta con la entidad que se está trabajando

return entidadTipoT;

}

public void setEntidadTipoT(Class<T> entidadTipoT) { //Asignar una entidad de algún tipo T

this.entidadTipoT = entidadTipoT;

}

public AbstractDAO05() {

super();

}

//Implementar los métodos CRUD heredades desde la interface

//Estos métodos pueden ser sobre-escritos desde las entidades tipo T

@Override

public List<T> listarTodo() {

// TODO Implement this method

String sqlString = "FROM " + entidadTipoT.getName();

Query query = entityManager.createQuery(sqlString);

return query.getResultList();

}

@Override

public void guardar(Object t) {

// TODO Implement this method

executeInsideTransaccion(entityManager -> entityManager.persist(t));

}

@Override

public void editar(Object t) {

// TODO Implement this method

executeInsideTransaccion(entityManager -> entityManager.merge(t));

}

@Override

public void borrar(Object t) {

// TODO Implement this method

executeInsideTransaccion(entityManager -> entityManager.remove(t));

}

//Transacciones que se pueden deshacer

private void executeInsideTransaccion(Consumer<EntityManager> action) {

EntityTransaction tx = entityManager.getTransaction();

try {

tx.begin();

action.accept(entityManager);

tx.commit();

} catch (RuntimeException er) {

tx.rollback();

throw er;

} catch (Exception er) {

tx.rollback();

throw er;

}

}

public EntityManager getEntityManager() {

return entityManager;

}

///////////////////////////////////////

//Otros métodos relacionados con los requerimientos del sistema o reglas de negocios

// globales a todas las entidades

///////////////////////////////////////

}

Tarea de verificación y control.

1. Hacer Run para ejecutar la aplicación, no debería haber errores.
2. Hacer Build para generar el archivo \*.JAR y verificar que no hay errores.
3. Ejecutar el archivo \*jar

Dado que la nueva clase programada o configuración realizada no depende de clases no creadas se podrá realizar pruebas de ejecución, esta técnica permite hacer pruebas continuas de ejecución.

Paso N.º 06 – Crear las clases que son entidades del dominio del problema a resolver

Primero crear las entidades que no se relacionan con otras entidades

Las entidades son clases que representan las tablas de la base de datos, la anotación @Entity identificará a una clase como una entidad, el manejador de entidades indicado en el archivo de persistencia operará sobre las tablas de la base de datos.

package entidades;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "cliente" )

public class Clientes06 {

public Clientes06() {

super();

}

public Clientes06(String dni, String nombre, String apellido) {

this.dni = dni;

this.nombre = nombre;

this.apellido = apellido;

}

@Column(name="DNI", nullable=false)

@Id

private String dni;

@Column(name="Nombre")

private String nombre;

@Column(name="Apellido")

private String apellido;

public void setDni(String dni) {

this.dni = dni;

}

public String getDni() {

return dni;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

public void setApellido(String apellido) {

this.apellido = apellido;

}

public String getApellido() {

return apellido;

}

@Override

public String toString() {

// TODO Implement this method

return "id=" + dni + ", Nombre=" + nombre + ", " + apellido + "\n";

}

}

Tarea de verificación y control.

1. Hacer Run para ejecutar la aplicación, no debería haber errores.
2. Hacer Build para generar el archivo \*.JAR y verificar que no hay errores.
3. Ejecutar el archivo \*jar

Dado que la nueva clase programada o configuración realizada no depende de clases no creadas se podrá realizar pruebas de ejecución, esta técnica permite hacer pruebas continuas de ejecución.

package entidades;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.ManyToOne;

import javax.persistence.Table;

@Entity

@Table(name = "correo")

public class Correos06 {

public Correos06() {

super();

}

@Id

@Column(name = "idCorreo", length = 6)

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int idCorreo;

@Column(name = "correo", length = 100, unique=true, nullable=false)

private String correo;

@ManyToOne(fetch = FetchType.LAZY) //Un cliente muchos correos

private Clientes06 clientes06=null; //Se crea un objeto tipo Cliente06{}

public Correos06(String correo, Clientes06 clientes06) {

super();

this.correo = correo;

this.clientes06 = clientes06; //El cliente relacionado con este correo

}

public void setIdCorreo(int idCorreo) {

this.idCorreo = idCorreo;

}

public int getIdCorreo() {

return idCorreo;

}

public void setCorreo(String correo) {

this.correo = correo;

}

public String getCorreo() {

return correo;

}

public void setClientes06(Clientes06 clientes06) {

this.clientes06 = clientes06;

}

public Clientes06 getClientes06() {

return clientes06;

}

@Override

public String toString() {

// TODO Implement this method

return "id=" + idCorreo + ", Correo=" + correo + "\n";

}

}

Tarea de verificación y control.

1. Hacer Run para ejecutar la aplicación, no debería haber errores.
2. Hacer Build para generar el archivo \*.JAR y verificar que no hay errores.
3. Ejecutar el archivo \*.jar

Dado que la nueva clase programada o configuración realizada no depende de clases no creadas se podrá realizar pruebas de ejecución, esta técnica permite hacer pruebas continuas de ejecución.

Paso N.º 07 – Crear todas las clases controlador, una para cada entidad

Las clases controladoras son una interfaz entre la vista y las entidades del sistema. Hay varias arquitecturas o combinaciones que se pueden hacer dependiendo si es una aplicación de escritorio o una aplicación de terminal o una aplicación Web.

package control;

import modelodatos.AbstractDAO05;

import entidades.Clientes06;

import java.util.List;

import javax.persistence.Query;

public class ClientesDAO07 extends AbstractDAO05<Clientes06> {

public ClientesDAO07() {

super();

this.setEntidadTipoT(Clientes06.class);

}

////Escribir los métodos que son reglas de negocios personalizados para los clientes

public Clientes06 unCliente(String dni) {

String sql = "FROM " + Clientes06.class.getName() + " WHERE dni='" + dni + "'";

Query query = getEntityManager().createQuery(sql).setMaxResults(1);

return (Clientes06) query.getSingleResult();

}

public List<Clientes06> listarClientes() {

String sql = "FROM " + Clientes06.class.getName();

Query query = getEntityManager().createQuery(sql);

return query.getResultList();

}

///////////////////////////////////////

//Opcional: Sobre-escribir los métodos CRUD heredados de la clase abstracta

////////////////////////////////////////////////////

}

package control;

import modelodatos.AbstractDAO05;

import entidades.Correos06;

import java.util.List;

import javax.persistence.Query;

public class CorreosDAO07 extends AbstractDAO05<Correos06> {

public CorreosDAO07() {

super();

this.setEntidadTipoT(Correos06.class);

}

////Escribir los métodos que son reglas de negocios personalizados para los correos

public Correos06 unCorreo(int idCorreo) {

String sql = "FROM " + Correos06.class.getName() + " WHERE idCorreo=" + idCorreo;

Query query = getEntityManager().createQuery(sql).setMaxResults(1);

return (Correos06) query.getSingleResult();

}

public List<Correos06> listarCorreos(String dni) {

String sql = "FROM " + Correos06.class.getName() + " WHERE clientes06\_DNI='" + dni + "'";

Query query = getEntityManager().createQuery(sql);

return query.getResultList();

}

//////////////////////////////////////

//Opcional: Sobre-escribir los métodos CRUD heredados de la clase abstracta

////////////////////////////////////////////////////

}

Paso N.º 08 – Crear las distintas vistas: Terminal

Las vistas, dependiendo de la arquitectura seleccionada, tienen acceso a los controladores y a las entidades para realizar las operaciones GRUD.

Tipos de vistas:

* Terminal
* Interfaz gráfica
* Aplicación Web

Vista de terminal

package app;

import control.ClientesDAO07;

import control.CorreosDAO07;

import entidades.Clientes06;

import entidades.Correos06;

import java.util.List;

import javax.persistence.RollbackException;

public class App08 {

public App08() {

super();

}

private static CorreosDAO07 correosDAO=null;

private static ClientesDAO07 clienteDAO=null;

private static Correos06 correo=null;

private static Clientes06 cliente=null;

public static void main(String[] args) {

App08 app08 = new App08();

System.out.println("Inicio");

List l = null;

clienteDAO = new ClientesDAO07();

correosDAO = new CorreosDAO07();

//Dar de alta a un cliente y su correo

cliente = new Clientes06("32", "Juan", "Pérez");

clienteDAO.guardar(cliente);

correo = new Correos06("ss4ss@dddd.com", cliente);

correosDAO.guardar(correo);

//dar de alta a un cliente sin correo

cliente = new Clientes06("20", "Pedro", "Martínez");

clienteDAO.guardar(cliente);

//Dar de alta un nuevo correo al cliente con DNI 32

cliente = new Clientes06("32", "", "");

correo = new Correos06("ss44ss@dddd44.com", cliente);

correosDAO.guardar(correo);

System.out.println("Listar todos los clientes");

l = clienteDAO.listarClientes();

System.out.println(l);

System.out.println("Listar un clientes");

Clientes06 unCli = clienteDAO.unCliente("32");

System.out.println(unCli.toString());

System.out.println("Listar los correos de un cliente");

String algunDNICliente = "32";

l = correosDAO.listarCorreos(algunDNICliente);

System.out.println(l);

System.out.println("Listar un Correo");

int idCorreo = 1;

Correos06 unCorreo = correosDAO.unCorreo(idCorreo);

System.out.println(unCorreo.toString());

//Borrar el primer correo ss4ss@dddd.com para el cliente DNI 32

idCorreo = 1; //Hay que saber el idCorreo para borrar

correo = correosDAO.unCorreo(idCorreo);

correosDAO.borrar(correo);

//Borrar un cliente

try {

String idCliente = "32"; //Hay que saber el idCleinte para borrar

System.out.println("Borrando el cliente: " + idCliente);

cliente = clienteDAO.unCliente(idCliente);

clienteDAO.borrar(cliente);

}catch(RollbackException err) {

System.out.println("Error: mo es posible borrar un cliente con correos asignados. " +

err.getMessage());

}catch(Exception err) {

System.out.println("Error: " + err.getMessage());

}

}

}

Tarea de verificación y control.

1. Hacer Run para ejecutar la aplicación, no debería haber errores.
2. Hacer Build para generar el archivo \*.JAR y verificar que no hay errores.
3. Ejecutar el archivo \*.jar

Dado que la nueva clase programada o configuración realizada no depende de clases no creadas se podrá realizar pruebas de ejecución, esta técnica permite hacer pruebas continuas de ejecución.

Tareas finales:

* Verificar que la base de datos está en linea
* Cambiar en el archivo POM.xml para que se ejecute el nuevo método main()
* Cambiar en las propiedades del proyecto en NetBeans el nuevo main()
* Realizar las tareas de verificación y control

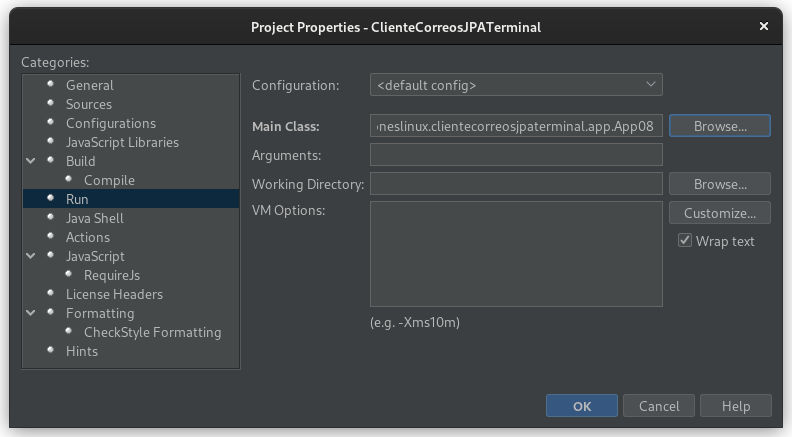
Hacer la siguiente modificación en el archivo POM.xml

...

<mainClass>aplicacioneslinux.clientecorreosjpaterminal.app.App08</mainClass>

…

Hacer la siguiente modificación en las propiedades del proyecto en NetBeans



Una vez configurado el nuevo método main() hacer las pruebas de control

Importante, si se hace un Clean and Build primero, no debería haber errores de compilación y el archivo \*.jar se bebería generarse sin problemas. Si se hace un Run en segundo lugar y se generan errores hay que verificar que la base de datos esté en linea y que corresponden los parámetros de autenticación en el archivo de persistencia.

Tarea de verificación y control usando el nuevo método main().

1. Hacer Run para ejecutar la aplicación, no debería haber errores.
2. Hacer Build para generar el archivo \*.JAR y verificar que no hay errores.
3. Ejecutar el archivo \*.jar

Otra vista de terminal

package app;

import control.ClientesDAO07;

import control.CorreosDAO07;

import entidades.Clientes06;

import entidades.Correos06;

import java.util.List;

import java.util.Scanner;

public class AppVista08 {

private static List l = null;

private static CorreosDAO07 correosDAO=null;

private static ClientesDAO07 clienteDAO=null;

private static Correos06 correo=null;

private static Clientes06 cliente=null;

public AppVista08() {

super();

}

public static void main(String[] args) {

AppVista08 appVista08 = new AppVista08();

clienteDAO = new ClientesDAO07();

correosDAO = new CorreosDAO07();

int teclaNum = 0;

Scanner miTeclado = new Scanner(System.in);

do {

System.out.print("[1]Alta cliente [2]Listar Clientes [3]Salir ==> ");

teclaNum = miTeclado.nextInt();

switch(teclaNum) {

case 1: altaCliente(); break;

case 2: listarClientes(); break;

case 3: break;

default: break;

}

}while(teclaNum != 3);

System.out.println("El programa ha terminado normalmente");

}

private static void altaCliente() {

String teclasStr = "";

Scanner miTeclado = new Scanner(System.in);

try {

System.out.print("Escribir el DNI del Cliente ==> ");

teclasStr = miTeclado.nextLine();

String dni = teclasStr;

System.out.print("Escribir el nombre del Cliente ==> ");

teclasStr = miTeclado.nextLine();

String nombre = teclasStr;

System.out.print("Escribir el apellido del Cliente ==> ");

teclasStr = miTeclado.nextLine();

String apellido = teclasStr;

cliente = new Clientes06(dni, nombre, apellido);

clienteDAO.guardar(cliente);

}catch(Exception err){

System.out.println("Error de entrada de teclado o alta cliente " + err.getMessage());

}

}

private static void listarClientes() {

//l = clienteDAO.listarClientes();

l = clienteDAO.listarTodo();

System.out.println(l);

int teclaNum = 0;

Scanner miTeclado = new Scanner(System.in);

do {

System.out.print("[1]Alta Correo [2]Listar Correos [3]Salir ==> ");

teclaNum = miTeclado.nextInt();

switch(teclaNum) {

case 1: altaCorreo(); break;

case 2: listarCorreos(); break;

case 3: break;

default: break;

}

}while(teclaNum != 3);

}

private static void altaCorreo() {

String teclasStr = "";

Scanner miTeclado = new Scanner(System.in);

try {

System.out.print("Escribir el DNI del Cliente ==> ");

teclasStr = miTeclado.nextLine();

String dni = teclasStr;

System.out.print("Escribir el queCorreo del Cliente ==> ");

teclasStr = miTeclado.nextLine();

String queCorreo = teclasStr;

cliente = new Clientes06(dni, "", "");

correo = new Correos06(queCorreo, cliente);

correosDAO.guardar(correo);

}catch(Exception err){

System.out.println("Error de entrada de teclado o alta cliente " + err.getMessage());

}

}

private static void listarCorreos() {

String teclasStr = "";

Scanner miTeclado = new Scanner(System.in);

System.out.print("Escribir el DNI del Cliente ==> ");

teclasStr = miTeclado.nextLine();

String algunDNICliente = teclasStr;

l = correosDAO.listarCorreos(algunDNICliente);

System.out.println(l);

}

}

Paso 9 – Crear el diagrama de clases UML

