**FORMATO PARA EL DESARROLLO DE COMPONENTE FORMATIVO**

| PROGRAMA DE FORMACIÓN | Análisis y desarrollo de software |
| --- | --- |

| COMPETENCIA | 220501096- Desarrollar la solución de software  de acuerdo con el diseño y las metodologías de desarrollo. | RESULTADOS DE APRENDIZAJE | 220501096-04 - Codificar el software de acuerdo con el diseño establecido. |
| --- | --- | --- | --- |

| NÚMERO DEL COMPONENTE FORMATIVO | 32 |
| --- | --- |
| NOMBRE DEL COMPONENTE FORMATIVO | FRAMEWORKS PARA CONSTRUCCIÓN DE APLICACIONES CON JAVA |
| BREVE DESCRIPCIÓN | Los requerimientos de software actuales cada vez son más exigentes en complejidad y tiempos de entrega, por lo cual se hace necesario incorporar Frameworks que faciliten la generación de código de alta calidad. A continuación, se hace la introducción al desarrollo de aplicaciones Java con los frameworks Hibernate y Sprint. |
| PALABRAS CLAVE | Hibernate, Sprint, Sprint Boot, Framework de JAVA |

| ÁREA OCUPACIONAL | 2 - CIENCIAS NATURALES, APLICADAS Y RELACIONADAS |
| --- | --- |
| IDIOMA | Español |

1. **TABLA DE CONTENIDOS:**

**Introducción**

1. **Hibernate**
   1. Operaciones básicas
   2. Asociaciones
   3. Ciclo de vida
2. **Sprint**
   1. Introducción
   2. Sprint Data
   3. Spring boot y MySQL
   4. Validaciones
3. **DESARROLLO DE CONTENIDOS:**

**Introducción**

Vamos a comenzar el presente componente formativo con un video donde se describe de una manera general la temática a tratar:

| titulo video |  |
| --- | --- |
| ruta |  |
| descripción |  |

Es en este contexto donde aparecen frameworks como Hibernate y Sprint para facilitar el proceso de transformación de información requerido entre el mundo de las bases de datos relacionales, la lógica de negocio y el mundo de la programación orientada a objetos.

1. **Hibernate**

Hibernate es software libre con licencia GNU LGPL que presta servicios de ORM para JAVA, es decir, es plugin que puede ser integrado al IDE de desarrollo en JAVA que facilita el proceso de mapeo de tablas en una base de datos relacional a objetos Java mediante el uso de archivos declarativos en formato XML o anotaciones.

Según Martinez (2015):

Hibernate utiliza JDBC para todos los procesos de comunicación con la base de datos funcionando como una capa adicional de abstracción de las funcionalidades del JDBC. En una aplicación JAVA podremos delegar la mayoría de las operaciones de bases de datos al framework de Hibernate pero en segundo plano todo estará pasando por el API de JDBC.

Así que cuando estamos configurando Hibernate para conectarse a una base de datos lo que realmente pasa es que se configura Hibernate para el Uso del JDBC y es este quien realiza las operaciones. A continuación se presenta el esquema general del funcionamiento:



Para el desarrollo de una aplicación JAVA que usa el entorno de desarrollo de Eclipse es necesario descargar todas las librerías requeridas por el framework de Hibernate las cuales puede encontrar en la página oficial del ORM <https://hibernate.org/orm/>. Una vez descargado el paquete este se debe descomprimir y podrá encontrar los archivos jar necesarios en la ubicación /lib/required los cuales deberá anexar al proyecto JAVA.

Debido a que Hibernate utiliza el Driver JDBC necesitará también descargar el Driver correspondiente a la base de datos a acceder desde su página oficial. Para el caso de MySQL podrá encontrar el Driver en: <https://dev.mysql.com/downloads/connector/j/>.

En el siguiente video se muestra un ejemplo de alistamiento de una aplicación Java en IDE de Eclipse para trabajar con Hibernate y una base de datos MySQL.



Teniendo en cuenta que Hibernate debe usar el Driver JDBC para la ejecución de las instrucciones en la base de datos se debe construir un archivo de configuración en formato XML donde se especifique la Url del JDBC y las credenciales de acceso a la base de datos. En la siguiente figura se presenta el esquema general de funcionamiento en el proceso de configuración:



Este archivo de configuración define un conjunto de propiedades que serán utilizadas para crear las sesiones mediante las cuales se inicia un conjunto de transacciones u operaciones sobre la base de datos, entre las propiedades más importantes se encuentran las siguientes:

| Propiedad | Definición |
| --- | --- |
| *connection.driver.class* | Parte de la configuración del JDBC donde se define el driver a usar. |
| *connection.url* | Parte de la configuración del JDBC donde se define la url de la base de datos a utilizar (protocolo: url base de datos: puerto / nombre DB) |
| *connection.username* | Parte de la configuración del JDBC donde se define el usuario de la base de datos a utilizar en la conexión. |
| *connection.password* | Parte de la configuración del JDBC donde se define el password del usuario de la base de datos a utilizar en la conexión. |
| *connection.pool\_size* | Número de conexiones simultáneas al JDBC. |
| *dialect* | Aunque SQL es un estándar, cada motor de bases de datos tiene una implementación particular. En esta propiedad se especifica el dialecto a utilizar por Hibernate. |
| *show\_sql* | Esta propiedad permite ver en la salida estándar las sentencias SQL que ejecuta en background Hibernate lo cual sirve mucho para entender lo que está sucediendo en entornos de desarrollo. |
| *current\_session\_context\_class* | Especifica el contexto de la sesión actual, por defecto se puede asignar thread. |

Es importante tener en cuenta que este archivo de configuración de Hibernate debe estar ubicado en la raíz de la carpeta src del proyecto de forma que pueda ser utilizado por Hibernate en cualquier contexto de la aplicación. A continuación, se muestra cómo debe verse un archivo de configuración de Hibernate completo donde se establece una conexión con una base de datos *MySQL* en un servidor que se está ejecutando en la máquina local en *localhost* y puerto *3306* con el usuario *root* y contraseña *12345*:

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- JDBC Database connection settings -->

<property name="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost:3306/prueba?useSSL=false&amp;serverTimezone=UTC</property>

<property name="connection.username">root</property>

<property name="connection.password">12345</property>

<!-- JDBC connection pool settings ... using built-in test pool -->

<property name="connection.pool\_size">1</property>

<!-- Select our SQL dialect -->

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Echo the SQL to stdout -->

<property name="show\_sql">true</property>

<!-- Set the current session context -->

<property name="current\_session\_context\_class">thread</property>

</session-factory>

</hibernate-configuration>

Se invita a revisar en los anexos el archivo de configuración mostrado anteriormente para que pueda ser descargado y utilizado en la construcción de su proyecto, teniendo en cuenta que debe editar las propiedades de acuerdo con sus necesidades.

Luego de esto se requiere definir todas las Entity Classes que serán utilizadas por Hibernate para realizar el proceso de mapeo hacia una base de datos relacional y viceversa. Una Entity Class básicamente es una clase Java tradicional que tiene las mismas características definidas una tabla de la base de datos, es decir, es una imagen de la tabla (ver siguiente figura) pero desde la perspectiva de la programación orientada a objetos que adicionalmente tiene unas anotaciones especiales que le permiten a Hibernate realizar el trabajo adecuado.



Las anotaciones pueden ser utilizadas al importar la librería de persistencia de Java javax.persistence.\*, entre las más utilizadas encontramos:

| Anotación | Significado |
| --- | --- |
| @Entity | Especifica que la clase definida en una Entity Class |
| @Table(name = “ ”) | Permite especificar que la clase será mapeada con una tabla en la base de datos especificada en la propiedad name. |
| @Column(name = “ ”) | Permite especificar que el atributo por ser definido será mapeado con una columna especificada por la propiedad name. Cuando el nombre del atributo coincide con el nombre de la columna en la tabla se puede omitir la especificación de la propiedad name y solo dejar la anotación @Column |
| @Id | Permite especificar que el atributo a definir corresponde con el identificador primario a ser mapeado en la tabla. |

****

A continuación, se muestra un corto video donde se agrega el archivo de configuración de Hibernate al proyecto y se crea la clase Entity para la gestión de una tabla en una base de datos MySQL.

****

* 1. **Operaciones básicas**

Una vez se tienen registradas todas dependencias, los archivos de configuración de Hibernate y las respectivas clases de entidad se puede utilizar las facilidades ofrecidas por Hibernate para la manipulación de la base de datos de forma transparente por medio de los procesos de mapeo que ofrece este framework.

Hibernate utiliza principalmente dos clases que debemos utilizar, las cuales se describen a continuación:

| Class | Descripción |
| --- | --- |
| SessionFactory | Esta clase lee el archivo de configuración de Hibernate y crea los objetos de sesión. Por ser una clase que genera objetos pesados por los procesos de los que es responsable solo debo crearlo una vez por toda la aplicación. |
| Session | Esta clase envuelve todo lo correspondiente a una conexión JDBC y es el tipo de objeto que será usado para guardar o recibir objetos desde y hacia la base de datos. Es instanciado a partir de una clase *SessionFactory*. Los objetos creados de este tipo son más livianos por lo que pueden ser creados y desechados tantas veces como se necesite dentro de la aplicación. |

En el siguiente fragmento de código se muestra un esqueleto de como es el proceso de definición de los objetos ***SessionFactory*** y ***Session*** para poder hacer la manipulación de la base de datos.

// creación del session factory

SessionFactory factory = new Configuration()

.configure("hibernate.cfg.xml")

.addAnnotatedClass(Usuario.class)

.buildSessionFactory();

// creación de la session a partir del Session factory

Session session = factory.getCurrentSession();

try {

// Ahora se puede usar el objeto session para la manipulación de la BD

}

finally {

// Se cierra el objeto session factory una vez se deja de utilizar

factory.close();

}

}

Note que en el código anterior para creación del objeto *SessionFactory* se indica el nombre del archivo de configuración de Hibernate en método *configure()*, si este parámetro no se incluyera Hibernate buscará en el directorio raíz del proyecto un archivo que contenga el nombre por defecto de este tipo de archivos de configuración el cual es *hibernate.cfg.xml*. También note que en el método *addAnnotatedClass()* se indica la clase de entidad a ser utilizada.

**Método Save**

El método *save()* del objeto Session permite hacer el proceso de inserción de un nuevo registro en la base de datos enviando como parámetro un objeto de tipo entidad con sus valores. La forma correcta de hacer un proceso de inserción de un nuevo registro de una base de datos con un objeto de tipo sesión incluye los siguientes pasos:

* Iniciar una transacción por medio del método *beginTransaction()*
* Ejecutar la lógica necesaria para crear un objeto de la clase de entidad con sus correspondientes valores.
* Invocar el método *save()* de la sesión enviando como parámetro el objeto de la clase entidad.
* Confirmar la transacción realizada por medio del método *getTransaction().commit(****)***

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

public static void main(String[] args) {

SessionFactory factory = new Configuration()

.configure("hibernate.cfg.xml")

.addAnnotatedClass(Usuario.class)

.buildSessionFactory();

Session session = factory.getCurrentSession();

try {

System.out.println("Creando un usuarios...");

Usuario usuario1 = new Usuario("Francisco","FFFFF#");

session.beginTransaction();

System.out.println("Guardando los nuevos usuarios ...");

session.save(usuario1);

System.out.println("El usuario se almaceno con el ID:"+usuario1.getUserId());

session.getTransaction().commit();

System.out.println("Proceso finalizado");

}finally {

factory.close();

}

}

}

**Método get**

El método *get()* del objeto *session* permite obtener un registro de la base de datos por medio de una consulta por su llave primaria. Este método recibe como parámetro la clase de entidad que será utilizada para hacer el mapeo del registro en la respuesta y un segundo parámetro que corresponde al valor a ser buscado en la llave primaria de la tabla. Este método ofrece como resultado un objeto de la clase de entidad especificado en la invocación del método. La forma correcta de hacer un proceso de consulta de un registro particular por medio de su llave primaria y un objeto de tipo session incluye los siguientes pasos:

* Iniciar una transacción por medio del método *beginTransaction()*
* Invocar el método *get()* de la sesión enviando como parámetro la clase de tipo entidad para hacer mapeo y el valor de la llave primaria a buscar. La invocación de este método requiere de un objeto compatible que reciba el resultado.
* Confirmar la transacción realizada por medio del método *getTransaction().commit()*

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

public static void main(String[] args) {

SessionFactory factory = new Configuration()

.configure("hibernate.cfg.xml")

.addAnnotatedClass(Usuario.class)

.buildSessionFactory();

Session session = factory.getCurrentSession();

try {

session.beginTransaction();

//consulta de un usuario cuya pk tiene el valor de 16

System.out.println("Consulta de un usuario");

Usuario resultado = session.get(Usuario.class, 16);

System.out.println("Resultado= "+resultado);

session.getTransaction().commit();

System.out.println("Proceso finalizado");

}finally {

factory.close();

}

}

}

**Método CreateQuery**

Hibernate permite el proceso de creación de consultas usando HQL, el cual es una versión especial de Hibernate para establecer consultas SQL pero es orientado a objetos y comprende nociones propias del paradigma orientado a objetos como la herencia, el polimorfismo y las asociaciones. Es importante tener en cuenta que aunque las palabras reservadas y sentencias de SQL no son sensibles a mayúsculas los nombres de las clases y sus atributos usados en HQL si lo son.

Para el caso en el que se requiera realizar consultas más avanzadas se puede utilizar la potencia de HQL para establecer las características de los registros a ser consultados, como este tipo de consulta puede producir como resultado más de un registro proveniente de la base de datos es necesario usar el método getResultList(), para poder mapear la respuesta a una lista de objetos java.

En la siguiente tabla se muestra algunas de las cláusulas HQL más comúnmente utilizadas, tengo en cuenta que en HQL se hace referencia a las clases de entidad JAVA con sus atributos los cuales son sensibles a mayúsculas y estas sentencias son mapeadas a su equivalente estructura en la base de datos usando los nombres de tablas y atributos los cuales ya no son sensibles a mayúsculas:

| Sentencia HQL | Equivalente en SQL |
| --- | --- |
| from Usuario | SELECT \* FROM usuario |
| from Usuario u where u.userName = 'Raul' | SELECT \* FROM usuario U  WHERE u.username = ‘Raul’ |
| from DomesticCat cat  order by cat.name asc | SELECT \* FROM domesticCat Cat  ORDER BY Cat.name ASC |
| from Cat as cat  inner join Cat.mate as mate | SELECT \* FROM cat cat  INNER JOIN cat\_mate mate ON Cat.id = mate.id |

Para más información sobre las características y particularidades del lenguaje de consulta de Hibernate HQL se invita a consultar en los anexos la guía oficial de este lenguaje.

A continuación, se muestran diferentes ejemplos haciendo uso de consultas por medio de HQL:

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

public static void main(String[] args) {

SessionFactory factory = new Configuration()

.configure(“hibernate.cfg.xml”)

.addAnnotatedClass(Usuario.class)

.buildSessionFactory();

Session sesion = factory.getCurrentSession();

try {

sesion.beginTransaction();

//Consultar todos los usuarios de la tabla Usuarios

List<Usuario> usuariosRegistrados = sesion.createQuery(“from Usuario”).list();

System.out.println(“\nUsuarios registrados”);

for(Usuario u: usuariosRegistrados) {

System.out.println(u);

}

//Consultar todos los usuarios cuyo username es igual a Raul

usuariosRegistrados = sesion.createQuery(“from Usuario u where u.userName = ‘Raul’”).getResultList();

System.out.println(“\nUsuarios registrados con username = Raul”);

for(Usuario u: usuariosRegistrados) {

System.out.println(u);

}

//Consultar todos los usuarios cuyo username es igual a Francisco o su password es 12345

usuariosRegistrados = sesion.createQuery(“from Usuario u where u.userName = ‘Francisco’ OR u.userPassword = ‘12345’”).getResultList();

System.out.println(“\nUsuarios registrados con userName = Francisco o userPassword = 12345”);

for(Usuario u: usuariosRegistrados) {

System.out.println(u);

}

//Consultar todos los usuarios cuyo username inicia por la letra S

usuariosRegistrados = sesion.createQuery(“from Usuario u where u.userName LIKE ‘S%’”).getResultList();

System.out.println(“\nUsuarios registrados con userName que inicia con la letra S”);

for(Usuario u: usuariosRegistrados) {

System.out.println(u);

}

sesion.getTransaction().commit();

System.out.println(“Proceso finalizado”);

}finally {

factory.close();

}

}

}

**Actualizaciones**

El proceso de actualización es muy similar al proceso de consulta ya que se puede hacer uso directo de los objetos obtenidos de las clases de entidad por medio de los métodos modificadores de sus atributos (Metodos set()) y confirmar las transacciones realizadas o también se puede hacer uso de *createQuery()* el cual permite enviar consultas HQL un poco más elaboradas pero asociándolo con el método *executeUpdate()* para que se reflejen los cambios en la base de datos una vez se confirma la transacción.

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

    public static void main(String[] args) {

        SessionFactory factory = new Configuration()

                .configure("hibernate.cfg.xml")

                .addAnnotatedClass(Usuario.class)

                .buildSessionFactory();

        Session session = factory.getCurrentSession();

        try {

sesion.beginTransaction();

            //Actualizando el usuario de llave primaria igual a 6

            Usuario consultaUsuario = session.get(Usuario.class, 6);

            consultaUsuario.setUserName("Nuevo Raul");

            System.out.println("\nUsuario  "+consultaUsuario.getUserId()+ " Actualizado!");

            session.getTransaction().commit();

            System.out.println("Proceso finalizado");

        }finally {

            factory.close();

        }

    }

}

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

    public static void main(String[] args) {

        SessionFactory factory = new Configuration()

                .configure("hibernate.cfg.xml")

                .addAnnotatedClass(Usuario.class)

                .buildSessionFactory();

        Session session = factory.getCurrentSession();

        try {

sesion.beginTransaction();

            //Actualizando el password Z@e34 a todos los usuario cuyo password actual sea 678

            System.out.println("\nActualizando varios usuarios ...");

            session.createQuery("update Usuario u set u.userPassword = 'Z@e34' where u.userPassword = '678'")

            .executeUpdate();

            session.getTransaction().commit();

            System.out.println("Proceso finalizado");

        }finally {

            factory.close();

        }

    }

}

**Método Delete**

El método *delete()* del objeto *session* permite eliminar un registro de la base de datos por medio de un objeto recibido como parámetro para identificar el registro de la tabla a eliminar. La forma correcta de hacer un proceso de eliminación de un registro particular incluye los siguientes pasos:

* Iniciar una transacción por medio del método *beginTransaction()*
* Utilizar un mecanismo para obtener una instancia de la clase entidad con el objeto que referencia del registro a eliminar ya sea por medio del método *get()* o CreateQuery.
* Invocar el método *delete()* del objeto de *session* pasando como parámetro el objeto que referencia el registro a eliminar.
* Confirmar la transacción realizada por medio del método *getTransaction().commit()*

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

    public static void main(String[] args) {

        SessionFactory factory = new Configuration()

                .configure("hibernate.cfg.xml")

                .addAnnotatedClass(Usuario.class)

                .buildSessionFactory();

        Session session = factory.getCurrentSession();

        try {

            session.beginTransaction();

            //Eliminando el usuario cuyo id es 9

            System.out.println("Eliminacion de un usuario");

            Usuario usuarioDelete = session.get(Usuario.class, 9);

            session.delete(usuarioDelete);

System.out.println("\nUsuario "+usuarioDelete.getUserName()+" eliminado");

            session.getTransaction().commit();

            System.out.println("Proceso finalizado");

        }finally {

            factory.close();

        }

    }

}

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.jonathan.hibernate.demo.entity.Usuario;

public class CreateUserDemo {

    public static void main(String[] args) {

        SessionFactory factory = new Configuration()

                .configure("hibernate.cfg.xml")

                .addAnnotatedClass(Usuario.class)

                .buildSessionFactory();

        Session session = factory.getCurrentSession();

        try {

            session.beginTransaction();

            //Eliminando todos los usuarios cuyo id es 8

            System.out.println("\nEliminando el usuario 8 ...");

            session.createQuery("delete from Usuario u where u.userId = 8").executeUpdate();

            session.getTransaction().commit();

            System.out.println("Proceso finalizado");

        }finally {

            factory.close();

        }

    }

}

A continuación, se muestran video de clases practicando cada una de las operaciones básicas necesarias para la realización de un CRUD en una tabla de la base de datos.

****

****

****

* 1. **Asociaciones**

Normalmente al desarrollar una aplicación que gestione una base de datos, se requiere mucho más que una única tabla para hacer la gestión de la información en la base de datos, por lo que es común tener modelos de datos con múltiples tablas relacionadas con asociaciones de diferentes tipos y diferentes cardinalidades. Por lo cual es importante establecer en las clases de entidad este tipo de relaciones para que el proceso de mapeo sea coherente. Los tipos de asociaciones más comúnmente utilizadas son las que expresan relaciones uno a uno, relaciones uno a muchos, relaciones mucho a uno y relaciones muchos a muchos.

Para mostrar cada uno de estos casos utilizaremos como referencia el siguiente modelo de datos que ya fue mapeado de forma individualizada y a medida que se agrega una asociación de diferente tipo se establecerán los cambios que debería estar presentes, y que se presenta en la siguiente figura:

****

**Relaciones OneToOne**

En este tipo de relaciones queremos expresar un vínculo en el que un ejemplar de una tabla tiene relación con un y solo un ejemplar de una segunda tabla. Desde el punto de vista de las bases de datos se creará una relación de integridad referencial vinculando una llave primaria con una llave foránea. Tomando como consideración el modelo de datos presentado anteriormente supongamos que se desea indicar que un usuario tiene asociado un recurso. Para resolver este nuevo requerimiento debemos modificar la clase de entidad de usuario para agregar un nuevo atributo que haga referencia al objeto recurso con el cual está relacionado y agregar la anotación correspondiente llamada @OneToOne de forma que Hibernate haga el mapeo correcto. El esquema se presenta a continuación en la siguiente figura:

****

Mientras que en la base de datos se debe agregar un nuevo campo foráneo en la tabla usuario que haga referencia a la llave primaria de la tabla recurso para poder establecer la relación, en las clases de tipo entidad bastará con especificar el nuevo atributo del tipo de objeto correspondiente que referencia agregando de la anotación de tipo @OneToOne. Si el campo nuevo requerido en la tabla usuario no existe el mapeador de Hibernate se encargará de agregarlo.

**Relaciones OneToMany - ManyToOne**

Las relaciones OneToMany tiene la particularidad de afectar cada una de las entidades vinculadas en la relación y es importante identificar el lugar que es una relación OneToMany y en qué lugar la relación es ManyToOne.

Recordando las reglas de transformación del modelo lógico de una base de datos relacional en este tipo de relaciones se indica que en la tabla del lado de la relación donde la cardinalidad máxima muchos (\*) deberá ser transferido como un campo foráneo la llave primaria de la tabla donde la cardinalidad máxima es 1, como se aprecia en la siguiente figura:

****

Para el caso de las clases de entidad deberá especificarse en la clase donde la cardinalidad máxima es 1, un atributo del tipo lista de la clase de entidad con la que tiene un vínculo *@OneToMany*, este atributo será de tipo lista porque se está vinculando con varios. En el ejemplo, la entidad usuario se relaciona con varios recursos, es decir un usuario puede estar vinculado con múltiples recursos. Por otra parte, la clase de entidad donde la cardinalidad máxima es muchos deberá especificar un atributo de tipo objeto con la entidad vinculada agregándole la anotación *@ManyToOne*. En el ejemplo un recurso estará siempre vinculado con un único usuario. Es importante anexar en la entidad de cardinalidad @OneToMany el nombre del atributo en la entidad @ManyToOne que será usada para el mapeo de la asociación por parte de Hibernate. En el ejemplo se le indica en la entidad Usuario que la relación @OneToMany será mapeada usando el atributo user de la entidad @ManyToOne.

**Relaciones ManyToMany**

Las relaciones ManyToMany obligan a la construcción de una nueva entidad de mapeo en la cual se alojarán las respectivas llaves primarias de las entidades vinculadas por medio de la relación @ManyToMany. Para su correcta implementación deberán agregarse en cada una de las clases entidad un objeto de tipo lista de la entidad a la cual está relacionada etiquetada con la anotación @ManyToMany. Adicionalmente en una de las clases de entidad deberá especificar el objeto de mapeo del otro extremo que será utilizado por Hibernate para realizar la asociación.

En el ejemplo siguiente, se establece una relación ManyToMany entre la clase Usuario y la clase Recurso por lo cual en cada una de ellas se agrega la correspondiente lista del tipo de entidad a la cual se asocian con la anotación @ManyToMany. En el caso de la clase Usuario se agrega el atributo de tipo lista de Recursos llamado recursos para indicar que un usuario puede estar vinculado con muchos recursos y por el lado de la clase Recursos se agrega una lista de tipo Usuario para indicar que un recurso puede estar vinculado con múltiples usuarios. Adicionalmente la relación entre usuario y recurso se mapea con el objeto users presente en la clase Recurso. La relación entre recurso y usuario no se mapea explícitamente por lo que generará la tabla intermedia recurso\_usuario, lo cual cumple las reglas de transformación de un modelo lógico para una relación de muchos a muchos.

****

* 1. **Ciclo de vida**

Los objetos manipulados por Hibernate para el proceso de gestión de la persistencia pasa por cuatro estados diferentes, a continuación, se describe cada uno de ellos:

Estado transitorio: corresponde al primer estado de un objeto de entidad y sucede cuando instanciamos y manipulamos el objeto de tipo entidad sin conectarlo por medio de la sesión a un registro de la base de datos. Así cualquier manipulación de este objeto no afecta la base de datos. En este estado un objeto de tipo entidad se comporta como cualquier otro objeto JAVA.

Estado persistente: corresponde al estado que logra el objeto una vez se conecta con una sesión de Hibernate, lo cual ocurre cuando se invoca por ejemplo una operación de tipo *save()* o cuando el objeto fue producto de realizar un proceso de consulta a la base de datos por un método *get()*. Todos los cambios que se realizan sobre un objeto en estado persistente son observados por Hibernate y podrán eventualmente realizar modificaciones en la base de datos.

Estado independiente: a este estado pasan los objetos en estado persistentes una vez se realiza un cierre de sesión y en este momento son dejados de observar por Hibernate y cualquier cambio no será propagado en la base de datos. Un objeto independiente puede volver a ser persistente una vez se vuelva a abrir una nueva sesión y se use uno de los siguientes métodos *save()* o *merge()*.

Estado eliminado: es el último estado en el ciclo de vida de un objeto de tipo entidad y ocurre cuando el objeto en cuestión es eliminado de la base de datos por medio del método *delete()*. Cualquier cambio realizado sobre este objeto no afectará la base de datos.

A continuación, se muestra un resumen de los estados del ciclo de vida y sus transiciones completas con todos los posibles métodos.

****

1. **Spring**

Es un Framework de código abierto que facilita crear aplicaciones empresariales de alto rendimiento en Java, además de Java tiene soporte para los lenguajes Groovy y Kotlin. La estructura que lo compone es modular esta estructura cuenta con una flexibilidad para implementar una variedad de arquitecturas dependiendo de los requerimientos de las aplicaciones haciendo que la programación sea lo más hacedera posible, rápida y muy segura.

**Figura 1**

*Figura logo Spring*

**Icono

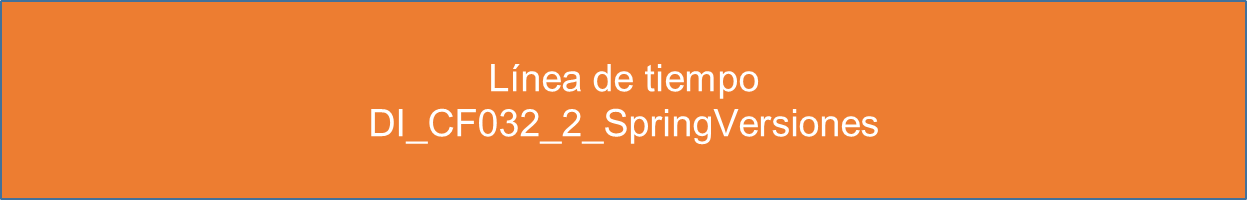
Descripción generada automáticamente**

Nota. (https://spring.io/)

Teniendo en cuenta Johnson,( 2004):

Historia: Rod Johnson creador de Spring se da cuenta que necesitan una herramienta para simplificar y facilitar la construcción de aplicaciones JEE utilizando EJB y escribe el libro expert one-on-one J2EE Development que es un conjunto de buenas prácticas para la creación de aplicaciones empresariales y en el año 2004 lanza primera versión 1.2.6 de Spring, según el autor era el inicio de J2EE tradicional

La siguiente figura muestra las Versiones de Spring Framework



En la página oficial de Spring <https://spring.io> /, se puede ver que actualmente el framework Spring se encuentra en la versión 5.

Unas de las características principales de Spring son:

Flexibilidad: contiene un grupo flexible de librerías y extensiones de terceros de Spring facilitando a los desarrolladores construir aplicaciones fácilmente.

Inyección de dependencia (DI): ofrece un patrón que permite ayudar en el acoplamiento bajo, esto quiere decir que cuando existe dependencias entre las clases y se necesita crear un objeto de esta clase lo que hace spring es suministrar este objeto obteniendo beneficios en la modularidad, escalabilidad y dependencia de clases.

POJOs (Plan Old Java Objects): facilita un sencillo desarrollo por medio de los POJOs.

ORM (Object Relational Mapping): suministra un fácil acceso a datos por medio del ORM.

Estructura y Módulos: Spring integra una variedad de módulos que se pueden integrar para el desarrollo de las aplicaciones.

La siguiente figura nos muestra lo que se puede crear con Spring Framework: Aplicaciones de consola, Aplicaciones de escritorio, Aplicaciones Web, Aplicaciones Móviles, Servicios SOAP, REST. Spring está orientado esencialmente al backend.



* 1. **Introducción**

El Framework Spring básicamente con su inyección de dependencias se encarga de construir objetos y configurar todas las dependencias que estos necesiten. En Spring estos objetos que son creados son llamados beans que son anotaciones que son agregadas a las clases así: @component, @Service, @Repository etc. Uno de los principios principales de Spring es el de Inversión de Dependencia (DIP) el cual habla de que los módulos de alto nivel no deben depender de los de bajo nivel, lo que quiere decir es que hay que hacer más referencia a las abstracciones que a las implementaciones por lo que no se debe depender de una clase ni de una implementación en particular. La implementación de este principio es llamado Inversión de Control (IOc) que se encarga de desacoplar el código utilizando eventos, también es llamado como el principio de Hollywood “no nos llames; nosotros te llamaremos”, que consisten en una manera de programar donde la aplicación es quien interviene en cambio del código, para realizar la implementación del IOc se necesita un agente externo que recibe el nombre de contenedor quién es el encargado de verificar el flujo de la aplicación. Los IOc utilizan Service locator, Eventos y los DI (Inyecciones de dependencias) enfocando el Spring Framework en la DI.

Inyección de Dependencias: es un patrón de diseño de software, subtipo de inversión de control cuyo objetivo es tener un código fácil de mantener o desacoplado. Su función es determinar lo que los objetos necesitan.

Existen 3 formas de realizar inyección de dependencias: Constructor, Propiedades(set) Servicio/Interfaz.

La siguiente figura muestra una descripción usual de la inyección de dependencia:



A continuación, se muestra un ejemplo de código donde los objetos dependen entre sí:

Acoplamiento hermético

1. public class TiendaCafe {
2. private Cafe producto;
3. public TiendaCafe() {
4. producto = new CafeOscuro ();
5. }
6. }

En el ejemplo anterior se muestra un código de una tienda de café que solicita cambiar el tipo de café y como se puede ver se realiza el cambio de café dentro de la propia clase controlando esta sus propias dependencias.

La intención es obtener un acoplamiento suelto o desacoplamiento donde no exista una dependencia entre los objetos. En el siguiente ejemplo se puede ver cómo se realiza:

1. public class TiendaCafe {
2. private Cafe producto;
3. public TiendaCafe (Cafe producto) {
4. this.producto = producto;
5. }
6. }

En el código anterior se puede observar que se obtuvo el objeto desde el exterior y que el objeto puede ser cualquier tipo de café, sin necesidad de tocar la clase TiendaCafe.

Resumiendo, la inversión de control es un patrón de programación para obtener un acoplamiento flexible logrando que los objetos independientes no sean responsables de sus dependencias si no que sean recibidas desde afuera; el contenedor de IOc recoge los metadatos que vienen de un archivo XML, o código Java, y recibe las indicaciones de los objetos que debe crearlos, enlazarlos, configurarlos y destruirlos por medio de POJOs (Objetos simples de Java) creando objetos por medio del proceso que recibe el nombre de Spring Beans y resultando objetos preparados para la inyección de dependencia.

La siguiente figura evidencia el funcionamiento de una vista abstracta de Spring.



En el video se explican aspectos generales de la inyección de dependencias:



Módulos del Framework: Spring Framework es un marco de trabajo muy amplio y brinda cerca de 20 módulo los cuales se asocian en el contenedor principal, “Accesos e integración de datos, Web, AOP (Programación orientada a aspectos, Instrumentación, Mensajes y Pruebas)” (Spring ,2021)

Para trabajar con Spring se necesita tener instalado los siguientes requerimientos:

* Un entorno de desarrollo (IDE): se puede seleccionar cualquiera de los relacionados en la tabla No 1
* Un JDK (Kit de desarrollo de java) se puede descargar cualquiera de los relacionados en la tabla No 2

**Tabla 1**

*Entornos para desarrollo en Spring*

| Entornos de Desarrollo | Enlace |
| --- | --- |
| [IntelliJ IDEA](https://www.jetbrains.com/idea/) | <https://www.jetbrains.com/idea/> |
| [Spring Tools](https://spring.io/tools), | <https://spring.io/tools> |
| [Visual Studio Code](https://code.visualstudio.com/docs/languages/java) | <https://code.visualstudio.com/docs/languages/java> |
| [Eclipse](https://www.eclipse.org/downloads/packages/) | <https://www.eclipse.org/downloads/packages/> |
| NetBeans | <https://netbeans.apache.org/> |

**Tabla 2**

*JDK para desarrollo en Spring*

| JDK | Enlace |
| --- | --- |
| JDK Oracle | <https://www.oracle.com/java/technologies/downloads/> |
| AdoptOpenJDK | <https://adoptopenjdk.net/> |

Para construir aplicaciones en Spring Framework necesitamos:

* Elegir jars con Maven: Lo primero que se debe hacer es la creación de un proyecto Maven o Gradle y realizar la descarga de las dependencias necesarias.

**Figura 2**

*Creación proyecto Maven*

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Configuración de las dependencias en el archivo pom.XML

**Figura 3**

*Configuración dependencias*

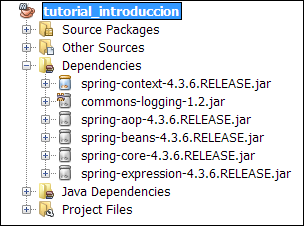
Texto

Descripción generada automáticamente

Las dependencias se deben visualizar en la carpeta Dependencias

**Figura 4**

*Dependencias*



* Creación de la Aplicación

Para el ejemplo se creará un programa con el ejemplo de HolaPlaneta.

public interface HolaPlaneta {

void enviarSaludo ();

}

public class HolaPlanetaIpml implements HolaPlaneta {

@Override

public void enviarSaludo () {

System.out.println("Hola Planeta estamos en Spring Framework");

}

* Configurar archivo XML

Una vez se tenga la aplicación se realizará la configuración en el archivo springXMLconfig

**<beans** xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd"**>**

**<bean** id="saludaService" class="sena.spring.introduccion.HelloPlanetaImpl"**/>**

**</beans>**

A continuación, se muestra un nuevo ejemplo de código explicando más a detalle el uso del Contenedor en Spring.Para el siguiente ejemplo se tiene un proyecto llamado “Sena” el cual tiene una clase aprendiz con los siguientes atributos:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | **package com.sena**    **public class AprendizBean {**  **private String primernombre;**  **private String apellidos;**  **private String cedula;**    **public String getPrimerNombre () {**  **return primernombre;**  **}**    **public void setPrimerNombre (String nombre) {**  **this.primernombre = nombre;**  **}**    **public String getApellidos () {**  **return apellidos;**  **}**    **public void setApellidos (String apellidos) {**  **this.apellido = apellidos;**  **}**    **public String getCedula () {**  **return cedula;**  **}**    **public void setCedula (String cedula) {**  **this.cedula = cedula;**  **}**    **public void verAprendiz () {**  **System.out.println(primernombre+" "+apellidos+", es la actual estudiante");**  **}**    **}** |
| --- | --- |

Al utilizar este proyecto Sena Spring framework cuenta con un archivo XML que gestiona objetos (Spring IOC Container) para este proyecto recibe el nombre de springconfig.xml y se ve de la siguiente manera.

| 1  2  3  4  5  6  7  8  9 | &amp;amp;lt;!DOCTYPE beans PUBLIC "-//PRING//DTD// BEAN 2.0//ES"  "<http://www.springframework.org/dtd/spring-beans-2.0.dtd>"&amp;amp;gt;  &amp;amp;lt;beans&amp;amp;gt;   &amp;amp;lt;bean id="aprendiz" class="com.sena.AprendizBean"&amp;amp;gt;   &amp;amp;lt;property name="primernombre" value="Zulema"&amp;amp;gt;   &amp;amp;lt;property name="apellidos" value="Leon"&amp;amp;gt;   &amp;amp;lt;property name="cedula" value="25.311.165"&amp;amp;gt;   &amp;amp;lt;/bean&amp;amp;gt;  &amp;amp;lt;/beans&amp;amp;gt; |
| --- | --- |

En la configuración del archivo se ve que la etiqueta “bean” referencia la clase AprendizBean.java, además el atributo “class” dirige a la dirección del proyecto y cuenta con un atributo id que permite agregar un nombre para poder identificarlo. Las demás etiquetas como “property” indican los atributos de la clase AprendizBean y con la etiqueta “value” se pueden agregar valores directos. Por medio de los métodos get y set el Spring Container (Contenedor de Spring) obtiene acceso y puede modificar los objetos de la clase AprendizBean. Con los archivos de configuración XML Spring facilita una clase de listas por cada uno de los beans existentes para poder hacer llamado de ellos y asi poder manipularlos desde otras clases sin tener que realizar varias instancias.

En el siguiente código se muestra la implementación de Spring del control de los objetos del proyecto Sena.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package com.sena    import org. springframework.beans.factory.BeanFactory;  import org.springframework.beans.factory.xml.XmlBeanFactory;  import org.springframework.core.io.ClassPathResource;  import org.springframework.core.io.Resource;    public class SenaLogic{      public static void main(String[] args){      Resource recurso = new ClassPathResource("springconfig.xml");        BeanFactory factory = new XmlBeanFactory(recurso);     AprendizBean aprendizBean = AprendizBean)factory.getBean("aprendiz");          aprendizBean.verAprendiz();     }  } |
| --- | --- |

Lo que mostraría el código anterior sería:

Zulema León, es la actual estudiante

La salida que muestra se logró realizando las configuraciones en el archivo XML, La utilidad del uso de Spring se evidencia en proyectos grandes con varias clases y varios objetos ya que facilita un manejo mucho más eficiente.

* 1. **Spring Data**

Spring ofrece una variedad de proyectos independientes que se pueden articular en un proyecto principal, cada proyecto permite resolver una necesidad en especial del negocio, algunos de ellos se muestran en la siguiente imagen.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Nota. <https://spring.io/projects>

Uno de los proyectos más utilizados es el SpringData ya que es uno de los módulos más grandes de Spring debido a que cuenta con muchos submódulos cuyo objetivo principal es proporcionar una serie de herramientas que permiten realizar integraciones muy sencillas con los diferentes entornos de almacenes de datos disponibles. Los datos que se utilizan pueden proceder de orígenes tanto de bases de datos relacionales como no relacionales.

Las funcionalidades de Spring Data son:

* Poderosos repositorios y uso personalizado de mapeo de objetos
* Creación de consultas dinámicas fundamentadas en los nombres que tienen los métodos.
* Plantillas de clases que brindan propiedades elementales que hacen más fácil utilizar los repositorios.
* Campos que hacen referencia a fecha de creación y modificación de las clases.
* Sencilla integración con Spring Framework por medio de la configuración de clase o archivos XML.

Módulos de Spring Data: Los módulos oficiales de Spring Data son:



Invitación a conocer todos los módulos de SpringData en el siguiente enlace: <https://spring.io/projects/spring-data>.

A continuación, se presenta un ejemplo de un mapeo para JPA que es uno de los submódulos de SpringData:

Para el ejemplo se realiza la clase Persona que representa una tabla en la base de datos llamada Persona.

Texto

Descripción generada automáticamente con confianza media

La creación de la tabla se especifica por medio de la anotación @entity y en el atributo “name” se coloca el nombre de la tabla.

Por medio de la anotación @Column se asigna cada uno de los campos de la tabla de la base de datos a los atributos de la clase Persona con la etiqueta “name” se hace referencia al nombre de la columna de la tabla.

Una vez implementada la clase Persona se crea una interfaz que extiende de la clase CrudRepository de SpringData que brinda varias funcionalidades del crud como: save (), delete (), find(),findAll() etc.Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Como se puede ver en el anterior código al extender de la clase CrudRepository nos facilita métodos como **findByNombre** que genera una consulta de todas las personas con el parámetro nombre que en base de datos relacionales seria **SELECT** \* **FROM** persona **WHERE** nombre=? como se ve reduce considerablemente las sentencias permitiendo obtener consultas dinámicas.

Otro de los grandes proyectos de Spring es Spring Boot que se describe a continuación.

* 1. **Spring Boot y MySQL**

Spring boot es una infraestructura liviana que suprime en gran parte la tarea de configurar las aplicaciones fundamentadas en Spring, Spring Boot a diferencia de Spring Framework realiza una simplificación en cuanto a configuración y el despliegue de las aplicaciones.

Las características de Spring Boot:

* Es intuitivo: permite crear aplicaciones rápidamente gracias a sus valores predeterminados.
* Permite crear aplicaciones que son independientes de Spring
* Incluye los servidores de Undertow, Tomcat o jetty.
* Usualmente realiza la configuración automática de librerías de Spring o de otros.
* No hay que realizar configuraciones de archivos XML.
* Agiliza el desarrollo.

Spring Boot cuenta con dos mecanismos que son los que permiten la simplificación de los procesos y son: Contenedor de aplicaciones integrado y Starters.

Contenedor de aplicaciones integrado: la compilación de las aplicaciones web las compila como archivo .jar desplegado en un servidor tomcat facilitando la distribución de las aplicaciones, anteriormente se trabajaba con .war que si se requiere se puede seguir utilizando.

Starters: son los encargados de brindar una variedad de dependencias las cuales se agregan al proyecto, estas pueden ser de terceros o de Spring. Los starters por defecto traen una configuración de valores por defecto.

Otra de las ventajas de Spring boot es que ofrece un mecanismo que genera la estructura del proyecto de Spring Boot automáticamente, este inicializador crea el archivo pom.xml, el main de la aplicación y un archivo de test para verificar que la aplicación corre de forma correcta.

**Figura 5**

*Inicializador de Spring*

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

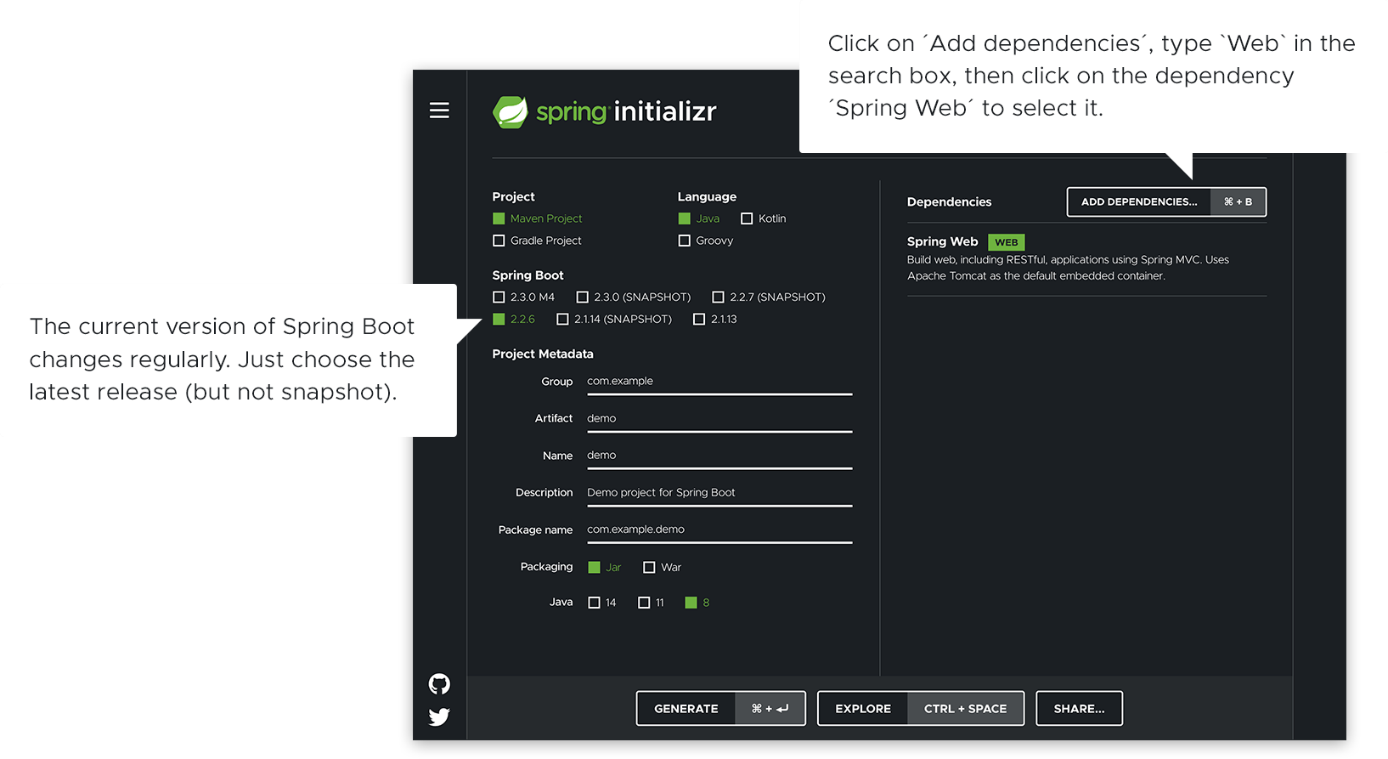
Descripción generada automáticamente

Nota. https://start.spring.io/

A continuación, se realizan los pasos para la creación de un proyecto en Spring Boot.

Recordar que se deben tener instaladas cualquiera de las herramientas de Entorno de desarrollo y JDK descritas anteriormente en la tabla 2 y tabla 3.

Paso 1: Comienza un nuevo proyecto: ingresar a  [start.spring.io](https://start.spring.io/) y crea un proyecto web.



Nota. <https://spring.io/images/quick-img-1-dark-fa196953fc04a4d6ab1133c05a622787.png>

Paso 2: Ingresa el código: después de descargar el archivo que genera el inicializador ejecute su IDE y abra el archivo, una vez ubicado en la clase principal copiar el siguiente código para ejecutar un Hola mundo.

Texto

Descripción generada automáticamente

Lo que hace el método hello() es recibir un parámetro de tipo String con el nombre de name y después concatenar este parámetro con la palabra Hello esto quiere decir que si se ingresa un nombre Zulema la salida seria “Hello Zulema”.

Paso 3: Ahora solo es probar el proyecto para esto se debe abrir una consola Windows +r, se debe ubicar en la carpeta del proyecto y ejecutar el siguiente comando:

mvnw spring-boot:run

y se mostrará la siguiente imagen:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Lo que hace es ejecutar el servidor Tomcat que ya viene por defecto dentro del Spring Boot, ahora se debe ingresar al navegador y escribir el localhost con el puerto 8080 y se mostrará la ejecución de la aplicación. En la siguiente video clase se explica de forma más detallada cómo utilizar Spring Boot:



Spring boot y MySQL: Para tener una aplicación web completa es necesario almacenar información por esta razón es importante realizar conexiones con bases de datos.

Para este ejemplo se mostrará cómo se realiza la conexión con la base de datos MYSQL.

Los requerimientos fundamentales son: Tener instalado MYSQL, el entorno de desarrollo y seguir estos pasos:

Declarar dependencias: JPA, MySQL JDBC Driver, Web como se mostró en el ejemplo anterior por medio del inicializador se pueden seleccionar las dependencias requeridas como se muestra en la siguiente imagen.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Después de agregar las dependencias clic en GENERATE y descargará un archivo zip el cuál se debe descomprimir y después importar el proyecto en el IDE para este caso Spring Tool Suite.

Una vez importado el proyecto se puede ver cómo queda la estructura.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

El paso siguiente es abrir el archivo “aplication.properties” y configurar todas las propiedades de la conexión así:

1.spring.jpa.hibernate.ddl-auto=update

2.spring.datasource.url=jdbc:mysql://**localhost:3306/demo**?useUnicode=true&characterEncoding=utf8&createDatabaseIfNotExist=true

3.spring.datasource.username=root

4.spring.datasource.password=password

5.spring.jpa.show-sql=true

A continuación, se explicará cada línea del anterior código que va en el archivo “aplication. properties”.

Línea 1: permite obtener los metadatos de la base de datos.

Línea 2: se muestra la url de la conexión con el motor de base de datos.

Línea 3. Se especifica el nombre del usuario de la base de datos para la conexión

Línea 4. Se especifica clave de la base de datos para la conexión

Línea 5: la propiedad true permite utilizar las consultas JPA a realizar en la base de datos.

Lo que queda es ejecutar la aplicación y comprobar que no existan problemas con la conexión a la base de datos.

Para ejecutar dar clic derecho en el archivo java seleccionar la opción >Run As y >Spring Boot App

Si todo está bien ahora solo queda ir al motor de bases de datos y verificar si creó el esquema.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

2.4 Validaciones:

Es muy importante realizar validaciones cuando se están desarrollando aplicaciones web en java para impedir que se ingresen datos que afecten al programa, en Sprint viene incluido el hibernate-validator en el paquete spring-boot-starter-web que permite validar valores de los campos Beans.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>2.0.0.RELEASE</version>

</dependency>

Para el ejemplo se realiza la creación de un bean con unas anotaciones que permiten la validación de ingreso de un usuario.

@Data

@NoArgsConstructor

public class Usuario{

@NotBlank(message="Debe ingresar un nombre de usuario")

private String nombreUsuario;

@NotBlank(message="debe ingresar un valor correcto para la edad")

@Pattern(regexp="^[0-9]{1,2}$",message="Edad incorrecta")

private String edad;

@AssertFalse(message = "Debe ser falso")

private Boolean esfalso;

/\*\*

\*/

@Pattern(regexp="^[0-9]{4}-[0-9]{2}-[0-9]{2}$",message="No es correcto el formato de la fecha ")

private String fechanaciemiento;

}

Una vez se crean las anotaciones se debe crear el controlador.

@Controller

public class userController{

@RequestMapping("/usuario")

public void create (@RequestBody @Validated Usuario usuario) {

System.out.println("hola:" + usuario.getNombreUsuario());

}

}

Una vez se realice el controlador se puede realizar la prueba enviando una solicitud Post.

**{"nombreUsuario":"Ana","edad":120,"esFalso”: true,"nacimiento":"21010-21-12"}**

La verificación que mostraría el resultado es:

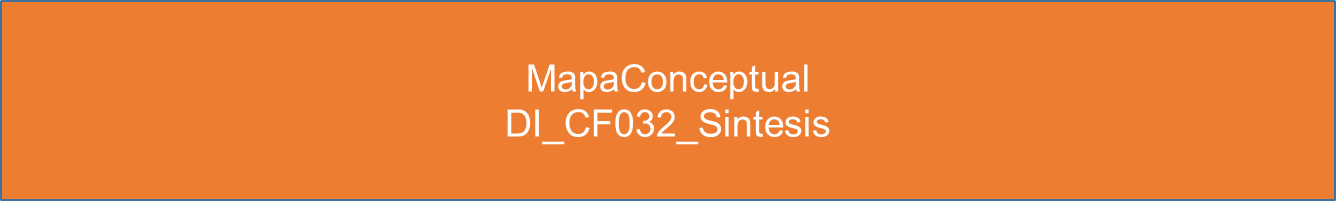
No es correcto el formato de la fecha

Debe ser falso

debe ingresar un valor correcto para la edad

1. **SÍNTESIS**

Los frameworks como Hibernate facilitan la conexión y manipulación de bases de datos relacionales ya que vienen configurados y optimizados para abstraer la complejidad propia de las bases de datos y permiten al desarrollador enfocarse en trabajar únicamente con el paradigma orientado a objetos. Adicionalmente una vez configurado el Framework facilita la realización de todo tipo de operación DML (SELECT, INSERT, UPDATE y DELETE) acelerando la velocidad en la que se pueden desarrollar aplicaciones JAVA que gestionan bases de datos relacionales. A continuación, se muestra un mapa conceptual de los elementos más importantes desarrollados en el primer apartado de este componente:

****

1. **ACTIVIDADES DIDÁCTICAS (OPCIONALES SI SON SUGERIDAS)**

| DESCRIPCIÓN DE ACTIVIDAD DIDÁCTICA | |
| --- | --- |
| Nombre de la actividad | Repaso de conceptos clave relacionados con el framework de Hibernate |
| Objetivo de la actividad | Afianzar los conceptos clave más importantes relacionados con el Framework de Hibernate. |
| Tipo de actividad sugerida | Arrastrar y soltar el concepto con la descripción que lo identifica. |
| Archivo de la actividad  (Anexo donde se describe la actividad propuesta) | Anexo/Actividad didáctica 1 |

| DESCRIPCIÓN DE ACTIVIDAD DIDÁCTICA | |
| --- | --- |
| Nombre de la actividad | Flujo de acciones Hibernate |
| Objetivo de la actividad | Identificar las generalidades del flujo de acciones para manipular una base de datos con Hibernate |
| Tipo de actividad sugerida | Arrastrar y soltar |
| Archivo de la actividad  (Anexo donde se describe la actividad propuesta) | Anexo/Actividad didáctica 2 |

**E. MATERIAL COMPLEMENTARIO**

| Tema | Referencia APA del Material | Tipo de material  (Video, capítulo de libro, artículo, otro) | Enlace del recurso o  archivo del documento o material |
| --- | --- | --- | --- |
| 1. Hibernate - HQL | Capítulo 15. HQL: El lenguaje de consulta de Hibernate. (s. f.). HIbernate Community Documentation. | Sitio web | <https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html> |

**F. GLOSARIO**

**API:** conjunto de métodos ofrecidos por una librería que puede ser utilizado por otro software por medio de un conjunto de reglas establecidas.

**Framework:** Es un marco o esquema de trabajo que define un conjunto de utilidades implementadas para un contexto particular para ser reutilizadas por los desarrolladores y así facilitar y acelerar el desarrollo de ciertas actividades propias de la construcción de software.

**Groovy**: lenguaje de programación efectuado sobre la plataforma Java, el cual permite realizar Apis de JAVA

**HQL:** lenguaje de consulta parecido al SQL implementado por el framework Hibernate.

**Kotlin:** lenguaje programación de tipado estático sencillo y fácil de entender y aprender.

**JDBC**: API para JAVA que permite el acceso y manipulación de bases de datos relacionales.

**JPA**: API de persistencia de JAVA.

**Plugin**: Programas pequeños que se anexan a otros para ofrecer alguna funcionalidad nueva.

**G. REFERENCIAS BIBLIOGRÁFICAS**

Johnson, R. (2004). *Expert one-on-one J2EE design and development*. John Wiley & Sons.

Martínez, P. E. (2015). Hibernate. Persistencia de objetos en JEE (1.a ed.). Ra-Ma.

Spring. (2021). makes Java simple. <https://spring.io>

**H. CONTROL DEL DOCUMENTO**

|  | Nombre | Cargo | Dependencia  *(Para el SENA indicar Regional y Centro de Formación)* | Fecha |
| --- | --- | --- | --- | --- |
| Autor (es) | Jonathan  Guerrero Astaiza | Experto temático | Centro de Teleinformática y Producción Industrial | Octubre 2021 |
| Zulema  León Escobar | Experta temática | Centro de Teleinformática y Producción Industrial | Octubre 2021 |
| Deivis Eduard  Ramirez Martinez | Diseñador  Instruccional | Centro para la Industria de la Comunicación Gráfica Regional Distrito capital | Diciembre 2021 |
| Silvia Milena Sequeda Cárdenas | Evaluadora instruccional | Centro de gestión industrial. Regional distrito Caìtal. | Diciembre de 2021 |
| Rafael Neftalí  Lizcano reyes | Asesor  Pedagógico | Centro Industrial de Diseño y la Manufactura | Diciembre 2021 |
|  | Sandra Patricia Hoyos Sepúlveda | Corrección de estilo | Centro de Diseño y Metrología - Distrito capital | febrero 2021 |

1. **CONTROL DE CAMBIOS**

**(Diligenciar únicamente si realiza ajustes a la Unidad Temática)**

|  | Nombre | Cargo | Dependencia | Fecha | Razón del cambio |
| --- | --- | --- | --- | --- | --- |
| Autor (es) |  |  |  |  |  |