

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

**Exploración del mapeo objeto-relacional**

**Clase persistente y cómo crearla**

**Fichero XML básico de mapeo**

**Objeto sesión**

**Extracción, guardado y modificación de datos con Hibernate**

## Clases persistentes

El concepto de un ORM es **coger** ciertos valores desde los **atributos de las clases de Java** y **persistirlos** a las tablas de una **base de datos**. Un **documento de mapeo** nos ayudará a determinar **cómo coger dichos valores** desde las clases de Java y **mapearlos a los campos** asociados en base de datos.

Los objetos de las clases persistentes en Hibernate son los que serán **almacenados** en base de datos.

Este tendrá mejor eficiencia si estas clases de las que hablamos, siguen algunas reglas simples o el **modelo de programación de objetos Java simples** (POJO Plain Old Java Object). A continuación, veremos algunas de las **reglas principales** de las clases persistentes, teniendo en cuenta que no son requerimientos puros:

- Todas las clases que van a ser persistidas necesitarán un **constructor por defecto**.
- Todas las **clases** deben contener un **atributo ID** para facilitar la identificación de los objetos tanto en Hibernate como en base de datos. Será **mapeado como primary key** en base de datos.
- Todos los **atributos** de la clase deberán ser definidos como **privados** y tener **métodos get() y set()**.
- Una característica de las clases persistentes de Hibernate suele ser que dichas **clases no** sean de tipo **"final"**.

Se usa el nombre de POJO para enfatizar que no es más que un **objeto ordinario Java** y **no es ningún objeto especial**.

Ejemplo de clase persistente:

```
public class Customer {
    private int id;
    private String firstName;
    private String lastName;
    private int customerNumber;

    /**
     * Al menos debe de haber estos dos constructores,
     * uno sin parámetros, y otro con todos los atributos.
     */
    public Customer() {
    }
    public Customer(String fname, String lname, int custNum) {
        this.firstName = fname;
        this.lastName = lname;
        this.customerNumber = custNum;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String first_name) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String last_name) {
        this.lastName = last_name;
    }
    public int getCustomerNumber() {
        return customerNumber;
    }

    public void setCustomerNumber(int customerNumber) {
        this.customerNumber = customerNumber;
    }
}
```

## Composición de un fichero de mapeo

Normalmente, las relaciones de mapeo objeto-relacional están definidas en un documento con extensión XML. Este documento guía a Hibernate, pero, claro, ¿cómo se mapean las clases previamente definidas a la base de datos?

Aunque muchos desarrolladores que usan Hibernate prefieren escribir el fichero de mapeo a mano, realmente existen diversas **herramientas** que generan dicho documento. Entre otras, **Xdoclet**, **Middlegen** y, para desarrolladores algo más avanzados, **AndroMDA**. Partiendo de la clase Java POJO de la página anterior, habría una tabla correspondiente a cada objeto que corresponde a la clase "Customer". En relación a dicho código que vimos anteriormente, para crear **en base de datos** dicha tabla, habría que ejecutar la siguiente sentencia:

**DDL** tabla Customer:

```
create table CUSTOMER (  
    id INT NOT NULL auto_increment,  
    first_name VARCHAR(20) default NULL,  
    last_name VARCHAR(20) default NULL,  
    customernumber INT default NULL,  
    PRIMARY KEY (id)  
);
```

Teniendo en cuenta la clase Java que mostramos anteriormente, a continuación, veremos el **fichero de mapeo**, el cual le indicará a Hibernate **cómo mapear dicha clase definida** a la tabla de la base de datos:

Fichero de mapeo:

```
<hibernate-mapping>  
    <class name = "Customer" table = "CUSTOMER">  
        <meta attribute = "class-description">  
            This class contains the customer info.  
        </meta>  
        <id name = "id" type = "int" column = "id">  
            <generator class="native"/>  
        </id>  
        <property name = "firstName" column = "first_name" type = "string"/>  
        <property name = "lastName" column = "last_name" type = "string"/>  
        <property name = "customerNumber" column = "customernumber" type = "int"/>  
    </class>  
</hibernate-mapping>
```

Este fichero lo guardaremos de la siguiente forma:

**"<nombre>.hbm.xml",**

por ejemplo Customer.hmb.xml.

### **Anotaciones Hibernate**

Una información interesante a destacar y que nos va a venir muy bien a la hora de enfrentarnos a nuestros desarrollos, es que por medio de las **anotaciones** Hibernate **evitaremos tener que escribir el fichero** de mapeo.

De esta forma a nuestra clase persistente le añadiríamos **anotaciones** como:

- @Entity
- @Table
- @Column
- @Id Generated Value...

Y muchas anotaciones más...

En estos casos, le estaríamos **indicando a Hibernate** directamente la **forma de mapear** nuestras clases persistentes.

## Análisis de los elementos de un fichero de mapeo

Teniendo en cuenta el fichero de mapeo que se mostró en la página anterior, veamos los diferentes elementos de mapeo que se han usado:

- El documento de mapeo es un fichero XML que tiene como **elemento principal**:

**<hibernate-mapping>**,

el cual, en su interior, **almacenará todas las clases definidas**.

- Los elementos de tipo

**<class>**,

son usados para definir mapeos especiales que van desde nuestras clases Java definidas a las tablas de base de datos. El **nombre de la clase** Java es especificado usando el **atributo name**, y la **tabla asociada** en base de datos es vinculada con el **atributo table**.

- Los elementos

**<meta>**,

son **opcionales** y pueden ser usados, por ejemplo, para definir la **descripción** de una clase.

- El elemento

**<id>**

mapea el **atributo ID** único en la parte de la clase Java y lo transforma en **Primary Key** en la **tabla** de la base de datos. El **atributo name** hace referencia al atributo de la clase Java y **column** se refiere a la columna real que hay en la tabla de base de datos. El **atributo type** proporciona a Hibernate la **tipología del objeto** y será convertido desde tipología Java a SQL.

- El elemento

**<generator>**,

junto con el elemento id, es usado para **generar la clave primaria** automáticamente. El atributo **class** del elemento generator se establece con el valor **native** para permitir a Hibernate **crear la Primary Key** con diferentes **algoritmos**: **identity**, **hilo** o **sequence**, dependiendo de las capacidades de la base de datos.

- El elemento

**<property>**

se usa para **mapear los atributos** o propiedades de **Java** y transformarlos en **columnas** de la **tabla** asociada en la base de datos. El atributo **name** del elemento hace referencia al nombre del atributo en la **clase Java** y **column** se refiere a la columna que existe en **base de datos**. El atributo **type** proporciona y transforma la tipología del **objeto Java** a objeto de **tipo SQL**.

## Ejemplo de Clase persistente

La clase “Vehículo” tiene los siguientes campos:

- Marca
- Motor
- Número de ruedas
- Kilómetros

Se requiere realizar una **clase persistente** con sus atributos, constructor por defecto y métodos getter y setter.

Realizaremos una clase persistente teniendo en cuenta los cuatro campos que se requieren y uno más que será el id, numérico y autoincrementable para base de datos.

Definiremos los cinco **atributos como privados**, añadiremos dos **constructores base**: un **constructor por defecto** y **otro con todos los atributos**, y por último, añadiremos los **métodos `get()` y `set()`**.

A continuación, vemos el código:

```
public class Vehiculo {
    private int id;
    private String marca;
    private String motor;
    private int numeroRuedas;
    private int numeroKilometros;

    public Vehiculo() {
    }

    public Vehiculo(String marca, String motor, int numeroRuedas, int
numeroKilometros) {
        this.marca = marca;
        this.motor = motor;
        this.numeroRuedas = numeroRuedas;
        this.numeroKilometros = numeroKilometros;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
```

```
        this.id = id;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getMotor() {
        return motor;
    }

    public void setMotor(String motor) {
        this.motor = motor;
    }

    public int getNumeroRuedas() {
        return numeroRuedas;
    }

    public void setNumeroRuedas(int numeroRuedas) {
        this.numeroRuedas = numeroRuedas;
    }

    public int getNumeroKilometros() {
        return numeroKilometros;
    }

    public void setNumeroKilometros(int numeroKilometros) {
        this.numeroKilometros = numeroKilometros;
    }
}
```



## Sesiones y objetos Hibernate

### Estados

Un **objeto de sesión** es usado para establecer una **conexión física** con una base de datos. Este objeto es **ligero** y diseñado para ser instanciado cada vez que se necesite una **interacción con la base de datos**. Los **objetos persistentes** son **almacenados y devueltos** a través del objeto de sesión (Session object).

El objeto de sesión no debe mantenerse abierto durante mucho tiempo, ya que podría ser peligroso por temas de **seguridad**. Deben ser **creados y destruidos** cada vez que sea necesario **utilizarlos**.

La función principal de estos objetos es **ofrecer, crear, leer y borrar operaciones** para las **instancias** de las **clases mapeadas**. Dichas instancias pueden estar en uno de los siguientes **estados**:

- **Transient (transitorio)**: Nueva **instancia de una clase persistente**, **no** está aún asociada a un **objeto de sesión** y no tiene **representación** en la base de datos ni **identificador** según Hibernate.
- **Persistent**: Una **instancia transient** se puede hacer **persistente** asociándola con una **sesión**. Una instancia persistente tiene una **representación** en la base de datos, un **identificador**, y la asociación con el objeto de sesión.
- **Detached (separada)**: Una vez se **cierra la sesión** de Hibernate, la instancia persistente se convertirá en una **instancia separada**.

Una instancia de sesión es serializable si sus clases persistentes lo son.

Una transacción puede tener el siguiente aspecto:

Ejemplo de uso de session Hibernate:

```
SessionFactory sessionFactory = null;
Session session = sessionFactory.openSession();
Transaction transaction = null;

try {
    transaction = session.beginTransaction();
    // Aquí realizaríamos operaciones
    transaction.commit();
}
/**
 * Si la sesión lanzara una excepción, se debería de hacer
 * roll-back de la transacción y la sesión descartada.
 */
catch (Exception e) {
    if (transaction != null) {
        transaction.rollback();
        e.printStackTrace();
    }
} finally {
    session.close();
}
```

## Métodos de la interfaz Session de Hibernate

Existen numerosos métodos en la interfaz Session, a continuación, veremos los más importantes y sus correspondientes funciones. Por supuesto en la página oficial de Hibernate <https://hibernate.org/> podremos estudiar el resto:

- **beginTransaction()**: Básicamente, permite comenzar o iniciar una **unidad de trabajo** y devolver el **objeto** asociado de la **transacción**.
- **cancelQuery()**: Cancela la ejecución de la consulta actual.
- **Clear()**: **Elimina** completamente la **sesión**.
- **Close()**: **Finaliza la sesión** liberando la conexión JDBC y limpiándola. **Devuelve** un objeto de tipo **Connection**.
- **createCriteria(Class clasePersistente)**: Crea una instancia nueva de Criteria **para** la **clase persistente** proporcionada como parámetro. **Devuelve** un objeto de tipo **Criteria**.
- **createCriteria(String entityName)**: Crea una instancia de tipo Criteria **para** la **entidad** que se le pasa como parámetro. **Devuelve** un objeto de tipo **Criteria**.
- **getIdentifier(Object objeto)**: **Devuelve** un objeto de tipo **Serializable** que es el **identificador** de la entidad proporcionada y asociada a esta sesión.
- **createFilter(Object colección, String consulta)**: Crea una nueva **instancia de consulta** en función a la colección pasada como parámetro y la consulta. **Devuelve** un objeto de tipo **Query**.
- **createQuery(String consultaHQL)**: Crea una **instancia de consulta** en función a la sentencia HQL que se le pasa como parámetro. **Devuelve** un objeto de tipo **SQLQuery**.
- **delete(Object objeto)**: **Borra** una **instancia persistente** del almacén de datos.
- **getSessionFactory()**: **Devuelve** un objeto de tipo **SessionFactory**, el cual creó la **sesión actual**.
- **refresh(Object objeto)**: Vuelve a **leer el estado de la instancia** dada como objeto proveniente de la base de datos.
- **isConnected()**: Comprueba si la **sesión** está **conectada** actualmente.
- **isOpen()**: Comprueba si la **sesión** aún está **abierta**.

## Carga, almacenamiento y modificaciones de objetos

En este apartado, veremos cómo podemos **recuperar objetos** de nuestra **base de datos** usando Hibernate, y también las diferentes formas para actualizarlos, guardarlos, etc. Para ello, disponemos de una serie de comandos que ejecutaremos **desde nuestro objeto Session**:

- **Carga de objetos:** Como su nombre indica, con los siguientes comandos obtendremos **datos** de nuestra base de datos. Podremos usar:

- Session.**get()**: Devuelve un **objeto persistente** según los parámetros de objeto de entidad e identificador, y **si no existe** objeto en base de datos, devolverá **null**.
- Session.**load()**: También devuelve un **objeto persistente** teniendo en cuenta los parámetros que se le pasan de entidad e identificador. Devolverá un **ObjectNotFoundException** en caso de **no encontrar** dicha entidad.

- **Almacenamiento de objetos:** Para **guardar la información** en base de datos desde Hibernate usaremos:

- Session.**persist()**: Ejecuta el comando **insert** del lenguaje **SQL** almacenando **filas** en la base de datos. Este método es de tipo **void()**, **no devuelve** nada.
- Session.**save()**: Igual que el método anterior, ejecuta internamente un método insert de SQL con la diferencia de que **devuelve** un objeto de tipo **Serializable**.

- **Modificación de objetos:** Con los siguientes comandos **actualizaremos** los diferentes objetos en base de datos.

- Session.**update()**: Realizaremos un update en base de datos. Es el método primitivo para actualizar filas y **necesita** que haya **otra instancia de session** ejecutándose, y **si no**, lanzará una **excepción**.
- Session.**merge()**: Se ejecutará un update también en base de datos, pero, en este caso, **no** tendremos que preocuparnos si existe ya una **instancia** ejecutándose, ya que este método **realizará la operación gestionando el resto**.

- **Otros métodos:**

- Session.**delete()**: Pasaremos como parámetros la **entidad persistente** y se realizará el **borrado en base de datos**.
- Session.**saveOrUpdate()**: Método de gran utilidad para permitir tanto la **actualización** de la entidad (**si existe** en base de datos) como el **insert** (**si no existe** en base de datos).

## Consultas SQL y HQL

El lenguaje Hibernate Query (HQL) es un lenguaje orientado a objetos muy parecido a SQL, pero en lugar de operar con tablas y columnas, **HQL trabaja con objetos persistentes** y con las **propiedades** de los mismos. Las consultas de tipo HQL son **traducidas** por Hibernate en **consultas corrientes SQL**, que más tarde son **ejecutadas en base de datos**.

En Hibernate, aunque **podemos usar sentencias SQL directamente** usando SQL nativas, se recomienda **usar HQL** para **evitar problemas de portabilidad** de la base de datos y aprovechar también las **estrategias de caché implementadas** en Hibernate.

### Ejemplo de consulta HQL:

```
String hql = "FROM Customer";
Query consulta = session.createQuery(hql);
consulta.setFirstResult(1);
consulta.setMaxResults(40);
List results = (List) consulta.list();
```

Este es un ejemplo muy sencillo de una consulta HQL donde consultamos la tabla “Customer”, con la que obtenemos todos los registros, y más tarde, con “**setFirstResult**” y “**setMaxResults**”, aplicamos un filtro de cuantas filas queramos obtener. En este caso, las **40 primeras**. En la **documentación oficial de Hibernate**, podremos encontrar cómo realizar el resto de sentencias como **insert, delete, update, etc.**

También, **podemos emplear sentencias de SQL** nativas si queremos usar funcionalidades específicas de la base de datos que tenemos conectada a nuestra aplicación. Un ejemplo de ello puede ser la **ejecución de procedimientos** almacenados en base de datos, que son **fragmentos de código o pequeñas aplicaciones** desarrolladas en función al lenguaje de dicha base de datos. Se ejecutan dentro del motor de la base de datos relacional. Con estos procedimientos, **nos ahorramos la recepción y el envío de datos al usuario**, pudiendo suprimir la parte de la recepción realizando un pequeño **informe** a dicho usuario con las operaciones realizadas o con un **reporte de resultados**.

Continuando con el tema de las **sentencias nativas SQL**, podremos ejecutarlas desde nuestro código con el comando:

```
createSQLQuery()
```

Este método recibirá como parámetro una **variable de tipo String** que contendrá la **consulta nativa SQL**.

**Devuelve** un objeto de tipo **SQLQuery** y una vez obtenido dicho objeto, se puede **asociar** perfectamente a una **entidad existente de Hibernate**.

### Ejemplo de actualización de una tabla con HQL:

```
public void ejemploUpdateHQLCustomer(int id, int customerNumber, String
firstName) {
    session = sessionFactory.openSession();
    transaction = session.beginTransaction();
    Query consulta = session.createQuery(
        "update Customer " +
        "set CUSTOMERNUMBER=:customerNumber " +
        "where ID=:id and FIRSTNAME=:firstName");
    consulta.setParameter("customerNumber", 2);
    consulta.setParameter("id", 27);
    consulta.setParameter("firstName", "Marcos");

    int status = consulta.executeUpdate();
    transaction.commit();
    // Si devuelve positivo habrá logrado con éxito el Query
    if (status > 0)
        System.out.println("Update realizado");
    else
        System.out.println("Update no realizado");
}
```

## Gestión de transacciones con Hibernate

Una transacción representa una **unidad de trabajo**. De tal forma que si **uno de los pasos falla**, la **transacción completa fallará** (relacionado con el concepto de **atomicidad**). Una transacción, como hemos visto en temas anteriores, se define por las **características ACID**:

- **Atomicidad**,
- **Consistencia**,
- **Isolation** (Aislamiento),
- **Durabilidad**.

En Hibernate, tenemos la interfaz Transaction que define la unidad de trabajo. Mantiene la abstracción desde su propia implementación.

Algunos de los **métodos** principales en la **interfaz Transaction** son:

- Void **begin**: Empieza una **transacción nueva**.
- Void **commit()**: **Finaliza** la unidad de trabajo **a menos que** estemos en el modo **"FlushMode.NEVER"**.
- Void **rollback()**: Fuerza a **cancelar** totalmente la transacción.
- Void **setTimeout(int segundos)**: Establece un **time out determinado** por parámetro en segundos.
- Boolean **isAlive()**: Comprueba si la transacción está aún **activa**.
- Void **registerSynchronization(Synchronization s)**: Registra la **respuesta sincronizada de un usuario** para esa transacción.
- Boolean **wasCommitted()**: Comprueba si se ha **cerrado** la transacción **satisfactoriamente**.
- Boolean **wasRolledBack()**: Comprueba si la transacción ha sido **cancelada satisfactoriamente**.

### Otro ejemplo de Sentencia HQL:

Tenemos una clase denominada "Customer", cuyos campos son:

```
private int id;

private String firstName;

private String lastName;

private int customerNumber;
```

Realizaremos una sentencia HQL de tipo **update** por medio de **parámetros**, en la entidad Customer y en la **columna**, "**customerNumber**", para **establecer** en este campo el **número 2** como número de cliente, siempre y cuando su "**Id**" sea **27** y cuando su "**firstName**" sea "**Marcos**".

```
// instanciamos nuestro objeto session
Session session = sessionFactory.openSession();
// iniciamos una transacción
Transaction transaction = session.beginTransaction();
// con el propio objeto session y su método createQuery, asignamos
// a una variable de tipo Query la consulta HQL que vamos a
// preparar.
Query consulta = session.createQuery(
// vamos a persistir sobre la tabla "Customer"
    "update Customer " +
// disponemos de tres parámetros:
// con el primero 'seteamos' el valor a la columna
// "CUSTOMERNUMBER"
    "set CUSTOMERNUMBER=:customerNumber " +
// el segundo y el tercero entran dentro de la cláusula WHERE,
// donde filtramos por id para que se haga coincidir con el 105, y
// "firstName", para que sea "Pepe".
    "where ID=:id and FIRSTNAME=:firstName");
consulta.setParameter("customerNumber", 2);
consulta.setParameter("id", 27);
consulta.setParameter("firstName", "Marcos");
// Se ejecuta la query y guardamos el valor resultante en una
// variable de tipo entero, si es 1, quiere decir que ha sido
// persistida una tabla, y, por tanto, es correcto.
int status = consulta.executeUpdate();
transaction.commit();
if (status > 0)
    System.out.println("Update realizado");
else
    System.out.println("Update no realizado");
```



## Buenas prácticas en la creación de clases persistentes

### 1. Uso de **Anotaciones de Hibernate**.

Utilizar anotaciones específicas de Hibernate, como:

- @Entity,
- @Id,
- @GeneratedValue,
- @Column...

**paramapear las clases** y propiedades a la base de datos de manera declarativa.

### 2. **Identificador Único (@Id)**.

Marcar una propiedad de la clase con la anotación @Id para indicar que es la **clave primaria** de la entidad.

### 3. Generación de **Identificadores (@GeneratedValue)**.

Usar @GeneratedValue para especificar cómo se **generarán automáticamente los valores de las claves primarias** (por ejemplo, mediante una estrategia de **incremento automático**).

### 4. **Relaciones entre Entidades**.

Definir relaciones entre entidades usando anotaciones como

- @ManyToOne,
- @OneToMany,
- @OneToOne,
- @ManyToMany,

según la naturaleza de la relación.

### 5. **Mapeo de Columnas (@Column)**.

Especificar detalles de mapeo, como el nombre de la columna, la longitud, la precisión, etc., utilizando la anotación @Column.

### 6. **Cascada de Operaciones (cascade)**.

Usar la opción cascade para especificar **cómo** las operaciones (como guardar, actualizar o eliminar) en una entidad **afectarán a las entidades relacionadas**.

### 7. **Fetch Type (fetch)**.

Configurar el tipo de recuperación (fetch type) utilizando la anotación

- @ManyToOne(fetch=FetchType.**LAZY**), (perezoso)
- @OneToMany(fetch=FetchType.**EAGER**), (ansioso)

según los requisitos de rendimiento y la naturaleza de la relación.

#### 8. Mapeo de Herencia (@Inheritance).

Manejar la herencia en las clases persistentes utilizando anotaciones como:

- @Inheritance,
- @DiscriminatorColumn,

para establecer la **estrategia de mapeo de herencia**.

#### 9. Implementación de equals() y hashCode().

Sobrescribir los métodos equals() y hashCode() correctamente para **garantizar la consistencia y la comparación adecuada** de objetos en colecciones.

#### 10. Implementación de toString().

Proporcionar una implementación significativa del método toString() para facilitar la **depuración y la representación legible** de las instancias de la clase.

#### 11. Manejo de Transacciones.

Asegurar que las operaciones de **persistencia** se realicen **dentro de una transacción**, ya sea mediante programación o mediante el uso de **anotaciones @Transactional**.

#### 12. Optimización nombradas de Consultas (@NamedQuery).

Utilizar consultas nombradas (@NamedQuery) para optimizar y reutilizar **consultas frecuentes**.

#### 13. Validación de Datos (@NotNull, @Size, etc.).

Aplicar anotaciones de validación, como @NotNull o @Size, para **garantizar la integridad** de los datos antes de ser persistidos.

## Ejemplo de mapeo de la clase PERSONAS.

Dicha clase contiene cuatro atributos: Id, nombre, edad y altura.

Debemos de conocer tanto los **atributos de la clase persistente** como de los **nombres reales de las columnas** en base de datos, así como los tipos definidos en ella.

Mapeo de la clase Personas:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<hibernate-mapping>
  <class name="Personas" table="PERSONAS">
    <meta attribute="class-description">
      Esta clase contiene características de un humano.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native" />
    </id>
    <property name="nombre" column="name" type="string" />
    <property name="edad" column="age" type="int" />
    <property name="altura" column="height" type="int" />
  </class>
</hibernate-mapping>
```

Con este fragmento de código sería suficiente para mapear la **entidad Personas** a la **tabla PERSONAS** de la base de datos.

Algunos puntos a señalar son:

- Dentro de la etiqueta **<meta>**, hemos usado el atributo **"class description"** para escribir una breve **descripción de la entidad**.
- Hemos tenido **especial cuidado** de colocar **bien escritos** tanto los **atributos** de la clase persistente como las **columnas** de base de datos y sus **tipos**.

<https://hibernate.org/>

<https://www.oracle.com/es/java/>

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

## **Bases de datos orientadas a objetos**

## Definición de las bases de datos orientadas a objetos

Relacionamos el concepto de bases de datos orientadas a objetos, cuando las técnicas de bases de datos se combinan con objetos. El resultado de esto es un **sistema gestor orientado a objetos: ODBMS (Object Data Base Management System)**. Hoy en día, realmente existe un aumento del desarrollo de aplicaciones orientadas objetos, facilitando, por tanto, la evolución de los sistemas gestores de base de datos orientados a objetos (OODBMS). Esto proporciona un gran valor para los desarrolladores que trabajan con este tipo de aplicaciones. Es evidente que estos profesionales tendrán cierta facilidad para realizar el desarrollo de la **aplicación orientada a objetos**, almacenarla en una base **de datos realizada** en el mismo paradigma, y **replicar o modificar objetos** existentes para crear nuevos **objetos dentro de este sistema** OODMBS.

El modelo de datos orientado a objetos (ODDM) son modelos lógicos que capturan la semántica de los objetos de la aplicación que va unida o relacionada. Realmente, son modelos que van en consonancia con el modelo de la aplicación. Los OODMs **implementan modelos conceptuales** directamente y pueden representar **complejidades** que van **más allá** de los sistemas de **bases de datos relacionales**. Por este motivo, han heredado muchos de los conceptos que fueron pensados e implementados en los lenguajes de programación orientados a objetos.

Realmente, una **base de datos orientada a objetos** no es más que:

→ una **colección de objetos** definidos por un **modelo orientado a objetos**.

Este tipo de bases de datos pueden **extender la existencia de los objetos** para que se **almacenen indefinidamente**.

Por lo tanto, los objetos son almacenados más allá de la finalización de la aplicación con la que estemos trabajando.

Pueden ser **recuperados** más tarde y **compartidos** por otras aplicaciones.

## Características de las bases de datos orientadas a objetos

- Mantiene una **relación directa** entre el **mundo real y los objetos** de la base de datos. Los **objetos no pierden ni su identidad ni su integridad**.
- Proporciona un **identificador de objeto, generado por el sistema** para cada objeto para que un objeto pueda identificarse fácilmente.
- Son **fáciles de extender** y de **añadir nuevos tipos de datos y operaciones** que se realizarán en ellos.
- Proporciona **encapsulación**. La representación de la información y las implementaciones de los métodos **ocultan entidades externas**.
- También proporciona propiedades de **herencia**, mediante la cual un objeto hereda las propiedades de otros objetos.

### Herencia de objetos

La herencia es una característica muy potente que nos permitirá **definir** una **clase** teniendo en cuenta, y tomando como **base, otra** ya existente. Una de las grandes ventajas de la herencia es la **reutilización de código**.

Los ODBMS proporcionan propiedades de herencia, permitiendo que un objeto **herede las propiedades de otros objetos**.

Los **sistemas relacionales** han implementado estrategias para modelar la herencia, como **tablas separadas o claves foráneas**, pero **no de manera tan inherente** como en los ODBMS.

### Conceptos asociados a los objetos:

- **Atributos:**

Son las **características** que suelen describir los objetos. También conocidos como **variables de instancia**. Cuando los atributos son **asignados a valores** en un momento dado, se asume que el objeto está en un **estado determinado** en ese momento.

- **Objeto:**

Un objeto es una **representación abstracta** de una **entidad** del mundo real, la cual tiene un **identificador único, propiedades embebidas** y la capacidad de **interactuar** con otros objetos por sí mismo.

- **Identidad:**

La identidad es un **identificador externo** (el ID del objeto) que **se mantiene** por cada objeto. El ID del objeto es **asignado por el sistema** cuando el **objeto es creado y no puede ser cambiado**.

Es distinto a las **bases de datos relacionales**, ya que, por ejemplo, este **ID** está **almacenado en el interior** del objeto y, además, se usa para **identificarlo**.

#### **Identificador de Objeto Generado por el Sistema:**

Los ODBMS proporcionan un **identificador único** generado por el sistema para cada objeto, facilitando la identificación única de los objetos.

Los **sistemas relacionales** también proporcionan identificadores únicos (**claves primarias**), pero la generación y gestión de estas claves pueden requerir una **configuración más explícita**.

*A diferencia de las **bases de datos relacionales**, donde los identificadores suelen ser **manejados internamente por el sistema** y **no están accesibles directamente al usuario**, en las **bases de datos orientadas a objetos**, el ID del objeto puede estar almacenado **dentro del objeto mismo** y ser **accesible para los programadores**.*

#### **Inmutabilidad del ID del Objeto:**

*Un aspecto importante es que una vez que se asigna un ID a un objeto, este ID generalmente **no puede ser cambiado** a lo largo de la vida del objeto. Permanece **constante** desde la creación hasta la eliminación del objeto.*

*En resumen, en **bases de datos orientadas a objetos**, la identidad se gestiona mediante un **identificador único** asignado a cada objeto, y este identificador a menudo está almacenado dentro del objeto mismo, siendo **inmutable** a lo largo de la vida del objeto. Este enfoque es diferente de las **bases de datos relacionales** donde los identificadores suelen ser **manejados internamente por el sistema**.*

## Sistemas gestores de bases de datos orientados a objetos

Un sistema gestor de base de datos orientado a objetos se constituye, básicamente, con un sistema gestor de almacenamiento de datos que soporta el modelado y la creación de los datos como objetos. Permite la **conurrencia** y la **recuperación**. Para los consumidores de bases de datos relacionales, significa **olvidarse de** la traducción de **filas** y **columnas**, y, por lo tanto, manipular **directamente con objetos**.

Un sistema gestor de base de datos posee una serie de **datos relacionados entre sí** y una aplicación o **aplicaciones** rodeando dicha base de datos que tendrá **acceso a ella**.

### Características de un sistema gestor de base de datos genérico:

- **Conurrencia.**
- **Persistencia.**
- **Recuperación de errores.**
- Gestión de **almacenamiento**.
- **Consultas.**

### Característica de un sistema gestor de base de datos orientado a objetos, además:

- **Abstracción.**
- **Modularidad.**
- **Jerarquía.**
- **Encapsulación.**
- **Tipología** de objetos.



## ObjectDB

Se puede usar para realizar un **ágil manejo de bases de datos** orientadas a objetos.

Posee una gran variedad de **funcionalidades**, pero habría que destacar, para este caso:

- **Data base explorer:**

Es la herramienta visual de la aplicación donde podremos realizar las **consultas**, **visualizar** los objetos y **editar** el contenido.

- **Database Doctor:** Realiza una serie de **diagnósticos** en relación a posibles problemas de la base de datos.

- **Replication:** Realiza **copias maestro-esclavo**.

- **Online Backup:** Podremos realizar una **copia de seguridad** a través de una **consulta** en un **EntityManager**.

## Lenguaje de consultas para objetos

El lenguaje de consulta de objetos u **Object Query Language (OQL)** es un **lenguaje declarativo** de **tipología SQL** que nos facilitará realizar **consultas** de modo efectivo en bases de datos orientadas a objetos y a estructuras de los mismos.

Este lenguaje **no contiene primitivas** que se ocupen de **modificar el estado** de dichos **objetos**, ya que este tipo de modificaciones se realizarán a través de **métodos que poseen los objetos**.

Partimos de una **estructura básica** de **SELECT, FROM, WHERE** como en el conocido lenguaje SQL. Podemos ver el siguiente ejemplo:

### Ejemplo OQL

```
SELECT c.name  
FROM c in customer  
WHERE c.department = 'e-commerce';
```

### Agregación y asociación

La agregación es un tipo específico de asociación.

**Asociación:** Cuando **borramos** un objeto que forma parte de una asociación, el resto de objetos **relacionados continúan existiendo**.

**Agregación:** Cuando introducimos o **borramos** un objeto que forma parte de una agregación es igual a insertar o **eliminar** el resto de sus **componentes relacionados**. De esta forma, un objeto que pertenece a otro **no puede ser introducido o borrado de forma aislada** en la base de datos.

### Especialización, generalización y herencia

Definimos una **clase** para organizar una serie de **objetos parecidos**.

En algunos casos, los **objetos de una clase** pueden ser **organizados de nuevo** formando ciertas **agrupaciones** que pueden ser **relevantes** para la base de datos.

Podemos ver, en el siguiente ejemplo, que cada una de las agrupaciones formadas son subclases de la clase VEHICULO, y que VEHICULO coge el mandato de superclase del resto de clases. Con esta relación, definimos el concepto de **generalización o especialización**:

*Un ciclomotor es un vehículo.*

*Un coche es un vehículo.*

*Un autobús es un vehículo.*

*Cada subclase de vehículo es una especialización de vehículo, y vehículo es una generalización de sus subclases ciclomotor, coche y autobús.*

## Consultas de un objeto

Vamos a ver distintas formas de consultar el objeto CUSTOMER que vimos en ejemplo OQL anterior.

Las **variables** llamadas **de tipo «iterador»** se pueden **nombrar de 3 formas distintas**:

- **C in Customer.**
- **Customer C.**
- **Customer as C.**

El resultado de este tipo de query podría ser **válido** si forma parte de la **tipología definida previamente en el modelo**.

Por este motivo, **no es obligatorio** que posea la cláusula **SELECT**, ya que, simplemente **nombrando** cualquier **objeto**, se **devolverán** todas sus **existencias**. Es decir:

- **Customer;** Devolverá una colección de todos los objetos de tipo Customer que existan en la base de datos. De la misma forma, si existiera un objeto concreto PremiumCustomer, se realizaría la siguiente consulta:
  - **PremiumCustomer;** Obtendremos el resultado de **ese tipo específico de Customer**.

Por lo general, una **consulta OQL** comienza con el **nombre del objeto** persistente y, a continuación, se le añade uno, varios **atributos** o ninguno, **mediante un punto**.

Veamos algún **ejemplo**:

**customer.empresa:** devolvería un objeto de tipo Empresa, que estaría **vinculado ese Customer**.

**customer.empresa.nombre:** se accede al atributo de la empresa llamado nombre. Por lo tanto, nos devolvería un objeto de tipo **String con el nombre de la empresa**.

**customer.empresas\_asociadas:** nos devolvería un Set<Empresa>, una colección de tipo Empresa donde podríamos ver **todas las empresas** que ese cliente tuviera asociadas.

## Ventajas en la utilización de las bases de datos orientadas a objetos

Estas son algunas de las ventajas y desventajas que presenta desarrollar nuestra aplicación orientada a objetos:

- Dispondremos de una capacidad mayor de realizar el **modelado**. Esto podemos considerarlo debido a:
  - Dentro de los objetos podremos **encapsular** tanto **comportamientos** como **estados**.
  - Las **relaciones** de un objeto pueden ser almacenadas **en su interior**.
  - Al agruparse, los objetos forman **objetos complejos**. Es el concepto que denominamos como **herencia**.
- También dispondremos de una **flexibilidad importante**. Esto se debe a:
  - Podremos construir **nuevos objetos** con tipologías nuevas, partiendo de los que ya tenemos.
  - Se **reduce la redundancia**, ya que podremos aunar características o propiedades de distintas clases y agruparlas en **superclases**.
  - Las **clases** existentes u **objetos** son **reusables**. Lo cual influye directamente en el **tiempo de desarrollo**.
- Por medio de la **intuición**, podremos realizar de forma práctica algunas de las consultas, ya que es un **lenguaje expresivo**. Es realmente fácil navegar entre objetos y sus herencias, ya que es un **acceso navegacional**.
- **Rendimiento** muy competitivo. Algunos autores han comparado rendimientos de bases de datos orientadas a objetos y bases de datos relacionales. **En aplicaciones orientadas a objetos**, el rendimiento de la base de datos orientada a objetos es **superior**.

## Desventajas en la utilización de las bases de datos orientadas a objetos

- La **falta** de un **modelo de datos universal**: No existe dicho modelo de datos universalmente aceptado y a la mayoría de los sistemas gestores de bases de datos orientados a objetos les falta una buena **base teórica**.
- **Falta de experiencia**: Es evidente que no es comparable, en madurez, con los sistemas de base de datos relacionales.
- **Falta de estándares**: Realmente, no existen estándares definidos como tal.
- La **competencia**: Tal y como hemos comentado anteriormente **respecto a los sistemas relacionales**, incluso los objeto-relacionales poseen una gran competencia, ya que dichos sistemas tienen un gran **estándar aprobado**, como SQL, y una base teórica sólida. Además, los sistemas relacionales disponen de un **entramado de aplicaciones** de soporte alrededor de ellas, tanto para usuarios como para programadores.
- La **encapsulación** es casi una forma **obligada** de realizar consultas: El desarrollo de los objetos de forma encapsulada es prácticamente una obligación, ya que es la **forma que accederemos** a futuro mediante el sistema de consultas.
- En relación a la **teoría matemática**: Podemos decir que el modelo de objetos aún **no posee una aprobación**.

## Programación de aplicaciones con acceso a bases de datos orientadas a objetos.

### Diseño de una API (Application Programming Interface)

**API (Application Programming Interface)** define una serie de **especificaciones**, de reglas a cumplir, para consumir ciertas **funcionalidades** de un **sistema externo** determinado.

Hoy en día, el concepto API está muy orientado a las **aplicaciones web** con patrón de diseño **REST**.

En esta arquitectura, se definen una serie de **Endpoints** o puertas de entrada al **código back** de la aplicación con la que hay que conectar y de la que hay que **obtener las funcionalidades** o servicios.

Estas funcionalidades han debido ser **previamente definidas y analizadas**, y deben cubrir la totalidad de la **lógica** que un sistema de **back-end puede devolver** a un front-end.

Si, por **ejemplo**, estamos ante una aplicación de gestión de clientes donde alguna de las **funcionalidades** de la propia aplicación son **crear** clientes, **modificar** clientes y **eliminar**, en la **API** o interfaz de la aplicación habrá **3 EndPoints**, que serán la **puerta principal al código back-end**.

Los EndPoints son, básicamente, **métodos** definidos como entrada para **realizar** dichas **funcionalidades vinculados por un path** que los denomina.

Este sería el caso de una API en tecnología o patrón de diseño REST.

Con respecto a las **bases de datos orientadas a objetos**, debemos ponernos en contexto sobre **qué disponemos y qué necesitamos**.

En nuestro caso, y atendiendo a la temática que concierne esta unidad, **disponemos** de una **aplicación** que en su tramo final **accede a una base de datos orientada a objetos**. En ese punto, sería de gran interés analizar y diseñar una API en la **capa DAO** (Data Access Object), capa de la aplicación donde **accedemos a datos**.

Sería recomendable (y signo de robustez y orden) diseñar una **interfaz** con una serie de **métodos** cuyo objetivo fuera poblar de todas las **respuestas necesarias** en cuanto a **datos** se refiere. Es decir, definir los llamados **Endpoints** cuyo **nombre** dan respuesta al objeto u **objetos que se van a devolver**.

Ejemplo EndPoints:

@GetMapping y @PostMapping serían las puertas de entrada.

```
@RestController
public class ReservationController {

    @Autowired
    private ReservationService reservationService;

    /**
     * Use this endPoint in order to get all the available (FREE) Reservation
     * @return List of ReservationDto
     */
    @GetMapping("/reservations/available")
    List<ReservationDto> getAvailableReservations() { return reservationService.getAvailableReservations(); }

    /**
     * Use this endPoint in order to update one Reservation
     * @param id reservation identifier
     * @param status reservation status: FREE/BLOCKED/CONFIRMED
     * @return ReservationDto Object
     */
    @PostMapping("/reservations/edit/{id}/{status}")
    ReservationDto editReservation(@PathVariable Long id, @PathVariable String status) {

        return reservationService.updateReservation(id, status);
    }
}
```

### Gestión de transacciones en las bases de datos orientadas a objetos

Realmente no vamos a observar gran diferencia en cuanto al concepto de transacción en una base de datos SQL.

Una **transacción** seguirá siendo un conjunto de sentencias que formaran una unidad de trabajo, seguirá manteniendo características de **atomicidad**, **consistencia**, **aislamiento**... solo que en el contexto de las bases de datos orientadas a objetos, dichas transacciones serán **realizadas mediante el uso de objetos**.

## Ejemplo de uso de lenguaje OQL

La siguiente base de datos está formada por **objetos**. Algunos de ellos y sus **atributos** son:

- **Alojamiento:**
  - Object Pared.
  - Object Suelo.
  - Object Persiana.
- **Pared:**
  - String tipoPared.
  - String color.
- **Suelo:**
  - Int extension.
- **Persiana:**
  - Int ancho.
  - Int largo.
  - String color.

Vamos a seleccionar todos aquellos **alojamientos** cuyo **suelo sea superior a 50** y coincidan con el **color de pared blanco**:

Selección de alojamiento, ejemplo OQL:

```
SELECT *  
FROM a in alojamiento  
WHERE a.suelo.extension > 50 AND a.pared.color='blanco'
```

Ahora vamos a seleccionar aquellos **alojamientos** donde la **persiana** tenga un **largo inferior a 100**, el **color de la persiana sea «metalico»** y el tipo de la **pared** sea «**gotelé**».

Especificación de persiana y pared:

```
SELECT *  
FROM a in alojamiento  
WHERE a.persiana.largo < 100 AND a.persiana.color='metalico' AND a.pared.tipoPared='gotele'
```





Un **ingeniero** es un objeto **independiente**, pero que **forma parte del** objeto **empleado**, que es otro **objeto**, y que, a su vez, **forma parte del** objeto **persona**, que **engloba a todos ellos**.

Para acceder al atributo **nombre** de un empleado de tipo **ingeniero**, valdría con:

**Persona.empleado.ingeniero.nombre**

De esta forma, devolveríamos el atributo nombre, que estaría definido como un String.

Cabe destacar que todos y cada uno de los **objetos** por los que se ha pasado tendrán **diferentes atributos**.

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

## **Bases de datos objeto relacionales**

### **Tablas y Tipos de objetos**

#### **VARRAY**

#### **Tabla anidada**

## Definición de base de datos objeto relacionales

Una base de datos objeto-relacional es una combinación de una base de datos orientada a objetos y un modelo de base de datos relacional. Esto significa que soporta objetos, clases, herencia, etc. que tendríamos en los modelos orientados a objetos y también tiene soporte para tipos de datos o estructuras tabulares, entre otras cosas, del modelo de datos relacional.

Uno de los mayores objetivos de los modelos de base de datos objeto-relacionales es acortar distancias entre las bases de datos relacionales y las practicas orientadas a objetos realizadas frecuentemente en diferentes lenguajes como C++, Java, C#, etc.

Con esta información que se ha adelantado de base, podríamos definir la base de datos objeto-relacional como aquella base de datos relacional que evoluciona desde dicho modelo hacia a algunas de las características del modelo de objetos, haciéndola una **base de datos híbrida**.

Uno de los **gestores** de bases de datos **más conocidos** hoy en día es **Oracle**. Este **implementa el modelo de objeto** como una **extensión del modelo relacional**.

Muchos **lenguajes**, tal y como hemos comentado anteriormente, han sabido adaptarse con **extensiones y frameworks** para poder **trabajar** con estas nuevas **bases de datos objeto relacionales de Oracle**.

El resultado es un modelo relacional de objetos que ofrece la **intuición y la economía de una interfaz de objetos**, al mismo tiempo que conserva su alta **conurrencia y el rendimiento de una relacional**.

## Características de las bases de datos objeto relacionales

Una de las principales características de este tipo de base de datos es que podremos crear **nuevos tipos de datos**, los cuales permitirán gestionar aplicaciones específicas con mucha **riqueza de dominios**. Estos nuevos tipos de datos pueden ser **tipos compuestos**, lo que nos lleva a pensar que **se podrán definir**, al menos, **dos métodos**:

- Uno para convertir de este **tipo a** caracteres **ASCII**.
- Y otro que haga esta función a la inversa, desde caracteres **ASCII hasta** nuevos **tipos** de datos.

Se soportarán distintos **tipos complejos** como, por ejemplo:

- **Registros**,
- **Listas**,
- **Referencias**,
- **Pilas**,
- **Colas**,
- **Arrays**.

Con dicha tipología de datos, podremos crear también **funciones** que tengan código en **diferentes lenguajes**, como **SQL, Java, C#**, etc.

### Características de las bases de datos objeto-relacionales:

- Dispondremos de una mayor capacidad de expresión para **definir conceptos y** diferentes **asociaciones**.
- Podremos crear también **operadores** asignando **nombre y existencia** de aquellas **consultas más complejas**.
- En los **tipos** de registro, **estilo relacional**, podremos usar **encadenamiento y herencia**.
- Podremos hacer uso de la **reusabilidad**, compartiendo **bibliotecas** de clases definidas previamente.
- Posibilidad de introducir **comprobación de reglas de integridad** por medio de **triggers**.

#### **Triggers**

Trigger también llamado **disparador**, es simplemente un **script** de código que puede estar **escrito en diferentes lenguajes**.

Consiste básicamente en **ejecutar** una serie de **procedimientos**, **según** ciertas **instrucciones**, **cuando se realicen** determinadas **operaciones** en la información de la base de datos.

## Definición de tablas y tipos de objetos

Cuando se crea un **tipo de dato**, realmente estamos definiendo cierto **comportamiento** para una **agrupación de datos** de nuestra aplicación. En **Oracle**, con la base de datos objeto-relacional, tendremos la opción de **definir nuestros propios tipos** de datos. Para los tipos de objetos, usaremos **object type** y, para los tipos de colecciones, **collection type**. Para **construir** dichos **tipos** de usuario, deberemos **usar los básicos** que poseemos en el sistema.

Un **objeto** representa una entidad en el mundo real y **se compone de**:

- **Nombre**: Con el que identificaremos el **tipo de objeto**.
- **Atributos**: Con los que definiremos la **estructura**. Los atributos pueden ser de **tipo creado** por el usuario **o básico** del propio sistema.
- **Métodos**: Que pueden ser **funciones o procedimientos**. Los encontraremos escritos en **código PL/SQL** cuando están almacenados **en la propia base de datos** y en el **lenguaje C** cuando se almacenan **externamente**.

Diremos que la **creación de un método** en **Oracle** se realiza **junto a la creación de su tipología** y debe llevar siempre el **tipo de compilación** como, por ejemplo:

```
PRAGMA RESTRICT_REFERENCES;
```

De esta forma: **evitamos la manipulación** de los diferentes **datos** o de las distintas **variables PL/SQL**.

Ejemplo de código de creación de un tipo de dato nuevo (address\_t) en el lenguaje, establecido por la base de datos Oracle:

```
CREATE TYPE address_t AS OBJECT (  
    street VARCHAR2(200),  
    city VARCHAR2(200),  
    prov CHAR(2),  
    postcode VARCHAR2(20)  
);
```

Como podemos, observar en la creación de la tabla:

- Indicaremos el **nombre del tipo a definir** "address\_t".
- **Estableceremos** cuatro **atributos** diferentes que **definen la estructura** creada:
  - **Street:** Es un tipo **texto** VARCHAR2, con **200 caracteres máximo**.
  - **City:** Otro VARCHAR2, también con 200 de extensión.
  - **Prov:** Válido para introducir **2 caracteres máximo**.
  - **Poscode:** Un VARCHAR2 de 20 caracteres máximo.

## Nulos

Vamos a explicar la cláusula nula (null) en los objetos de tipo usuario. Cuando creamos un objeto de tipo, tendrá “x” atributos, y por lo tanto el objeto nunca será atómicamente nulo, es decir que no será nulo completamente. En el ejemplo a continuación se crea un objeto llamado person\_typ, y se definen sus atributos. Vamos a usar este objeto person\_typ: creamos una tabla llamada contact, la cual tiene un atributo tipo person\_typ. Al hacer un INSERT incluimos un objeto person\_type, el cual tiene algunos atributos nulos (idno NUMBER, name VARCHAR y phone VARCHAR). Pero como ya hemos dicho, un objeto de tipo no es atómicamente nulo, y sus atributos pueden ser inicializados a nulo como en este caso, o reemplazados mas tarde. Esta es una característica de los objetos tipo usuario.

```
CREATE OR REPLACE TYPE person_typ AS OBJECT (  
    idno 1 NUMBER,  
    name VARCHAR2 (30),  
    phone VARCHAR2 (20),  
    MAP MEMBER FUNCTION get_idno RETURN NUMBER,  
    MEMBER PROCEDURE display details (SELF IN OUT NOCOPY person_typ ) );  
/  
CREATE OR REPLACE TYPE BODY person_typ AS  
    MAP MEMBER FUNCTION get_idno RETURN NUMBER IS  
    BEGIN  
        RETURN idno;  
    END;  
    MEMBER PROCEDURE display_details (SELF IN OUT NOCOPY person_typ ) IS  
    BEGIN  
        — use the PUT_LINE procedure of the DBMS_OUTPUT package to display  
        details  
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' - ' || mame || ' - ' || phone);  
    END;  
END;  
/  
CREATE TABLE contacts (  
    contact person_typ,  
    contact_date DATE );  
  
INSERT INTO contacts VALUES (  
    person_typ (NULL, NULL, NULL), '24 Jun 2003' );  
  
INSERT INTO contacts VALUES (  
    NULL, '24 Jun 2003' );
```



## “OBJECT TYPES”

Un tipo de objeto es un tipo de dato que puede utilizarse de manera similar a tipos de datos estándar, como NUMBER o VARCHAR2 en Oracle Database. Puede especificarse como el tipo de datos de una columna en una tabla relacional y declarar variables de ese tipo. Una instancia de un tipo de objeto se denomina objeto.

Los tipos de objetos actúan como planos o plantillas que definen tanto la estructura como el comportamiento. Son objetos de esquema de base de datos y están sujetos al mismo control administrativo que otros objetos de esquema. El código de aplicación puede recuperar y manipular instancias de estos objetos.

Se utiliza la instrucción SQL CREATE TYPE para definir tipos de objetos. Para crear un tipo de objeto llamado person\_typ podríamos seguir el siguiente ejemplo:

```
CREATE TYPE person_typ AS OBJECT (  
    idno  
    NUMBER,  
    first_name  
    VARCHAR2(20),  
    last_name  
    VARCHAR2(25),  
    email  
    VARCHAR2(25),  
    phone  
    VARCHAR2(20),  
    MAP MEMBER FUNCTION get_idno RETURN NUMBER,  
    MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ );  
/  
CREATE TYPE BODY person_typ AS  
    MAP MEMBER FUNCTION get_idno RETURN NUMBER IS  
    BEGIN  
        RETURN idno;  
    END;  
    MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS  
    BEGIN  
        -- utilizar PUT_LINE del paquete DBMS_OUTPUT para mostrar detalles  
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' || first_name || ' ' || last_name);  
        DBMS_OUTPUT.PUT_LINE(email || ' ' || phone);  
    END;  
END; /
```

El símbolo / que vemos en el código anterior sirve para indicar al motor de base de datos que ese bloque de código ya está listo y puede ser ejecutado.

## Ejemplo de creación de nuevos tipos de objetos

Vamos a crear estos dos objetos:

- **Objeto Vehículo** con los atributos: número ruedas, peso, largo.
- **Objeto Coche** con los atributos: tipo, marca.

Hay que tener en cuenta que el objeto Coche es un **tipo de Vehículo**.

En Oracle, con la base de datos objeto-relacional, podremos agregar nuestros propios tipos de objetos a la base de datos.

**Definiremos los 2 objetos** que se requieren. Habría que reflexionar sobre **qué tipología** tienen dichos objetos.

En primer lugar, definiríamos el objeto vehículo teniendo en cuenta que, tanto para el número de ruedas, como el peso, y el largo, son variables del tipo básico del sistema NUMBER:

Definición de objeto vehículo

```
CREATE TYPE vehiculo_t AS OBJECT (  
    nRuedas NUMBER,  
    peso NUMBER,  
    largo NUMBER);
```

En la siguiente definición, habría que caer en la cuenta de que **conlleva herencia**, por lo tanto, la **tipología** de este **coche es vehículo**, precisamente el mismo objeto que definimos previamente.

Por último, la marca nos valdría con un Varchar de 50 caracteres.

Definición objeto coche

```
CREATE TYPE coche_t AS OBJECT (  
    tipo vehiculo_t,  
    marca VARCHAR2(50) );
```

## Explotación de tablas y tipos de objetos

Una vez se ha realizado la definición de **tipos**, podemos tener **distintos objetivos** para esos nuevos datos. Podemos usarlos:

- para definir **nuevos tipos**,
- para **almacenarlos en tablas** de ese tipo de datos,
- para definir los distintos **atributos de una tabla**.

Sabemos que una **tabla de objetos** es una **clase** específica de tabla que almacenará un **objeto por cada fila** y que, al mismo tiempo, **facilita el ingreso de los atributos** del mismo objeto, como si se tratara de **columnas** de la tabla. Por ejemplo, podríamos tener una tabla de vehículos del año actual y otra para guardar vehículos de años anteriores:

### Tablas Oracle

Tabla que **almacena objetos** con su **propio ID**:

```
CREATE TABLE vehiculos_año_tab OF vehiculo_t (  
    numVehiculo PRIMARY KEY);
```

No es una tabla de objetos, sino una tabla con una columna cuyo tipo de dato es un objeto.

Posee una columna con un tipo de datos complejo y sin identidad de objeto. Es una de las ventajas que ofrece Oracle:

```
CREATE TABLE vehiculos_antiguos_tab (  
    anio NUMBER, vehiculo vehiculo_t);
```

Aparte, **Oracle** nos permite **definir como tabla**:

- Una **columna** con tipología de **objeto**. (Según el documento original: una tabla con una sola columna cuyo tipo es el de un tipo de objetos).
- Aquella que tiene el **mismo número de columnas** como **atributos** que almacena. (O mejor dicho desde el documento original: una tabla que tiene **tantas columnas como atributos los objetos que almacena**).

## Tipos de colección: array

Para poder establecer relaciones «uno a muchos» (1:N), Oracle nos permite definir **colecciones**. Una colección está formada por un **número no definido de elementos** y todos ellos deben ser del **mismo tipo**. De esta forma, podemos guardar en un simple atributo un conjunto de datos en forma de array (**Varray**) o, también, tendríamos la opción de la **tabla anidada**.

### EL VARRAY

Como bien sabemos, podríamos definir un array como una serie de **elementos ordenados** que son del **mismo tipo**.

Estos elementos llevan asociado un **índice** que nos sirve para saber su **posición** dentro del array.

Oracle permite que el tipo VARRAY sea un tipo de dato **variable**, pero sí se debe **establecer** el **máximo de elementos** una vez se declara dicho tipo. Podemos ver algún ejemplo:

VARRAY

```
CREATE TYPE numeros AS VARRAY(10)
  OF NUMBER(10);
numeros ('6', '18', '75870');
```

Tal y como se observa en el código de arriba, hemos definido un nuevo tipo «números» como un VARRAY con máximo 10 elementos y con posiciones cuyo tipo serán NUMBER máximo 10 cifras.

Utilizaremos el VARRAY para:

- Definir la **tipología de una columna** de una **tabla** relacional.
- Definir la **tipología de un atributo** de un tipo **objeto**.
- Definir una **variable del lenguaje PL/SQL**.

Una vez declaramos este objeto VARRAY, **no se reserva** realmente ninguna cantidad de **espacio**. Si el espacio está disponible, se almacena **igual que el resto de columnas**, pero, si por el contrario, es **superior a 4.000 bytes**, se almacenará en una **tabla aparte**, como un dato de tipo **BLOB**.

## BLOB

BLOB viene de las iniciales **Binary Large Object**. Es un lugar en la base de datos donde se va a almacenar **información binaria** de este tipo de array que añadamos. Existen los BLOB y los **CLOB** (**Character Large Object**), que son objetos grandes que almacenan **cadena de caracteres**.

## Tipos de colección: tablas anidadas

Los tipos de colección en Oracle objeto-relacional son:

- **VARRAYS**,
- **Tablas anidadas**.

### Tablas anidadas

Tenemos la posibilidad de **anidar objetos con herencia** en nuestra base de datos objeto-relacional.

Una tabla anidada es una lista de **elementos no ordenados** que mantienen una **misma tipología**. El **máximo no está especificado** en la definición de la tabla, y el orden de los elementos no se mantiene.

Realizaremos **SELECT, INSERT, DELETE y UPDATE** de la misma forma que lo hacemos con las tablas comunes, **usando la expresión TABLE**.

Una **tabla anidada** puede ser vista o interpretada **como una única columna**.

Si la **columna** en una tabla anidada es un tipo de **objeto de usuario**, la tabla puede ser vista como una **tabla multicolumna**, con **una columna por cada atributo** del **objeto usuario** que fue definido.

Sintaxis para crear una tabla anidada:

```
CREATE TYPE nombre_tipo AS TABLE OF tabla_tipo;
```

Con este código, estaremos creando una **tabla anidada tabla\_tipo**, la cual contendrá **objetos de tipo de usuario nombre\_tipo**.

La definición que hemos visto justo arriba **no asignará espacio**. Una vez **definido el tipo**, podremos **usarlo para**:

- El **tipo** de datos de una **tabla relacional**.
- Un **atributo** de un **objeto** de tipo usuario.
- Una **variable PL/SQL**, un **parámetro** o una **función que devuelva** un tipo.

### **Ejemplo de tabla anidada**

```
CREATE TYPE nested_table_type AS TABLE OF VARCHAR2(50);
```

Aquí, se crea un tipo llamado `nested_table_type`, que es una tabla anidada de tipo `VARCHAR2` con una longitud máxima de 50 caracteres.



## Referencias

La base de datos objeto-relacional Oracle permite que los **identificadores únicos** que se les asigna a los **objetos de una tabla** puedan ser **referenciados desde** los **atributos** de otros objetos distintos o desde la **columna** de una tabla.

Hablamos del **tipo denominado REF**, cuyo **atributo** guardará una **referencia** (un enlace) a un **objeto** de la tipología definida y genera una **relación entre ambos** objetos.

Este tipo de referencias se usarán para **acceder a los objetos relacionados** y **actualizarlos**, pero **no es posible** realizar operaciones **directamente sobre las referencias**. Para usar una referencia o actualizarla, usaremos **REF o NULL**.

Una vez hemos definido una columna de tipo REF, se puede **acotar el alcance a los objetos** que se guarden en una determinada **tabla**. A continuación, veremos, en el siguiente Código, un atributo de tipo REF que **restringe su dominio a una determinada tabla**.

### Referencias

```
CREATE TABLE clientes_tab OF clientes_t;

CREATE TYPE ordenes_t AS OBJECT (
    ordennum NUMBER,
    cliente REF clientes_t,
    fechapedido DATE,
    direntrega direccion_t);

CREATE TABLE ordenes_tab OF ordenes_t (
    PRIMARY KEY (ordennum),
    SCOPE FOR (cliente) IS clientes_tab);
```

Tal y como podemos observar, al inicio se **crea una tabla** de tipo clientes\_t. A continuación, se crea un **tipo de objeto** llamado ordenes\_t con 4 atributos. El segundo de ellos será el **objeto que será referenciado**.

Por último, tenemos la **creación de la tabla ordenes\_tab**, donde define una primary key y el segundo dato será, precisamente, esa **referencia** que hemos definido en el objeto anterior. De modo que:

*El **atributo referenciado** se construye agregando a continuación “REF” y seguidamente la **restricción** a objetos de cierta **tabla**.*

*Para **referenciar el atributo**, se usa: “**SCOPE FOR**” seguido del **atributo referenciado entre paréntesis** y a continuación, fuera de los paréntesis, la **tabla** en la que tiene **restringido el dominio** el atributo referenciado.*

*Para ganar consistencia en la BBDD, utilizamos la **cláusula SCOPE IS** para indicar que sí o sí tiene que existir el curso al que hace referencia desde la tabla estudiante:*

```
CREATE TABLE Estudiantes OF TipoEstudiante (  
    PRIMARY KEY (id),  
    curso REF TipoCurso SCOPE IS Cursos  
);  
CREATE TABLE Cursos OF TipoCurso (PRIMARY KEY (id));
```

## Herencia de tipos

La herencia de tipos nos permite crear **jerarquías** de tipos.

Una jerarquía de tipos es una serie de niveles sucesivos de subtipos, **cada vez más especializados**, que derivan de un tipo de objeto ancestro común, denominado **supertipo**. Esto, como se puede observar, no es un concepto nuevo, ya que, en programación orientada a objetos, lo usamos muy frecuentemente, sobre todo en lenguaje Java.

Los subtipos derivados **heredarán las características** del tipo de objeto principal y pueden **ampliar la definición** de este.

Los tipos especializados pueden **añadir nuevos atributos o métodos**, o **redefinir métodos** heredados de la clase tipo padre. La jerarquía del tipo resultante facilita un **nivel superior de abstracción** para manejar la complejidad de un modelo de una aplicación.

A continuación, mostraremos un pequeño esquema donde podremos ver la herencia entre objetos de tipo usuario.

Podremos observar cómo partimos de una clase principal e iremos heredando atributos, al mismo tiempo que los hijos podrán ir aportando nuevas características a los de los objetos padre:

### Esquema herencia

Podemos observar que el tipo «Persona» poseerá una serie de atributos que serán heredados tanto por el tipo «Estudiante» como por el tipo «Empleado».

«Estudiante» y «Empleado», a su vez, agregarán, en caminos diferentes, nuevos atributos para los tipos hijos que se puedan crear. En este caso, un hijo de «Estudiante» es «EstudianteTiempoParcial» que, finalmente, heredará atributos de su padre «Estudiante» y del padre de su padre, «Persona».

## Ejemplo de Herencia de tipos

Teniendo en cuenta los siguientes tipos:

Vehículo\_t, coche\_t, ciclomotor\_t, cocheCarreras\_t, cochePaseo\_t, motocicleta\_t, ciclomotorPaseo\_t.

Situamos el tipo padre arriba.

La relación de tipos padre a hijos sería la siguiente:

Esquema herencia de tipos

**“Vehículo”**

es el padre,

del que heredan:

**“Ciclomotor” “Motocicleta” y “Coche”**

**“CiclomotorPaseo”** hereda de **“Ciclomotor”**

y

**“CocheCarreras” y “CochePaseo”** heredan de **“Coche”**

## Ejemplo de creación de una base de datos objeto relacional, un objeto y una tabla.

El objeto será de tipo **usuario**, tendrá el nombre **vehículo**, y dispondrá de 2 **atributos**:

- Nombre
- Marca

Marca tendrá una **referencia directa** al objeto en sí, y nombre será de tipo STRING.

La tabla tendría que ser creada de **objetos de tipo vehículo**.

Se requiere de la creación de, básicamente, dos elementos:

- Creación de un **objeto** nuevo de tipo usuario,
- Creación de una **tabla**.

### Creación del objeto

```
CREATE TYPE vehiculo_typ AS OBJECT (  
    nombre VARCHAR2(30),  
    marca REF vehiculo_typ);
```

En este código, podemos ver cómo creamos un **nuevo type**, que se denominará vehiculo\_typ y que tendrá **2 atributos**, como indica el ejercicio: nombre (Varchar) y marca (será una referencia al propio objeto).

## Definición de la tabla

```
CREATE TABLE vehiculo_tabla OF vehiculo_typ;
```

De esta forma, estaremos definiendo una tabla cuyas filas serán objetos del tipo anteriormente definido.

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

## **Introducción a las bases de datos No-SQL**

### **Concepto Big Data**

## Características de las bases de datos No-SQL

Tradicionalmente, la industria del software como ya bien sabemos, usa las bases de datos relacionales para almacenar y manejar datos de forma persistente. **No solo bases de datos SQL y No-SQL** son las que han emergido en este pasado reciente.

**No-SQL** se refiere a **todas** las bases de datos y almacenes de datos que **no están basadas** en el sistema tradicional de bases de **datos relacionales** en sí, o no están basadas en sus **principios**.

Suele relacionarse con **grandes conjuntos de datos** a los que se accede y se manipulan a **escala web**. El concepto No-SQL no representa un producto único o una única tecnología, representa un **grupo de productos** y un conjunto relacionado de **conceptos** para el almacenamiento y su administración. Fue también un **hashtag** escogido para una **gran reunión técnica** para discutir las **nuevas bases de datos** (un hashtag es un conjunto de caracteres precedidos por una almohadilla (#) que sirve para identificar o etiquetar un mensaje en las webs de microblogs, p. e. el hashtag #elecciones en Twitter).

Las bases de datos No-SQL tienen una serie de **características comunes** como:

1. Modelo de datos **no relacional**.
2. Funcionan bien en **clusters** (cluster podría traducirse al español como aglomerado, grupo, racimo o conjunto. Dentro de las Tecnologías de la Información (TI), cluster significa integrar **dos o más computadoras** para que trabajen **simultáneamente** en el **procesamiento de una determinada tarea**).
3. Suelen ser en su **mayoría** de **código abierto**.
4. Su construcción está orientada para las **aplicaciones web de nueva generación**.
5. Son bases de datos con **ausencia de esquema** (**schemaless**).

Dentro de las diferentes características mencionadas aclararemos algunas de ellas que puedan dar lugar a dudas:

**Funciona bien en modo cluster:** Podemos definir básicamente este concepto como un conjunto de máquinas implementadas como servidores de procesamiento paralelo. Funcionan entre sí como un único recurso. Y como bien sabemos a cada elemento o máquina lo llamamos nodo.

**Cluster de servidores:** es un grupo de servidores vinculados que trabajan en estrecha **colaboración** y **se implementan** para mejorar el **rendimiento** y/o la **disponibilidad**, en comparación a los recursos ofrecidos por un solo servidor.

**Bases de datos con ausencia de esquema o schemaless:** Es una característica **muy flexible**; se puede almacenar **información no uniforme** y se facilita así la **evolución**.



## Fundamentos de las bases de datos No-SQL

Con la explosión del **social-media** y el **contenido manejado** por el usuario, han **incrementado** el volumen y el tipo de datos que se **producen, se manejan, analizan y se archivan**, por lo que han provocado la rápida eclosión de las bases de datos No-SQL. Además, a esta cantidad ingente de datos, hay que sumarle las aportaciones de las nuevas **fuentes de información** como:

- **sensores**,
- sistemas globales de posicionamiento o **GPS**,
- **rastreadores**,
- otros tipos de **sistemas que monitorean grandes cantidades** de información de forma regular.

Estos grandes paquetes de información, han introducido nuevos retos y oportunidades para el almacenamiento de información. Además, los datos son cada vez más **semiestructurados y escasos**. Esto quiere decir que las **bases de datos relacionales son examinadas** y requieren una **definición de esquema inicial y referencias** relacionales.

Para resolver el problema relacionado con los grandes volúmenes de información y los datos semi-estructurados, surge **nuevas clases de bases de datos** dentro de la familia **No-SQL**. Este tipo de base de datos consisten en el **almacenamiento basado en:**

- **columnas**,
- **clave/valor**,
- **documentos**.

A continuación, veremos las **diferencias más notables** entre las bases de datos relacionales y las No-SQL:

### Diferencias BBDD relacionales y NoSQL

#### Bases de datos relacionales

- La información está almacenada en un modelo relacional con **filas y columnas**.
- Una **fila** contiene información sobre un **elemento** mientras que las **columnas** contienen información específica.
- Sigue un **esquema fijo**: Las columnas son **definidas y establecidas** antes de la entrada de datos. Además, **cada fila** contiene información de **cada columna**.
- A favor del **escalado vertical**.
- Atomicidad, consistencia, aislamiento y durabilidad. (**ACID**)

#### Bases de datos NoSQL

- La información está almacenada en un cliente o en una base de datos diferente con **modelos de datos distintos**.
- Sigue un **esquema dinámico**, puedes añadir columnas en cualquier momento.
- Facilita el **escalado horizontal**. Se puede escalar a través de **servidores múltiples**. Los servidores múltiples tienen la **ventaja** del **precio** en relación al Hardware que se va añadiendo comparado con el escalado vertical.
- **No** está a favor de los principios **ACID**.

## Beneficios de las bases de datos No-SQL

A continuación, en las próximas diapositivas, veremos en profundidad los diferentes beneficios técnicos de una solución No-SQL a nivel empresarial:

### a) Capacidad de fuente de datos primaria y de análisis

El primer criterio de una clase empresarial NoSQL es que debe servir como **fuentes principales** de datos que **recibe** información de **distintas aplicaciones** de negocio. También debe actuar como **segunda fuente de información o análisis** las aplicaciones de “**business intelligence**”. Desde el punto de vista de negocio, este tipo de bases de datos deben de ser capaces de **integrarse de una forma rápida** a todos los tipos de **datos estructurados, semiestructurados o sin estructura**. Además, deben ser capaces de ejecutar consultas de alto rendimiento.

### b) Capacidad Big Data

Las bases de datos NoSQL no se limitan a trabajar con Big Data. Una base de datos de este tipo de clase empresarial, puede escalar para **administrar grandes volúmenes de datos** desde Terabytes **hasta Petabytes (10<sup>6</sup> GB, o 1 millón de GigaBytes)**. Además de almacenar grandes volúmenes de datos, ofrece un **alto rendimiento** para la **velocidad, variedad y complejidad** de los datos.

### c) Disponibilidad continua

Para que una base de datos sea considerada de clase empresarial, debe ofrecer disponibilidad continua, sin un solo punto de fallo.

Además, en lugar de proporcionar la función de disponibilidad continua fuera del software, la solución NoSQL ofrece una disponibilidad continua integrada, que debe incluir las siguientes características clave:

1. Todos los nodos de un clúster deben poder **atender solicitudes** de lectura incluso **si algunas máquinas no funcionan**.
2. Debe ser capaz de **replicar y segregar datos fácilmente** entre diferentes partes físicas en un centro de datos. Esto **evitará cortes** de hardware.
3. Debe poder admitir **diseños de distribución de datos** que sean **centros de datos múltiples** ya sea en instalaciones físicas o en la nube.

### d) Capacidad de tener **múltiples centros de datos**:

Por lo general, en un entorno profesional, las empresas poseen bases de datos altamente distribuidas que se encuentran en varios centros de datos y ubicaciones geográficas distintas.

La replicación de datos es una característica que ofrecen todas las bases de datos relacionales. Sin embargo, ninguna puede ofrecer un **modelo simple de distribución de datos entre varios centros** de datos **sin causar problemas** de rendimiento.

Una buena solución empresarial NoSQL debe admitir la **implementación de varios centros de datos** y debe proporcionar una **opción configurable** para mantener un **equilibrio entre el rendimiento y la coherencia**.

e) **Fácil replicación** *independientemente* de la ubicación:

Para evitar que la pérdida de datos afecte a una aplicación, una buena solución NoSQL proporciona una **gran capacidad de replicación**. Estos incluyen una capacidad de lectura y escritura **en cualquier lugar** con **compatibilidad** total y *independencia* de ubicación.

Esto significa que se pueden escribir datos en **cualquier nodo de un clúster**, hacer que se repliquen en otros nodos, y ponerlos a disposición de todos los usuarios **independientemente de su ubicación**.

Además, la capacidad de escritura en cualquier nodo debe garantizar que los **datos estén seguros en caso de un corte** de suministro eléctrico o cualquier otro tipo de incidente.

f) **Sin capa de almacenamiento en caché separada**

Una buena solución NoSQL es capaz de **utilizar múltiples nodos** y **distribuir datos** entre **todos los nodos** adjuntos.

Una vez aclarado esto, podemos decir que **no requiere** una capa de almacenamiento en **caché específica** para almacenar datos. Las **memorias caché de todos los nodos** pueden almacenar datos para una **entrada y salida rápida** o para un **acceso entrada/salida**. La base de datos NoSQL **elimina el problema de sincronizar** los datos de la **caché con la base de datos** persistente. Por lo tanto, admite una escalabilidad simple con **menos problemas de administración**.

g) **Lista para la nube**

Dado que la adaptación de la **infraestructura de la nube** aumenta día a día, una solución NoSQL de nivel empresarial debe estar preparada para la nube.

Un clúster de base de datos NoSQL debe poder funcionar en una configuración de la nube, como **Amazon EC2**, y también debe poder **extender y reducir un clúster** cuando sea necesario. También debe admitir una **solución híbrida** en la que parte de la base de datos se aloje dentro de las **instalaciones de la empresa** y otra parte se aloje en la **nube**.

h) **Alto rendimiento con escalabilidad lineal**

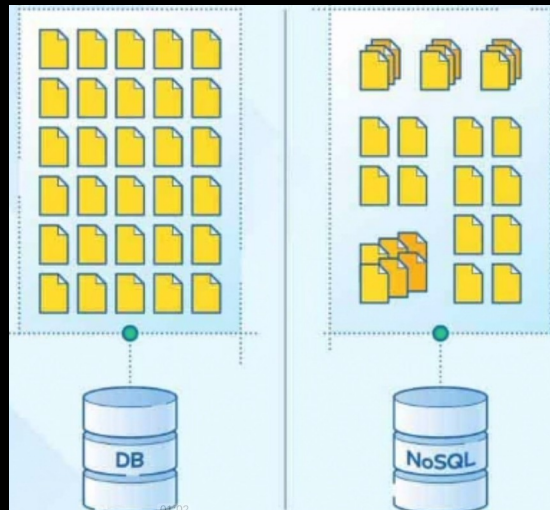
Una base de datos de este tipo puede mejorar el rendimiento **agregando nodos** a un clúster. Normalmente, el rendimiento de otros sistemas de bases de datos puede disminuir cuando se agregan los nodos adicionales, sin embargo, una buena solución NoSQL **aumenta el rendimiento** de las operaciones de lectura y escritura cuando se agregan **nodos adicionales**. Estas **ganancias** de rendimiento son **de naturaleza lineal**.

Finalmente, algunos beneficios más de las bases de datos No-SQL serían:

- Soporte de **esquema flexible**.
- Admite **lenguajes y plataformas clave** para desarrolladores.
- **Fácil de implementar, mantener y extender**.
- **Comunidad de código abierto**.

## Desventajas NoSQL

Podemos ver aquí esquemáticamente un resumen de lo que son la base de datos relacionales y la base de datos no SQL.



Referente a la base de datos NoSQL, ya hemos dicho que son bases de datos que incluyen diferentes tipos de almacenes, como almacenes de tipo columna, de documento, de clave valor, de gráficos, etc., cosas que hacen fácil y rápido el acceso; por ejemplo, en las de **clave valor**, cuando se asigna un hash directamente a una clave para obtener el valor, este tipo de acceso y de entrega de documentos es muy eficiente.

Los beneficios ya los hemos visto, pero lo que realmente no hemos visto aún son algunas **desventajas**, que también hay que decirlas:

1. Hay que decir que la mayoría de bases de datos no SQL **no admiten funciones de fiabilidad** y que son soportadas por sistemas de bases de datos relacionales, estas de las que hablamos. Estas características las podemos resumir como los principios de:
  - atomicidad,
  - consistencia,
  - aislamiento,
  - durabilidad.
2. Con el fin de apoyar estas características de fiabilidad y coherencia, los propios desarrolladores de bases de datos no SQL **deben implementar su propio código**, lo que esto le da un plus de **complejidad** al sistema. Al tener el propio tipo de sistema desarrollado por ellos mismos, ellos son los propios que tienen que desarrollar el código. Podría ser otra desventaja. Esto podría limitar el número de aplicaciones en las que podemos confiar para relacionar, por ejemplo, transacciones seguras y confiables.

En este tipo de **aplicaciones bancarias y de sistemas de datos** no SQL estas acciones **no se pueden** hacer.

3. La mayoría de bases de datos no SQL **no tienen funciones de fiabilidad y coherencia**. Esto, por ejemplo, puede ser que con el fin de apoyar estas características de fiabilidad y coherencia no se pueda hacer.
4. Otras formas de complejidad encontradas en la mayoría de las bases de datos no SQL incluyen que no tienen compatibilidad con las famosas consultas SQL. Esto significa que tienen que tener su propio lenguaje de consulta de datos manual y hace el proceso un poco más lento y complejo.

## Tipos de bases de datos NoSQL

Existen **cuatro tipos** de bases de datos NoSql en el mercado:

### 1. Bases de datos **clave/valor**:

Velocidad brutal.

La base de datos clave valor, es realmente una **tabla hash de claves y valores**.

Algunos de los **ejemplos** son:

- **Riak**,
- **Tokyo Cabinet**,
- servidor **Redis**,
- Memcached,
- Scalaris.

### 2. Bases de datos **basadas en documentos**:

Este tipo de bases de datos basadas en documentos almacena documentos compuestos por **elementos etiquetados**.

Algunos ejemplos son:

- **MongoDB**,
- **CouchDB**,
- **OrientDB**,
- **RavenDB**.

### 3. Bases de datos **basadas en columnas**:

**Cada bloque** de almacenamiento contiene **datos de una sola columna**.

Ejemplos:

- **BibTable**,
- **Cassandra**,
- **Hbase**,
- **Hypertable**.

### 4. Bases de datos **basadas en gráficos**:

Una base de datos basada en gráficos es una base de datos de red que **utiliza nodos para representar y almacenar datos**.

Algunos ejemplos son:

- **Neo4J**,
- **InfoGrid**,



- **Infinite Graph,**
- **FlockDB.**

La **disponibilidad de opciones** en las bases de datos NoSQL tiene sus propias

- **ventajas** (permite elegir un diseño **de acuerdo a los requisitos** de su sistema),
- e **inconvenientes** (aun ajustando el producto a los recursos del sistema **no siempre funcionará** correctamente).

## Ejemplo REDIS NoSQL Database

Para realizar una **conexión directa** con una base de datos **No-SQL Redis**, imaginemos que nuestra aplicación está desarrollada en **lenguaje Java** y tenemos que implementar la parte de la **conexión a base de datos**.

Vamos a realizar la **construcción de la capa de acceso a datos** de la aplicación.

Los **pasos a seguir** serán los siguientes:

1. Agregación de **dependencias** a nuestro fichero pom.xml del proyecto, para cargar la **librería** necesaria y así **acceder a la base de datos Redis**.

### Dependencia Redis

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

Como vemos en el código anterior de esta forma **agregaremos la versión** específica que nos realizará la **conexión con la base de datos**.

2. **Implementación** de la nueva capa usando las **librerías previamente importadas** en **Maven**.

### Integración Redis

```
import redis.clients.jedis.Jedis;

public class RedisConnector {
    public static void main(String[] args) {
        Jedis cliente = new Jedis("localhost");
        cliente.set("clave", "valor");
        String value = cliente.get("clave");
        System.out.println(value);
        cliente.close();
    }
}
```

De esta forma podremos **instanciar tanto el cliente**, como **establecer valores en la base de datos** al igual que **obtenerlos** con los diferentes métodos que observamos en el código anterior.

## Introducción a Big Data

**DATA:** Realmente son las **cantidades, los caracteres o símbolos** con los que se realizan **operaciones** mediante una **máquina**. Pueden **almacenarse y transmitirse** en forma de señales eléctricas y registrarse en soportes de almacenamiento **magnéticos, ópticos o mecánicos**.

Una vez conocido el concepto de datos veremos Big Data.

**Big Data:** Son realmente **datos**, pero con la diferencia que su **tamaño** es **muy grande**. Con Big Data definimos en realidad una **colección de datos** que es **gigante** y, sin embargo, **crece** cada vez más con el tiempo. Los datos que manejamos son **tan grandes y tan tediosos de administrar** que no se podía manejarlos con las herramientas que había hasta la fecha.

¿**Por qué** las bases de datos Big Data **son sistemas diferentes**?

Para poder trabajar con Big Data necesitaríamos **en principio el mismo conocimiento** como si fuésemos a trabajar con **datos de menor volumen**.

Pero algunas características como la **escalabilidad** a gran volumen, la **rapidez** de mover información y de procesarla, y las **propiedades** de algunos de los datos que se tratarán en el proceso, presentan nuevos desafíos importantes a la hora de diseñar soluciones. El objetivo de la mayoría de los sistemas de Big Data es sacar a la luz **conocimientos y conexiones de grandes volúmenes de datos heterogéneos** que no serían posibles con métodos convencionales.

## Aplicaciones Big Data



Vamos a ver a qué están orientadas las bases de datos Big Data y en qué tipo de aplicaciones se usan.

Las bases de datos Big Data, sobre todo se usan para la **web y redes sociales**, comprenden toda la información que podemos tener de los usuarios gracias a sus interacciones en distintas redes sociales, por ejemplo, Facebook, Twitter, Instagram y LinkedIn, entre otras. ¿Qué significa esto? Que hay **gran cantidad de datos, gran nivel de interacciones**.

Si queremos una **buena eficiencia** en la base de datos, con la cual podamos obtener y generar **gran cantidad de datos al mismo tiempo**, encaja perfectamente este modelo.

Otras opciones son para trabajar con **grandes transacciones**. No son datos tan sencillos de obtener y de segmentar como los anteriores, que hablábamos de las redes sociales, pero son aquellos que provienen de **redes sociales**. Estos datos son los **registros de facturación** que tenemos almacenados, así como las **llamadas**. Estos datos van a ser estructurados y no estructurados.

Otro tipo de datos que tenemos son los datos **M2M**, de las siglas **Machine to Machine**. Estos datos se obtienen a través de las tecnologías que **se conectan a otros dispositivos**. Por ejemplo, cuando un usuario accede a nuestro Wi-Fi, se conecta a nuestro dispositivo mediante Bluetooth. También se puede acceder a otros dispositivos mediante redes inalámbricas o híbrida. Estos datos son los M2M Machine to Machine.

También se almacenan en este tipo de bases de datos **datos biométricos**. Por ejemplo, el escaneo de las retinas, las huellas digitales o las tecnologías de reconocimiento facial, permiten recoger este tipo de datos biométricos. Son datos realmente interesantes. Para algunas áreas como por ejemplo la de seguridad e inteligencia. Este tipo de datos suelen almacenarse también de esta forma.

## Tipos de Big Data

Podemos encontrar Big Data de **tres formas** diferentes:

### a) Datos Estructurados

Definimos datos estructurados como toda aquella información que pueda ser **almacenada**, se pueda **acceder** y se accede, **de forma fija**.

Poco a poco conforme ha ido pasando el tiempo, los sistemas de almacenamiento han sido mejorados con distintas **técnicas innovadoras** que **mejoran el trabajo** con este tipo de datos. Pero en la actualidad existe un **problema real**, los datos crecen en tal medida que alcanzan tamaños como el **zettabyte**:

**$10^{21}$  bytes** forman 1 zettabyte o lo que es lo mismo **un billón de terabytes** forman 1 zettabyte.

Viendo estas cifras podemos entender fácilmente por qué se le da el nombre de Big Data.

### b) Datos No estructurados

Cualquier dato cuya **forma o estructura sea desconocida** se clasificará como datos no estructurados. Además de que el tamaño es enorme, los datos no estructurados plantean múltiples desafíos si nos referimos al procesamiento de los mismos. Hoy en día **muchas compañías** disponen de sistemas no estructurados donde **mezclan** colecciones que contienen **imágenes de texto, números, direcciones, ficheros media, etc.** Esto puede suponer un **problema real** si estos datos **no son procesados** y si no se sabe sacar partido de ellos.

### c) Datos Semi-estructurados

Aparentemente los datos semi-estructurados tienen la similitud de ser en parte, datos estructurados en cuyo interior poseen datos no estructurados. La realidad es que **no están planteados para ser plenamente estructurados como una tabla**.

Un **ejemplo** sería los datos **representados** como en el **formato XML**.

## Ejemplos de diferentes tipos estructuras Big Data

Vamos a **clasificar** entre los distintos tipos de **estructuras existentes de Big Data**, los siguientes tipos:

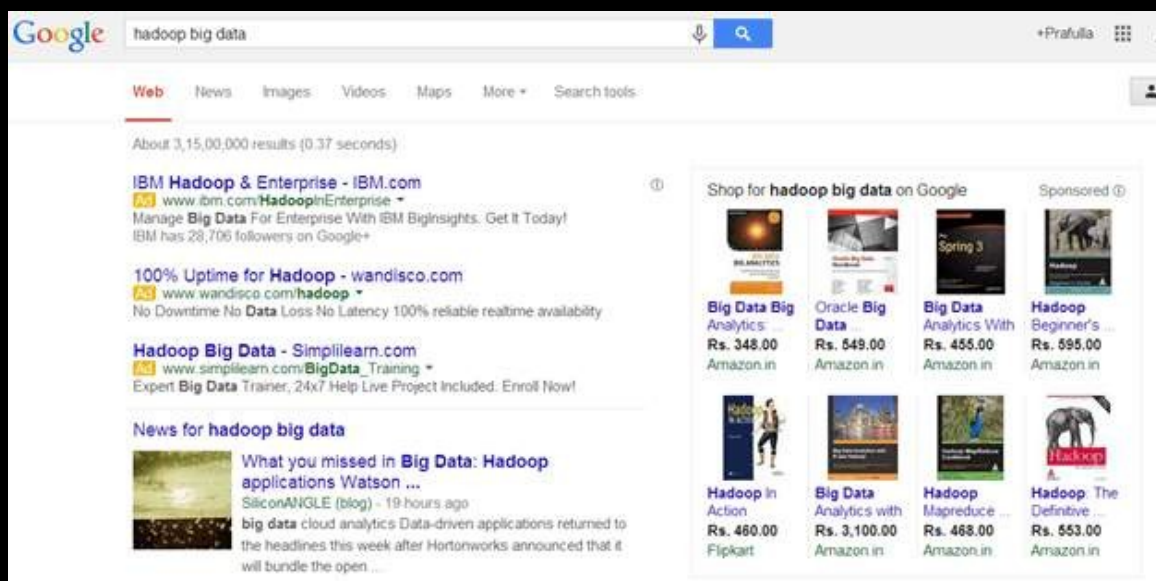
Employee_ID	Employee_Name	Gender	Departament	Salary_In_lacs
2365	Rajesh Kulkarni	Male	Finance	650000
3398	Pratibha Joshi	Female	Admin	650000
7465	Shushil Roy	Male	Admin	500000
7500	Shubhojit Das	Male	Finance	500000
7699	Priya Sane	Female	Finance	550000

```
<rec>
  <name>Prasnant Kao</name>
  <sex>Male</sex>
  <age>35</age>
</rec>
<rec>
  <name>Seema R.</name>
  <sex>Female</sex>
  <age>41</age>
</rec>
<rec>
  <name>Satish Mane</name>
  <sex>Male</sex>
  <age>29</age>
</rec>
<rec>
```

```

<name>Subrato Roy</name>
<sex>Male</sex>
<age>26</age>
</rec>
<rec>
  <name>Jeremiah J.</name>
  <sex>Male</sex>
  <age>35</age>
</rec>

```



Vamos a identificar la **tipología de Big Data** en función a los parámetros estudiados previamente.

Si nos fijamos en la **tabla** podemos ver que es un tipo de **datos estructurados**, es el ejemplo de una tabla de empleado con sus columnas bien definidas.

En el **código XML** podemos ver que tenemos un ejemplo claro de **datos semi estructurados**, un buen ejemplo de datos semi-estructurados son los ficheros XML y su estructura.

En la imagen del **navegador con el buscador de Google**, podemos observar un ejemplo de **datos no estructurados** en este caso es la respuesta de una búsqueda en Google Chrome. Podemos ver que tenemos una respuesta de **texto, etiquetas, información multimedia, etc.**

## Top 5 NoSQL

Las cinco **soluciones NoSQL más relevantes y más conocidas** en el mercado, son:

- **MongoDB**: Estamos ante la base de datos NoSQL **más conocida** a nivel de solución empresarial. Es un importante gestor de datos que almacena documentos en un **formato muy parecido a JSON**.
- **Apache Cassandra**: Base de datos **basada en columnas**, diseñada para almacenar cantidades muy grandes de datos y realizar **balanceo a distintos nodos**. A modo curiosidad es una de las herramientas que se usan en **Facebook**.
- **CouchDB**: Nació con la idea de ser la principal base de datos NoSQL de la red, pero en 2008 el proyecto pasó a formar parte del proyecto Apache Incubator. Su intención principal es la de **adaptar y compatibilizar la web** con distintos tipos de **dispositivos**. Almacena los datos en formato **Json**.
- **Redis**: Es otro importante sistema de base de datos NoSQL en este caso **clave-valor**. Es de **código abierto** y patrocinada por RedisLabs. Está basado en el almacenamiento de **tablas tipo hash**, aunque no se cierra solo a esta tipología, no obstante, es **su fuerte**.
- **Neo4J**: Fue desarrollada en **software libre**, es totalmente distinta a las 4 anteriores que hemos presentado ya que es una base de datos **orientada a grafos**, desarrollada en **lenguaje Java**.



# Instalación y conexión a base de datos Cassandra

## Instalación Cassandra

1. Instalación de Cassandra. Descarga e instala Cassandra en tu máquina desde el sitio oficial.

[https://cassandra.apache.org/\\_/download.html](https://cassandra.apache.org/_/download.html)

2. Iniciar el Servidor de Cassandra. Inicia el servidor de Cassandra ejecutando el comando correspondiente a tu sistema operativo.

3. Creación de una Keyspace

- a. Con CLI

```
CREATE KEYSPACE demo
```

```
With placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'  
and strategy_options = { replication_factor :1 };
```

- b. Con CQL

```
CREATE KEYSPACE demodb
```

```
WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor': 1};
```

4. Crear tabla

```
CREATE TABLE MiTabla (id UUID PRIMARY KEY, nombre TEXT, edad INT);
```

## Conexión con Java

Para conectar Cassandra con una aplicación Java existen varios Drivers de conectores. Uno de los conectores populares es **DataStax** Java Driver para Cassandra. Para usarlo lo añadimos a las **dependencias** Maven del proyecto:

```
<dependency>  
  <groupId>com.datastax.oss</groupId>  
  <artifactId>java-driver-core</artifactId>  
  <version>4.13.0</version><!-- Verifica la última versión en el  
  repositorio de Maven -->  
</dependency>
```

### Código Java de ejemplo:

```
import com.datastax.oss.driver.api.core.CqlSession;

import com.datastax.oss.driver.api.core.cql.Row;

import java.util.UUID;

public class CassandraConnector {

    public static void main(String[] args) {
        CqlSession session = null;
        try {
            // Conectar a Cassandra
            session = CqlSession.builder().build();

            // Ejemplo de inserción
            UUID id = UUID.randomUUID();
            String nombre = "John Doe";
            int edad = 30;
            session.execute(
                "INSERT INTO MiTabla (id, nombre, edad) VALUES (?, ?, ?)",
                id,
                nombre,
                edad
            );

            // Ejemplo de consulta
            com.datastax.oss.driver.api.core.CqlResult result = session.execute(
                "SELECT *
                FROM MiTabla"
            );
            for (Row row : result) {
                UUID resultId = row.getUuid("id");
                String resultNombre = row.getString("nombre");
                int resultEdad = row.getInt("edad");
                System.out.printf(
                    "ID: %s, Nombre: %s, Edad: %d%n",
                    resultId,
                    resultNombre,
```

```
                resultEdad
            );
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (session != null) {
            session.close();
        }
    }
}
}
```

## **Sectores donde el Big Data está más extendido y sus usos**

### **1. Banca**

- Plataformas de análisis de riesgos financieros.
- Sistemas de detección de fraudes.
- Herramientas de personalización para servicios financieros.

### **2. Salud**

- Análisis de datos clínicos para diagnósticos precisos.
- Gestión de registros médicos electrónicos.
- Investigación biomédica y farmacéutica.

### **3. Comercio Electrónico**

- Recomendaciones personalizadas para compradores.
- Análisis de patrones de compra y comportamiento del cliente.
- Optimización de la cadena de suministro.

### **4. Telecomunicaciones**

- Análisis de datos de uso para mejorar la calidad del servicio.
- Predicción de la demanda de servicios.
- Gestión de redes y prevención de fallas.

### **5. Manufactura**

- Mantenimiento predictivo de maquinaria.
- Optimización de la cadena de producción.
- Control de calidad basado en datos.

### **6. Educación**

- Análisis de datos para mejorar la retención estudiantil.
- Personalización del aprendizaje.
- Evaluación del desempeño de los estudiantes.

### **7. Energía**

- Monitoreo y gestión de la red eléctrica.
- Optimización de la distribución de energía.
- Análisis de datos para eficiencia energética.

### **8. Medios y Entretenimiento**

- Recomendaciones de contenido personalizado.
- Análisis de audiencia y comportamiento de visualización.

- Producción de contenido basada en datos.

El uso de Big Data se extiende a través de diversos sectores, desde la banca y la salud hasta la manufactura y la educación. Las aplicaciones varían, desde análisis de riesgos financieros y diagnósticos médicos hasta recomendaciones personalizadas y optimización de la cadena de suministro. La capacidad de extraer conocimientos significativos de grandes conjuntos de datos está transformando la forma en que las organizaciones toman decisiones y ofrecen servicios.

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

**MongoDB**  
**Instalación**  
**Operaciones**

## Base de datos MongoDB

Para realizar un primer acercamiento a esta base de datos, podríamos destacar las siguientes características:

- Es una base de datos NoSQL **orientada a documentos**.
- La estructura de los documentos es en **formato JSON**. **Internamente** los documentos son almacenados en **formato BSON**. BSON son los binarios de JSON.
- Al tener documentos más ricos, se reduce el I/O (input/output) sobre la base de datos. **No existe la necesidad de hacer Joins**.
- Permite **operaciones CRUD** (Create, Read, Update, Delete) con una sintaxis parecida a JavaScript.
- Proporciona **replicación y alta disponibilidad** a través de **Replica Sets**.
- También dispone de **balanceo y escalado horizontal** usando Sharding. El **balanceo** de los datos se realiza **automáticamente**.
- Ofrece un mecanismo de **procesamiento masivo** de datos a través de operaciones de **agregación**.
- El grueso de los **datos** reside **en memoria**, por lo que las lecturas y escrituras son muy rápidas.
- Permite también crear **colecciones de tipo circular** de tamaño **fijo**, y mantiene el orden según se han ido insertando los datos (Capped Collections o Colecciones limitadas).
- Tienen **múltiples motores de almacenamiento** diferentes, **importante**
  - **nmap**,
  - **WiredTiger**,
  - **In-memory**
  - **Encrypted**.
- Es **enorme**, forma parte de **Big Data**.

### Sharding (fragmentación)

El sharding es una expresión que en su propia traducción nos hace referencia a un **particionado de disco**. Concretamente significa **fragmentación**. Es una **técnica** que utilizan bases de datos como Mongo DB para **gestionar automáticamente** la **carga** en sus servidores. Van distribuyendo los datos en diferentes “**shards**” o fragmentos, es decir, conjuntos de servidores que **guardan los diferentes datos**.

## MongoDB el teorema de CAP (importante)

El teorema de **CAP** o también de **Brewer**, sostiene, que en sistemas distribuidos **no es posible garantizar** al completo la consistencia, la disponibilidad y tolerancia:

- **Consistencia:** La **información** que obtenemos a la hora de realizar una consulta debe ser **siempre la misma**. Debe de dar siempre el mismo resultado.
- **Disponibilidad:** Todos los clientes deben de **poder realizar las operaciones** de lectura y escritura, **aunque** un nodo se haya **caído**.
- **Tolerancia a particiones:** Los sistemas, como vimos en la unidad anterior, pueden estar divididos en **particiones distribuidas** en distintos puntos. Esta propiedad consiste en, a pesar de esta división, asegurar el **funcionamiento**.

Según el teorema de CAP, podemos **clasificar** todas las **bases de datos** según las características comentadas anteriormente. **No todas cumplen** los mismos puntos del teorema.

- Las bases de datos más cercanas al vértice AP (Availability Partition tolerance):
  - ➔ aseguran la **disponibilidad y tolerancia** a particiones,
  - ➔ pero **no la consistencia**, al menos en su totalidad.
    - Algunas de ellas a través de la **replicación y verificación**, sí consiguen **parte** de consistencia.
    - **Dynamo DB, Couch DB, Cassandra, Infinite Graph.**
- Aquellas que su vértice este más cercano a CP (Consistency Partition tolerance):
  - ➔ estarán del lado de la **consistencia y tolerancia** a particiones.
  - ➔ **Sacrifican la disponibilidad** al replicar los datos a través de nodos.
    - **MongoDB, Hbase, Redis.**
- Por último, las bases de datos más cercanas al vértice CA (Consistency Availability):
  - ➔ poseerán más **consistencia y disponibilidad**
  - ➔ **dejando** un poco de lado la **tolerancia a particiones**.
    - Este problema se solucionará **replicando** los datos.
    - **BBDD relacionales (MySQL, Oracle, SQL Server), Neo 4J.**

Según el teorema CAP podemos encontrar a **MongoDB en el vértice CP**.



## Casos de uso y términos

MongoDB es un producto de **propósito general** y es muy útil para **múltiples casos de uso** tales como:

- **CMS** (Content Management System), **Aplicaciones móviles**.
- **Gaming**.
- **E-commerce**.
- **Business intelligence**.
- **Analytics**.
- Proyectos **Big Data**.
- **Web Caché**.

A continuación, vamos a exponer algunos **términos utilizados en MongoDB** y su correspondencia en el mundo relacional:

## Comparativa relacional - MongoDB

### Relacional → MongoDB

Base de datos → Base de datos

Tabla → Colección

Fila → Documento

Índice → Índice

Insert → Insert

Select → Find

Update → Update

Delete → Remove

## Comparativa relacional y mongo

Estamos haciendo una introducción básica de la base de datos MongoDB. En la tabla anterior comparamos las bases de datos relacionales y bases de datos orientadas a objetos que trabajan con documentos.

Hemos visto que este tipo de base de datos, como es MongoDB, casi siempre (o, digamos, a nivel comercial), vamos a ver su objetivo en salidas como:

- gaming,
- aplicaciones de e-commerce,
- algunas de analíticas,
- de business intelligence,
- aplicaciones que están conectadas a nivel de compras con business intelligence como hemos dicho
- y, sobre todo proyectos de Big Data; aplicaciones que muevan ficheros de información muy grandes y que tengan la facilidad de almacenarse bien con este sistema.

Pero algo que venimos a ver aquí en la tabla es que, si vemos la diferencia entre las bases de datos relacionales y MongoDB, pues todo parte de un origen y realmente hay diferencias, pero con los conocimientos previos que solemos tener de las bases de datos relacionales, no nos va a costar mucho adaptarnos a MongoDB, por ejemplo. Ya veréis.

Vamos a ver a continuación que, cuando en una **tabla** de una base de datos relacional (o sea, cuando en una base de datos relacional tenemos una tabla), aquí tendremos su símil, sería una **colección**. Porque en MongoDB tenemos una base de datos y dentro de esa base de datos tenemos diferentes colecciones. Dentro de la colección tenemos **documentos** y, en la base de datos relacional, tendremos **filas**.

El resto de cosas, como podemos ver, índices seguimos teniendo igual. Los inserts se hacen muy parecidos. En la base de datos relacional, tenemos el **select** y aquí tenemos el **find**, con algunas especificaciones que veremos más adelante. Update, tenemos update, y **delete**, en vez de delete, tenemos **remove**.

Es una forma muy fácil de aprender en MongoDB porque, digamos, que los conceptos de la base de datos relacional ya los tenemos, entonces adaptarnos un poco a lo que nos pide esta base de datos orientada a objetos u **orientada a colecciones**, en este caso.

## Documentos

Veamos algunas características relacionadas con el concepto de documento:

- Cada **entrada de una colección** es un documento.
- Son estructuras de datos compuestas por campos de clave/valor.
- Los documentos tienen una estructura similar a objetos JSON.
- Los documentos se corresponden con tipos de datos nativos en los lenguajes de programación.
- En un documento es posible embeber otros documentos o arrays.
- Se tiene un **esquema dinámico** que permite polimorfismo de manera fluida.
- Las operaciones de escritura son solo atómicas a nivel de documento.

A continuación, mostraremos un ejemplo:

### Ejemplo documento Mongo

```
{
  "id": ObjectId("5457a502e308f720d8999e97"),
  "Nombre": "Francisco",
  "Apellidos": "Fernandez Rioja",
  "Edad": 30,
  "Aficiones": "{ \"Comics\" : null, \"Deportes\": [\"squash\",\"natacion\"]
    },
  \"Empresa\": \"XXXSA\",
  \"Cargo\": \"MongoDB DBA\",
  \"Tecnologias\": [\"Openstack\", \"Openshift\", \"MongoDB\"],
  \"Proyectos\": \"{ \"Openstack\": [\"Cliente1\", \"Cliente2\"],
    \"Openshift\": [\"Cliente4\" ]
  }
}
```

## Instalación MongoDB

Intentaremos instalar **instalar en un servidor Linux**. (Ubuntu, Debian)

En la web de MongoDB encontraremos los enlaces de descarga en diferentes formatos. Seguiremos las instrucciones que nos indica en sus tutoriales oficiales:

<https://docs.mongodb.com/guides/server/install/>

1. En primer lugar, nos descargaremos los ficheros binarios desde [MongoDB Download Center](#).

2. Extraeremos los ficheros descargados con:

```
tar -xvzf <tgz file>
```

3. Copiaremos la carpeta que acabamos de extraer a la localización en la que hayamos elegido ejecutar MongoDB con el comando “cp”.

4. Cargaremos en la variable de entorno PATH nuestra instalación. Los ficheros binarios en MongoDB se encuentran en la carpeta bin. Añadiremos a nuestro fichero .bashrc la siguiente línea:

```
export PATH=<mongodb-install-directory>/bin:$PATH
```

5. Antes de arrancar MongoDB podemos crear un directorio donde el **proceso “mongod” escribirá los datos**. Por defecto, el proceso usa la **ruta /data/db**. Si creamos otro directorio distinto deberemos de especificarlo en el arranque. Para crear el directorio por defecto:

```
mkdir -p /data/db
```

6. Antes de ejecutar el proceso habría que asegurarse que el usuario con el que se va a ejecutar “mongod” tiene los **permisos de escritura en el directorio**.

7. Para arrancar la base de datos MongoDB con la ruta por defecto mencionada anteriormente, habría que ejecutar el proceso “mongod”:

```
mongod
```

Si no se usa el directorio por defecto, habría que aplicar:

```
mongod -dbpath <directorio>
```

8. Finalmente, habría que verificar que MongoDB ha arrancado correctamente, comprobando que se ha mostrado la siguiente línea:

```
[initandlisten] waiting for connections on port 27017
```

Esta explicación es muy pobre...

Este parece que instala mongo con .deb y luego la shell para el comando mongo:

<https://www.youtube.com/watch?v=L5bRKSBflwM> en debian

Este instala en server debian, parece más completo pero no se si funcionará por lo del server debian: este creo que es el mejor, instalar antes debian server

<https://www.youtube.com/watch?v=ls2mpE9W6FQ>

En este se instala en windows, y se parece al temario, con la variable path

[https://www.youtube.com/watch?v=\\_C6AuXNySqq](https://www.youtube.com/watch?v=_C6AuXNySqq)

otra instalación que parece buena, y casi funciona:

<https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-debian/>

primero instalé esto:

FUNCIONÓ ESTA pero no conecta a la bbdd:

<https://www.ochobitshacenunbyte.com/2015/01/12/mongodb-en-gnu-linux/>

para manejo de mongo esta bien esto:

<https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/mongodb-en-debian/>

Esta instalacion funciona:

<https://medium.com/@hackthebox/installing-mongodb-7-0-community-edition-on-debian-12-bookworm-a-comprehensive-guide-19906ddb47ad>

y cuando probelam con libssl1.1:

<https://askubuntu.com/questions/1403619/mongodb-install-fails-on-ubuntu-22-04-depends-on-libssl1-1-but-it-is-not-installed>

Una vez que está todo hecho:

```
sudo systemctl enable mongod --now
```

se queda parado activo...

disable para apagar

en otra terminal:

```
sudo systemctl start mongod
```

y comprobamos:

```
sudo systemctl status mongod
```

y está activo!

## Ficheros binarios de MongoDB

El paquete de MongoDB contiene algunos ficheros binarios. Se usan **para arrancar el servidor** de la base de datos y para acceder al shell de la misma. Algunos de ellos son:

- **mongod:**

Es el **servicio principal** de MongoDB. Maneja los **accesos a los datos**, las **peticiones** de datos, y ejecuta tareas de **mantenimiento** en background. Su fichero de configuración es "mongod.conf".

- **mongo:**

Es la **shell interactiva** de MongoDB. Aporta un **entorno funcional** completo para ser usado con la base de datos.

- **mongos:**

Es un servicio propio del modo de **despliegue Shard**. Su función es la de enrutar las peticiones de la capa de aplicación y determinar la **ubicación de los datos en los diferentes shards** del despliegue.

- **mongodump:**

Es una utilidad para crear un **export binario** del contenido de una base de datos. Podemos considerar MongoDB como una herramienta más para realizar copias de seguridad. Podremos usar esta herramienta **contra "mongod" o "mongos"** teniendo en cuenta que **"mongod" podrá estar arrancado o parado** indistintamente.

- **mongorestore:**

En conjunción con mongodump, se utiliza para **restaurar los respaldos** realizados con **mongodump**.

- **mongooplog:**

Es una herramienta que permite hacer **"polling" del oplog de un servidor remoto**, y aplicarlo sobre el servidor local. Esta utilidad la podemos usar para realizar cierta clase de **migraciones** en tiempo real, donde se requiere que el **servidor fuente se mantenga Online** y en funcionamiento.

## Herramientas exportación / importación

A continuación, vamos a ver las herramientas que disponemos para la exportación e importación de datos:

**bsondump:**

Convierte ficheros **BSON a algún formato legible** por humanos, incluido a JSON. Se trata de una **herramienta de análisis**, en ningún caso debe ser utilizada para otro tipo de actividades.

**mongoexport:**

Utilidad que permite **exportar** los datos de una **instancia de MongoDB** en formato **JSON o CSV**. En conjunción con mongoimport son útiles para hacer backup de una parte bien definida de los datos de la BBDD MongoDB o para casos concretos de inserción de datos.

**mongoimport:**

Utilidad que permite **importar** los datos de una instancia de MongoDB desde ficheros con **formato JSON o CSV**.

**mongofiles:**

Utilidad que permite manejar ficheros en una instancia de MongoDB con objetos GridFS, desde la línea de comandos. En Replica Set sólo podrá leer desde el primario.

## Herramientas análisis

También disponemos de algunas herramientas para el análisis de **performance y actividad**:

`mongoperf`:

Utilidad para **comprobar el performance I/O** de forma independiente a MongoDB.

`mongostat`:

Utilidad que proporciona una rápida visión del **estado actual de los servicios mongod y mongos**. Es similar a la utilidad `vmstat`.

`Mongotop`:

Proporciona un método para **trazar el tiempo que una instancia** de MongoDB emplea en las **operaciones de Lectura/Escritura** de datos. Proporciona **estadísticas** a nivel de colección, por defecto, **cada segundo**.



## Shell de MongoDB

Para interactuar con esta base de datos utilizamos la **shell mongo**, que básicamente:

- Es una **shell interactiva en JavaScript**.
- Permite **ejecutar scripts escritos en JavaScript** para manipulación de datos, ejecución de comandos en la base de datos, aplicación de índices, etc.
- La shell puede ser utilizada tanto para la **visualización** de datos, como para la **administración** de la base de datos, sea **Standalone, Replica Set o Sharding**.

Para trabajar con la shell, en primer lugar, nos conectamos a la instancia que hemos levantado anteriormente ejecutando el **comando mongo**.

Una vez obtenemos el prompt, ejecutaremos algunos comandos de ejemplo:

### Show dbs

Con este comando podremos **ver las diferentes bases de datos** que tenemos.

### use miBD

Mediante **use miDB**, estaremos **creando una base de datos y posicionándonos en ella**.

```
db.createCollection("holaMundo")
```

Con createCollection() **creamos una colección nueva (tabla en relacional)**, a continuación, vamos a insertar datos.

```
db.holaMundo.insert({"Nombre" : "Ernesto", "Apellido" : "Perez", "Edad" : "45"})
```

De esta forma realizaremos **inserciones en nuestra nueva colección** creada previamente.

Para **ver todos los registros** de nuestra colección:

```
db.holaMundo.find()
```

## Comandos

En la **shell de Mongo** podremos encontrar una serie de comandos que nos facilitarán la realización de alguna tarea o mostrar información sobre la base de datos.

Vamos a revisar **algunos de los más destacados**:

### **Helper** - Descripción

*help* - Muestra ayuda general.

*db.help()* - Muestra ayuda sobre los comandos de ejecutables sobre BBDD.

*db.<collection>.help()* - Muestra ayuda sobre comandos de ejecutables sobre colecciones.

*Show dbs* - Muestra las BBDD del servidor.

*db* - Devuelve el nombre de la BBDD donde nos encontramos posicionados.

*show collections* - Muestra las colecciones contenidas en la BBDD donde estamos posicionados.

*use <db>* - Nos posicionamos sobre la BBDD db.

*show users* - Muestra los usuarios sobre la BBDD actual.

*load("<ruta\_script>")* - Carga en la sesión actual el script contenido en la ruta ruta\_script.

*It* - Itera el cursor sobre el que se haya hecho una query.

## Operaciones en el Shell

En esta ocasión, abordaremos el tema de la shell de MongoDB, la cual desempeña un papel fundamental en la interacción con la base de datos. La shell es el entorno donde proporcionamos instrucciones para realizar diversas operaciones, como **escritura, lectura, actualización de datos**, entre otras. Además, nos brinda la capacidad de realizar importaciones y exportaciones de datos, así como otras operaciones diversas.

Una **alternativa gráfica** para administrar la información de la base de datos es **Robomongo**, una aplicación que ofrece una interfaz visual para este propósito. Sin embargo, la **verdadera potencia se encuentra en la shell**.

Dentro de la shell, encontramos diversos **comandos esenciales**. Entre ellos, el comando ``help`` nos proporciona asistencia general, mientras que ``db.help`` nos muestra información específica sobre los comandos disponibles en la base de datos. Si especificamos el nombre de una colección, como en ``db.nombreColeccion.help``, obtenemos una **lista de comandos aplicables** a esa colección en particular.

A continuación, algunos **comandos básicos** que podemos ejecutar en la shell:

1. ``help``: Muestra la ayuda general.
2. ``db.help``: Proporciona ayuda sobre los comandos disponibles en la base de datos.
3. ``show dbs``: Lista las bases de datos disponibles.
4. ``db``: Devuelve la **base de datos actual**.
5. ``show collections``: Muestra la **cantidad de colecciones** en la base de datos actual.
6. ``use db``: **Mueve o crea una base de datos y se sitúa** en ella.
7. ``show user``: Muestra los usuarios de la base de datos actual.

Estos comandos permiten obtener información y desplazarse entre bases de datos. Para acceder a una guía completa de comandos, se recomienda visitar los documentos en ``docs.mongodb.com``, donde el manual de referencia proporciona detalles sobre todas las colecciones y comandos disponibles.

Entre los comandos avanzados, encontramos ejemplos como ``db.collection.count``, que cuenta la cantidad de documentos en una colección, o ``db.collection.createindex``, que facilita la creación de **índices en la colección**.

En resumen, MongoDB ofrece una amplia variedad de opciones que pueden explorarse y aprenderse con detenimiento en la sección correspondiente del sitio web oficial de MongoDB.

A parte de la shell de mongoDB, como acabamos de decir, tenemos otra opción para poder administrar la información de nuestra base de datos mongo. Existen varias **herramientas gráficas** que también nos permiten acceder a nuestra BBDD y administrarla de manera más intuitiva. Una de las más conocidas es Robomongo.

Compass es fácil y funciona.

## Operaciones CRUD

Las operaciones CRUD no son más que las habituales operaciones de Insertar, leer, actualizar y borrar. De ahí sus siglas (**Create/Read/Update/Delete**).

A continuación, mostraremos una tabla con las **distintas operaciones en las diferentes bases de datos y su equivalencia**:

### Operación - Mongo - SQL

Create - Insert / save - INSERT

Read - Find / findOne - SELECT

Update - Update / findAndModify / save - UPDATE

Delete - Remove / drop - DELETE

### Inserción de datos

Todo documento insertado en MongoDB tiene un campo `_id`, que **identifica unívocamente el documento**. Genera **valor por defecto**.



```
db.users.insert (
{
  name: "sue",
  age: 26,
  status: "A"
})
```

The diagram illustrates the MongoDB `insert` command with annotations. A purple arrow points from the text 'collection' to the `users` part of `db.users.insert`. Three purple arrows point from the text 'field: value' to the `name: "sue"`, `age: 26`, and `status: "A"` fields within the document object. A large curly brace on the right groups the entire document object `{ name: "sue", age: 26, status: "A" }` and is labeled 'document'.

Base de datos, punto, colección, punto, insert, abro paréntesis y abro corchete. A continuación el campo seguido de dos puntos, y el valor entre comillas dobles si es string, si es número es sin comillas. Se cierra el corchete y el paréntesis al terminar el insert.

## Consulta de datos

Este es un ejemplo claro de cómo se consultarían datos en nuestra base de datos Mongo:

```
db.users.find(           ← collection
  { age: { $gt: 18 } },   ← query criteria
  { name: 1, address: 1 } ← projection
).limit(5)               ← cursor modifier
```

Base de datos, punto, colección, punto, find. Abrimos paréntesis y corchete, la query criteria se forma con el campo y dos puntos, seguido de el signo del dólar, el criterio o filtro gt y el valor (18). Añadimos una coma y la projection se forma igual, entre corchetes, y en el interior el campo y valor separados por dos puntos, cerrando a continuación el paréntesis del find, poniendo un punto y añadiendo un cursor modifier, que tendrá entre paréntesis un valor:

```
collection → db. users. Find(
query criteria → { age: { Sgt 18 } },
projection → { name: 1, address: 1 }
cursor modifier → ). Limit(5)
```

## Actualizar información

Podemos observar cómo se realizaría el Update:

```
db.users.update(           ← collection
  { age: { $gt: 18 } },     ← update criteria
  { $set: { status: "A" } }, ← update action
  { multi: true }           ← update option
)
```

Base de datos, punto, colección, punto, update, abre paréntesis (abre corchete y aquí va el update criteria, que será el campo seguido de dos puntos, abre corchete, signo de dólar con el criterio gt y seguido de dos puntos, y el valor (18), cerrando los dos corchetes, coma, y abriendo corchete para el update action, con el signo del dólar y set, dos puntos, abre corchete y otro campo (status) seguido de dos puntos y el valor ("A" entre comillas en este caso), cerrando los dos corchetes, y una coma, y terminamos abriendo corchete para el option update, que será: multi seguido de dos puntos y el valor a true. Cerramos el corchete del update option y el paréntesis del update de la colección.

```
collection → db.users.update(  
  update criteria → { age: { $gt: 18 } },  
  update action → { $set: { status: "A" } },  
  update option → { multi: true }  
)
```

## Borrado de documentos

```
db.users.remove(           ← collection  
  { status: "D" }         ← remove criteria  
)
```

Base de datos, punto, colección, punto, remove y abre paréntesis. Ahora el remove criteria entre corchetes, que será un campo (status) seguido de dos puntos, y el valor entre comillas ("D"). Por último cerramos el paréntesis del remove.

```
collection → db.users.Remove(  
  remove criteria → { status: "A" }  
)
```

## Ejemplo de Insertar y borrar

Vamos a insertar 2 documentos cuyo equipo sea:

- RealMadrid, blanca, Madrid.
- FCBarcelona, azulgrana, Barcelona.

La inserción se debe realizar en la **colección "equipos"** que tiene la siguiente estructura:

```
{{"Nombre" : "X", "Camiseta" : "X", "Ciudad" : "X"}}
```

Estudiadas las sentencias CRUD básicas para el manejo de datos realizaremos la inserción de datos con el **comando insert**.

Suponiendo que estamos ya **situados en la base de datos** adecuada, deberemos recordar y aplicar, cual es la sentencia para agregar datos, en este caso insert.

Para ello, en primer lugar, pondremos la **palabra clave "db"**, a continuación, la **colección donde vamos a insertar los datos**, justo después la sentencia que en este caso es **insert**, y, por último, agregaremos los datos.

Las dos inserciones quedarían de la siguiente forma:

```
db.equipos.insert({"Nombre" : "RealMadrid", "Camiseta" : "blanca", "Ciudad" : "Madrid"}) Código. 7  
Insert 1
```

```
db.equipos.insert({"Nombre" : "FCBarcelona", "Camiseta" : "azulgrana", "Ciudad" :  
"Barcelona"})
```

## Ejemplo de creación de colección Empleados y consulta en MongoDB

Una vez levantado nuestro servicio Mongod, ejecutando el **comando “mongo” sobre el shell** de nuestra base de datos, realizaremos las siguientes operaciones:

1. **Use** MiEmpresa: cuando ejecutemos este comando, estaremos **creando** el almacén de datos y a su vez estaremos **entrando en él**. Podremos usar el comando `show dbs`, para comprobar que **nuestra base de datos aparece** entre las creadas.
2. Diseñaremos la colección Empleados de la siguiente **forma**:

**({"Nombre": "XXX", "Edad" : "xxx", "Antigüedad" : "xxx", "Especialidad" : "xxx"})**

Una vez tenemos claro el modelo, el próximo paso será realizar la **creación de la colección Empleados**:

**db.createCollection("Empleados")**

3. Por último, se requiere **realizar una consulta** en nuestra base de datos imaginando que hubiéramos insertado ya una cantidad de registros en dicha colección, nos dispondríamos a realizar la consulta con la palabra clave **find**.

**db.Empleados.find({Edad: {\$gt : 40}})**

Con el comando `find` podremos realizar distintas consultas en la base de datos Mongo y directamente en nuestras colecciones.

Para mostrar aquellos empleados cuya edad sea **superior a 40** años, podemos ver que se usa **“\$gt”**. En este caso hemos usado el **operador Greater Than** que nos mostrará los resultados mayores que la cantidad que indiquemos, en este caso 40.

<https://docs.mongodb.com/>

<https://www.paradigmadigital.com/>



ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

**MongoDB**

**Índices**

**Roles y usuarios**

**Exportación e importación**

## Índices en MongoDB

El concepto de índice que maneja MongoDB es el mismo que el que se maneja en otros repositorios de información.

### Concepto índice

Un índice permite que una **consulta devuelva información** de forma **más efectiva**. Los índices están relacionados a tablas, o en el caso de MongoDB, a **colecciones específicas o campos** de las mismas con una o más **claves definidas**.

### Características de los índices

- Ofrecen grandes **mejoras en** las operaciones de **lectura**.
- Pueden **penalizar** las operaciones de **escritura**.
- Se almacenan **en memoria**.
- **Por defecto** se crea un índice único sobre el **campo \_id**.
- No sólo se indican los campos del **índice** sino el **orden** del mismo.
- Los operadores de **negación no utilizan índices**.
- Los índices los emplean las **queries** y las “**agregaciones**”.

A continuación, veremos la sintaxis para la creación de un índice en MongoDB:

#### Sintaxis índice

```
db.collection.createIndex(  
  { <field_1>: <index type>,  
    <field_2>: <index type> ... } {<options>})
```

## Tipologías de los índices en MongoDB

Los índices en MongoDB pueden ser:

**Monoclave:** Sólo indexan por un campo de los documentos de la colección.

Sintaxis índices monoclave

```
db.collection.createIndex( { <field>: <-1|1> } )
```

Características:

- Sólo afectan a un campo de búsqueda.

Indicar:

-1 si el índice es ascendente,

1 si el índice es descendente.

- Índices sobre entidades embebidas:

Sólo se produce un acierto en el índice si las **entidades son exactamente iguales**, incluyendo orden.

**Compuestos:** Indexan por varios campos de los documentos de la colección.

El -1 indica ordenación descendente y la 1 ordenación ascendente.

Sintaxis índices compuestos

```
db.collection.createIndex( { <field_1>: <-1|1>,<field_2>: <-1|1> } )
```

**Hashed:** Indexan de forma clave-valor.

Son índices en los que un elemento del índice hace referencia a un **campo** que tiene **valores de array**.

Pueden ser índices de **un único campo o compuestos**.

Se crean **tantas entradas** del índice **como valores** contenga el array.

**Solo uno** de los elementos del índice puede tener **valores de array**.

**Text:** Índices de texto.

**Geoespaciales:** Indexan por coordenadas espaciales

**Unique (o de tipo único):** Sólo puede haber **una correspondencia** entre entrada del índice y documento.

Indica que **cada valor del índice** debe corresponder con **un único valor**.

Se puede utilizar tanto para índices de un solo campo o multicampo.

Sintaxis índice únicos

```
db.collection.createIndex({ <field>: <index_type> }, {unique: <true|false>} )
```

**Sparse: No indexa** campos con valores **nulos**.

Es un índice de valores únicos que admite valores nulos. Los valores nulos **no se introducen en el índice**. Una búsqueda que utilice este índice no devolverá documentos que no estén en el índice:

Sintaxis sparse

```
db.collection.createIndex( { <field>:<index_type> }, {sparse: <true|false>} )
```

**Background:** Más **lento**, pero **no bloquea a los lectores/escritores**.

**TTL: Elimina documentos** de la colección cuando:

- pasa cierto tiempo,
- alcanza la fecha de expiración.

**Partial:** Indexa aquellos documentos **que cumplan una condición**.

## Operaciones con los índices

En MongoDB podemos destacar las siguientes operaciones con los índices:

- **Covered query:** Es una **consulta** que se resuelve **contra un índice sin consultar datos de la colección**. (Como ya está guardado en memoria, consultamos directamente esos datos, sin consultar los datos de la colección).

Ejemplo covered query

```
db.users.find( { score: { "$lt": 30 } }, { score: 1, _id: 0 } )
```

- **Borrado de índice:** El borrado de índice lo realizaríamos de la siguiente forma:

Sintaxis borrado índice

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

- **Regeneración índice:** Para volver a crear un índice ya creado previamente usaremos. (la explicación del profesor fue: Regenera todos los índices de la colección):

Sintaxis Reindex

```
db.accounts.reIndex()
```

- **Listado de índices:** Para listar los índices **asociados a una colección** usaremos:

Sintaxis listado índices

```
db.accounts.getIndexes()
```

## Documentación oficial

En primer lugar, quiero presentarles la página de la documentación oficial web de MongoDB:

<https://www.mongodb.com/docs/>

Aquí encontramos un árbol que abarca toda la documentación, y gran parte de la teoría que hemos utilizado se basa en esta documentación oficial, la cual recomiendo completamente por su exhaustividad. Aunque está en inglés, podemos traducirla fácilmente, lo que también nos resulta útil para practicar el idioma.

Como podemos observar, en el tema anterior exploramos el shell, cubriendo la mayor parte o la totalidad de la teoría sobre su uso, los requisitos necesarios, cómo iniciar los servicios de Mongo y cómo configurar el shell. Contamos con una abundancia de información relacionada con el tema actual, incluyendo índices. Aquí encontramos explicaciones gráficas sobre qué es un índice, los tipos de índices que encontraremos, y cómo crear un índice, aspecto que abordaremos en esta unidad.

<https://www.mongodb.com/docs/manual/>

Además, disponemos de diversos tipos de índices para explorar, acompañados ocasionalmente de ejemplos gráficos. Esta web de [mongodb.com/docs](https://www.mongodb.com/docs/) se presenta como una herramienta valiosa para contrastar y ampliar la información, ya que todo lo que se presenta aquí constituye la documentación oficial sobre el código de Mongo.

Simplemente quiero mostrarles esta herramienta. Aquí pueden encontrar secciones sobre introducción e instalación, lo que les permite contrastar lo que hemos estado viendo y aprender nuevas cosas, como transacciones, agregaciones y la operación CRUD, que también se trató en el tema pasado. Les dejo aquí la herramienta para que le echen un vistazo y la utilicen como refuerzo conceptual.

## Ejemplo de creación índices

Se requiere realizar la creación de diferentes índices para las siguientes colecciones:

1. Para la colección de documentos **factories**, escribir un **índice de tipo monoclave** al campo **metro**:

```
{  
  metro: { city: "New York", state: "NY" },  
  name: "Giant Factory"  
}
```

2. Para la misma colección anterior se requiere crear un **índice compuesto** para poder mostrar con el **campo metro ascendente** y con el **campo name descendente**.

3. Para la colección **vehículos**, se ha insertado el siguiente documento:

```
{  
  vehiculo: { marca: "Renault", modelo: "c3" },  
  ruedas: null  
}
```

Se requiere crear un índice de tipo **sparse** que ordene por el **campo vehículo** y el segundo campo **si es nulo, sea true**.

Una vez estudiados los diferentes tipos de índices veremos cómo podemos resolver los diferentes ejemplos.

1. Para el primer ejercicio la creación del índice de tipo monoclave será el siguiente:

```
db.factories.createIndex( { metro: 1 } )
```

2. Para el segundo ejercicio realizaremos un índice compuesto:

```
db.factories.createIndex({a:1, b:-1})
```

3. Para el último ejemplo realizaremos un índice sparse o nulo:

```
db.vehiculos.createIndex( { a: 1 }, {sparse:true} )
```

## Creación de usuarios y roles

Mongo DB emplea el control de acceso basado en roles (RBAC), para determinar el acceso de los usuarios. A un usuario se le otorgan uno o más **roles** que **determinan el acceso o los privilegios del usuario**, a los **recursos** de nuestra base de datos MongoDB, y a las **acciones** que dicho usuario puede realizar.

Un usuario debe tener solo el **conjunto mínimo de privilegios** necesarios para garantizar un **sistema de privilegios mínimos**. Cada **aplicación y usuario** de un sistema Mongo debe asignarse a un **usuario distinto**.

Este aislamiento de acceso facilita la **revocación del acceso** y el **mantenimiento** continuo del usuario. Para crear un usuario en Mongo DB podemos ejecutar el comando:

**db.createUser()**

Si el usuario es creado no devolverá nada, si el **usuario existe** previamente, nos devolverá un **error de duplicado**.

Veamos a continuación la sintaxis:

Sintaxis create user

```
{
  user: "<name>",
  pwd: passwordPrompt(), // Or "<cleartext password>"
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  authenticationRestrictions: [
    {
      clientSource: [ "<IP>" | "<CIDR range>", ... ],
      serverAddress: [ "<IP>" | "<CIDR range>", ... ]
    },
    ...
  ],
  mechanisms: [ "<SCRAM-SHA-1|SCRAM-SHA-256>", ... ],
  passwordDigestor: "<server|client>"
}
```



En el código anterior, podemos observar:

- **user**: String que indica el **nombre** del nuevo usuario.
- **pwd**: String donde indicaremos la **contraseña**.
- **customData**: Documento opcional.

En este campo indicaremos **información adicional** como puede ser el **nombre completo** del usuario o el **ID del empleado**.

- **roles**: En este **array** indicaremos los permisos que queramos otorgarle al nuevo usuario. **Vacío si no lo queremos dotar de permisos**.
- **authenticationRestrictions**: Es un **array opcional**. Se establecerá un **rango de direcciones IP** desde las cuales dicho usuario se **podrá conectar**. (Se suele usar para que los empleados no se conecten fuera de la oficina).
- **mechanisms**: **array opcional** que especifica si dicho usuario tiene **permisos para crear credenciales SCRAM**.
- **passwordDigestor**: **String opcional** que indica si el servidor o el cliente **traducen la contraseña**.

## Acceso de control basado en roles (RBAC)

Como hemos adelantado en el apartado anterior, MongoDB emplea el control de acceso basado en roles (RBAC) para controlar el acceso a un sistema MongoDB, además, a un usuario se le otorgan uno o más roles que determinarán el acceso del usuario a los recursos y operaciones de la base de datos, lo que quiere decir, que **fuera de las asignaciones de funciones**, el usuario **no tiene acceso** al sistema.

MongoDB no habilita el control de acceso basado en roles por defecto. Se puede **habilitar** la autorización mediante una de estas configuraciones:

- “—auth”.
- “security.authorization”.

Si se habilita la **autenticación interna también** se habilita la del **cliente**. Una vez que se habilita el control de acceso, los **usuarios deben autenticarse**.

Un rol otorga privilegios para realizar **acciones específicas sobre nuestros recursos**.

Cada privilegio:

- se **especifica explícitamente** en el rol
- o **se hereda** de otro rol.

Un privilegio se aplica **sobre un recurso** especificado, y sobre las **acciones permitidas de ese mismo**.

Un recurso puede ser:

- una **base de datos**,
- una **colección**,
- un **conjunto** de ellas,
- o incluso el **clúster**.

Si el recurso es el **clúster**, las acciones afiliadas afectan al **estado del sistema** en lugar de a una base de datos o colección determinada.

Una **acción** especifica la **operación permitida sobre el recurso**.

Podemos usar el comando “**rolesInfo**” para **visualizar los privilegios de un rol** con el parámetro “**showPrivileges**” y “**showBuiltinRoles**” a **true**.

Sintaxis rolesInfo

```
{  
  rolesInfo: { role: <name>, db: <db> },  
  showPrivileges: <Boolean>,  
  showBuiltinRoles: <Boolean>,  
  comment: <any>  
}
```

## Ejemplo de creación de rol

A continuación, veremos un ejemplo de cómo crear un rol que provee **privilegios para correr dos bases de datos**.

- En primer lugar, nos **conectaremos a nuestra base de datos** MongoDB con los **privilegios** pertinentes:

Acceso con privilegios

```
mongo --port 27017 -u user -p '1234' --authenticationDatabase 'admin'
```

- A continuación, crearemos un nuevo rol para administrar las operaciones actuales. Crearemos un rol llamado "manRole":

Creación role

```
use admin.db.createRole(
  {
    role: "manRole",
    privileges:
      [
        {
          resource: { cluster: true },
          actions: [ "killop", "inprog" ]
        },
        {
          resource: { db: "", collection: "" },
          actions: [ "killCursors" ]
        }
      ],
    roles: []
  }
)
```

En la operación anterior podemos observar cómo hemos creado un rol que garantiza permisos para hacer **"kill" de cualquier operación**.

## Importación de datos

A continuación, procederemos a estudiar el proceso por el cual realizaremos el proceso de importación de datos a través de una herramienta visual llamada **Compass**.

1. El primer paso sería **conectarnos a nuestra base de datos** mongoDB como hemos estudiado previamente.
2. Después haríamos clic en el botón “**Add Data**” y seleccionaríamos la opción “**Import File**”.
3. La aplicación nos mostrará un cuadro de diálogo donde deberemos de indicar la **ruta del fichero** que queremos introducir a nuestra base de datos.
4. Una vez elegida la ruta, seleccionaremos el **tipo de fichero** que vamos a introducir:

### **JSON o CSV.**

Si importamos un **fichero CSV** deberemos especificar los **campos que vamos a importar** y los tipos de los mismos. El tipo de datos **por defecto es String**.

Como vemos en la imagen superior, tendremos que configurar las opciones de importación acorde a nuestro caso. Si importamos un CSV tenemos que indicar cómo están delimitados los campos.

5. Una **barra de progreso** mostrará el estado actual de la importación. Si **ocurre algún error** durante el proceso de importación, la **barra** se mostrará en **color rojo**, y aparecerá un **mensaje** en el cuadro de dialogo. Una vez finalizado el proceso, la **aplicación mostrará los datos importados**.

## Instalación Compass

Abordaremos los documentos de Mongo y exploraremos la documentación oficial.

Hemos observado varias herramientas visuales de gestión, pero Compass es la opción principal proporcionada por Mongo, ya que es desarrollada por ellos. Pueden acceder y descargar el software directamente desde la siguiente URL.

<https://www.mongodb.com/es/products/tools/compass>

Esta herramienta ha sido utilizada en los ejemplos de importación y exportación de colecciones que hemos estado aprendiendo.

La instalación es sencilla: descarguen el software desde la URL proporcionada y seleccionen la versión adecuada para su sistema operativo (Windows, Mac o Linux). Elijan entre alguna versión beta o la última estable. Sigan los pasos de instalación y consulten la documentación para obtener información detallada sobre la aplicación.

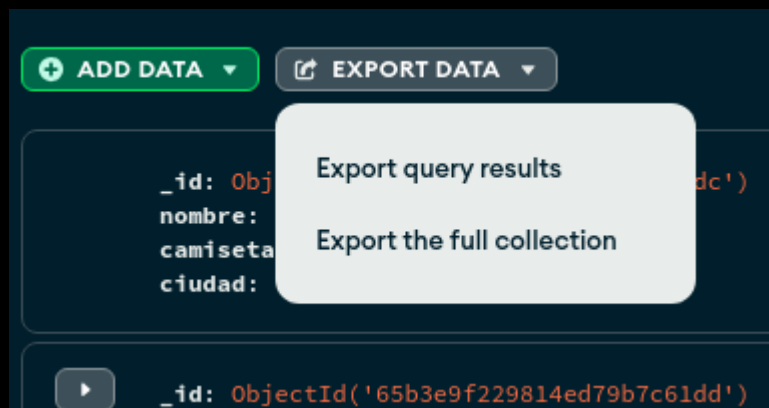
Al conectarnos a la base de datos, la documentación nos guía en el proceso de conexión y las interacciones con la información disponible. La página se presenta como un tutorial completo que cubre desde el inicio hasta el final de su funcionamiento. Aunque nos hemos centrado en la importación y exportación de datos, Compass es una herramienta potente que complementa al Shell. Es una excelente opción para aquellos que pueden sentir cierta aprensión hacia el Shell o simplemente prefieren una interfaz más visual.

Además de la gestión de datos, la aplicación ofrece funciones como importación-exportación, la cual hemos explorado en este tema. Les recomiendo explorar a fondo esta herramienta, ya que puede ser de gran utilidad para gestionar su base de datos Mongo. Aquí les dejo la herramienta para que la estudien y le saquen el máximo provecho.

## Exportación de datos

A continuación, procederemos a realizar la operación inversa. Realizaremos una exportación de datos con la aplicación visual Compass.

1. El primer paso sería **conectarnos a nuestra base de datos** MongoDB y navegar hasta encontrar la información, **colección/es** que deseemos exportar.
2. Haremos Clic en el menú “Collection” de nuestra aplicación y a continuación en la opción “Export Collection” y Compass nos mostrará el siguiente cuadro de dialogo:



El cuadro muestra la “query” por medio de la que se va a realizar la operación. Si queremos ignorar la “query” y exportar directamente la colección completa podemos seleccionar el radio button “Export Full Collection” y hacer clic en “Select Fields”.

3. En esta parte se nos muestra otro cuadro de diálogo donde seleccionaremos los **campos a exportar**.

Si nuestra aplicación **no nos detecta algún campo** podremos **añadirlo manualmente** con el botón de “Add Field”. Si todo está correcto hacemos clic en “Select Output”.

4. Básicamente en esta parte deberemos de seleccionar el formato del fichero que queremos exportar:

**JSON** o **CSV** son las opciones disponibles.

5. Hacemos clic en Export.

## Ejemplo de importación de fichero .csv

Disponemos de una base de datos MongoDB cuyo nombre es **Empleados**.

Se requiere importar a nuestra base de datos una colección de datos a través de un fichero .csv. Dicho fichero contiene la información personal de los empleados:

- Nombre
- Apellidos
- Edad
- Dirección

Un dato a tener en cuenta, es que no interesa disponer del campo “Dirección” en la colección que se importe en base de datos.

Para realizar la importación solicitada usaremos la herramienta visual ya utilizada, Compass.

1. En primer lugar, nos **conectaremos a la base de datos** donde vamos a importar la colección deseada.
2. Haremos clic sobre el botón de “Add Data” y justo después a la opción “Import File”.
3. Después elegiremos la ruta del fichero .csv que nos han proporcionado, elegiremos la **opción de csv**, pulsaremos en “Ignore empty Strings” para ignorar los campos vacíos y haremos clic en “Import”.
4. Aquí se nos abrirá nuestro **documento con los campos delimitados** por el csv introducido. El ejercicio en este punto nos especifica que no se quiere disponer del campo “Dirección” en la colección final de la base de datos.
5. La implementación de esto es tan sencilla como **desmarcar dicho campo**, y ese campo no será importado.
6. Hacemos clic en **Import** y tendremos nuestra colección nueva importada en la base de datos Empleados.

## Ejemplo de creación de usuario y rol

Se requiere crear un **nuevo usuario** en la **base de datos “Maths”**, en MongoDB, con las características de **role “readwrite”** para tener los siguientes permisos:

- *collStats*
- *convertToCapped*
- *createCollection*
- *dbHash*
- *dbStats*
- *dropCollection*
- *createIndex*
- *dropIndex*
- *find*
- *insert*
- *killCursors*
- *listIndexes*
- *listCollections*
- *remove*
- *renameCollectionSameDB*
- *update*



Para ello deberemos de ejecutar el comando en la base de datos:

**db.createUser()**

y dentro añadiremos el siguiente fragmento:

Ejemplo creación usuario

```
{
  user: "vGonzalez
  pwd: "rFgtR3456H",
  customData: { Este usuario tendrá privilegios sobre el rol readwrite },
  roles: [
    { role: "readWrite", db: "Maths" }
  ],
  mechanisms: [ "SCRAM-SHA-1" ],
  passwordDigestor: "server"
}
```

De esta forma estaremos:

1. creando el usuario "vGonzalez"
2. con la contraseña indicada en pwd,
3. y además añadiéndole el rol "readWrite" sobre la base de datos anteriormente mencionada, "Maths".

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

**Bases de datos nativas XML**  
**Almacenamiento**  
**Conexiones**

## Bases de datos XML

Las **bases de datos XML nativas** son bases de datos que almacenan documentos y datos XML de una forma muy **eficiente**. Igual que en las bases de datos relacionales, permiten que los datos se almacenen, consulten, combinen, aseguren, indexen, etc.

Las bases de datos XML nativas **no se basan en tablas**, sino en los llamados **contenedores**. Cada contenedor puede almacenar **grandes cantidades de documentos XML** o datos, que tienen alguna relación entre ellos.

Los contenedores también pueden tener subcontenedores. La gran diferencia con las bases de datos relacionales es que la **estructura** de los datos XML en un contenedor **no tiene por qué ser fija**. Podremos almacenar **distintas unidades de negocio** sin mucha relación dentro del mismo contenedor, la infraestructura lo permite, y podemos hacerlo, otra cosa es que sea más o menos recomendable.

Las bases de datos XML nativas no son consultadas por sentencias SQL, son **consultadas por expresiones Xpath**.

Xpath es un estándar mundial establecido por el “**W3C**” para **navegar** a través de documentos **XML**. Es un lenguaje que se puede utilizar para consultar datos de documentos XML. Las consultas Xpath se pueden utilizar para **escanear el contenido** de documentos XML.

Este lenguaje se basa en una **representación de árbol** del documento XML, y **selecciona nodos** según diferentes **criterios**.

Cuando se consulta una base de datos XML nativa:

1. el usuario generalmente **abre un contenedor**
2. y posteriormente, envía dicha **expresión Xpath contra todos los documentos XML** en la base de datos.
3. A continuación, se devuelve un conjunto de documentos XML.

## Ventajas y desventajas de las bases de datos XML con respecto a las relacionales

### Ventajas

- Las bases de datos XML nativas son capaces de **almacenar, mantener y consultar mayores cantidades de documentos XML** en comparación a las relacionales.
- A diferencia de las relacionales, **no es necesario configurar tablas**, y por tanto, no se necesita realizar diseños complicados antes de configurar la base de datos.

Una tabla de **base de datos clásica** tiene la desventaja de ser solo **bidimensional**, por lo que la estructura “más profunda” debe implementarse utilizando **claves secundarias**, lo que puede hacer que el diseño de una base de datos sea bastante complicado.

- La **facilidad de importación o exportación** hacia/desde otras aplicaciones con otros formatos.

### Nativas XML

Antiguamente estaba muy difundida la consideración que las bases de datos nativas son más lentas en las consultas que las bases de datos relacionales, pero hoy en día ya no es del todo correcta.

Las bases de datos XML actuales son seguramente **tan rápidas como las relacionales**, y además, **más fáciles de mantener y soportar**, con respecto a datos XML se refiere.

### Desventajas

- Las bases de datos relacionales están muy bien establecidas, es tecnología ya probada. Las bases de datos XML son un fenómeno algo **más reciente** ya que algunas de las primeras bases de datos XML **no** tienen mucho **más de una década** de antigüedad, por lo que la experiencia aún es limitada.
- **Xpath no es** un lenguaje excesivamente **fácil** de aprender, mientras que SQL está muy extendido. **No muchos desarrolladores y administradores** de bases de datos dominan Xpath. SQL está más relacionado con el lenguaje y la forma humana de ‘pedir’ cualquier cosa en general, por lo que es algo más sencillo de aprender.
- **No todas las aplicaciones gestoras de datos** soportan dicho lenguaje, tendríamos que buscar cuales son los compatibles con las bases de datos mencionadas.

## Gestores XML libres y comerciales

A continuación, veremos algunas herramientas gestoras de bases de datos XML nativas. Se presentarán herramientas comerciales de pago y otras de código abierto o libres.

### Sistemas gestores de bases de datos XML nativas **de pago**:

- *Excelon XIS*:

Con este gestor, las empresas pueden aprovechar de manera **completa y rentable** la extensibilidad y flexibilidad de XML para **crear, auditar y cambiar** continuamente aplicaciones que almacenan y manipulan **cantidades limitadas** de datos XML.

- *GoXML DB*:

Es una base de datos XML nativa con un motor de consultas de **alto rendimiento**. Los documentos XML **se almacenan directamente**, lo que elimina la necesidad de descomponer datos o configurar cómo se almacenarán los datos.

- *Infonyte DB*:

La tecnología de base de datos de Infonyte se distingue por el soporte nativo para XML, la **independencia** de la plataforma y el **uso moderado de los recursos** del sistema. Por lo tanto, se adapta perfectamente a los requisitos tanto de arquitecturas de componentes como de **dispositivos pequeños**.

En estos mercados emergentes, su liderazgo tecnológico les da una ventaja competitiva sobre las soluciones actuales.

### Sistemas gestores de bases de datos XML nativas **libres o de código abierto**:

- *Exist DB*:

Es un proyecto de software de código abierto para **bases de datos NoSQL** construido sobre tecnología XML. Está clasificado como un sistema de base de datos **orientado a documentos** NoSQL y una base de datos XML nativa.

También proporciona **soporte** para documentos:

- **XML**,
- **JSON**,
- **HTML**,
- **binarios**.

A diferencia de la mayoría de los sistemas de administración de bases de datos relacionales, proporciona **Xquery** y **XSLT** como lenguajes de **programación** de aplicaciones y **consultas**.

- *X-Hibe DB:*

Es una poderosa base de datos XML nativa diseñada para desarrolladores de software que requieren **funciones avanzadas de procesamiento y almacenamiento** de datos XML dentro de sus **aplicaciones**.

- *Tamino DB:*

Es una base de datos XML nativa. Si la comparamos con una base de datos relacional, tiene la desventaja de no ser muy popular. Sin embargo, si para trabajar con ella se parte de unos **datos ya estructurados**, se pueden encontrar muchas **ventajas** y posiblemente sea una opción preferente, **frente** a un motor **SQL** tradicional.

## Instalación y configuración de EXISTDB

Dedicaremos este apartado para realizar la instalación y configuración necesaria para la base de datos XML nativa ExistDB. Para ello, deberemos de tener en cuenta unos requisitos de sistema recomendados (hay unos mínimos menos restrictivos) que son los siguientes:

- Si instalamos una versión final superior a la versión 3.0 tendremos que tener instalado previamente el **JDK Java 8**.
- Debemos asignar **128 MB de memoria caché**.
- Debemos de tener **1024 MB de memoria RAM** disponibles para poderle asignar al aplicativo.

Una vez descargado el archivo .jar (<https://exist-db.org/exist/apps/homepage/index.html>), se realizará la instalación.

Para ello, hacemos doble clic en el fichero y lanzaremos directamente la aplicación.

*En Linux: ejecutamos el comando:*

***java -jar eXist-db-setup-[version].jar***

*desde la carpeta del paquete .jar.*

En el proceso de instalación se nos preguntara sobre el directorio habitual de nuestra aplicación por si queremos modificarlo. También se nos ofrecerá un directorio de donde cogerá los ficheros de información, podemos dejar el que nos ofrece Exist o cambiarlo al que decidamos.

A continuación, nos encontraremos la configuración de memoria y también la contraseña del usuario Admin. Esta cuenta pertenecerá a la persona que está llevando a cabo la instalación, por lo que determinadas funcionalidades en este gestor, solo podrán ser ejecutadas con dicho usuario. Así que una buena recomendación es **cumplimentar el campo password del usuario Admin**. con una **contraseña fuerte**.

Finalmente, en este punto, es necesario configurar y establecer la **cantidad de memoria RAM** que queremos darle a nuestro proceso Java y a nuestra **memoria Caché**.

El instalador dice:

*En Windows y Linux, utilice el elemento del menú de inicio de eXist-db o el icono del escritorio para iniciar eXist-db. En Mac OS, haga doble clic en el icono eXist-db.app dentro de la carpeta en la que está instalado eXist-db.*

*También puede iniciar eXist-db haciendo doble clic en start.jar o llamando a "java -jar start.jar" en una línea de comando.*

*Al iniciar eXist-db de una de estas formas se mostrará una pantalla de presentación al iniciar la base de datos. Después del inicio, encontrará un icono de eXist-db en la bandeja del sistema/barra de menú. Haga clic (Windows y Mac OS) o haga clic derecho (Linux) en el icono para obtener un menú emergente con más opciones.*

## El paquete de instalación

Siguiendo con el proceso de instalación y configuración, el siguiente paso sería el paquete de instalación.

Estos son algunos de los distintos **paquetes que podemos agregar** a la instalación:

- El paquete “**core**” es requerido para la instalación, ya que es uno de los elementos que nos permitirá **‘correr’ la base de datos**.
- El paquete “**sources**” es opcional; deseleccionaríamos dicho paquete si tuviéramos problemas de espacio, si no, **es preferible instalarlo**.
- El paquete “**apps**” nos permite seleccionar o deseleccionar una serie de **apps que serán instaladas cuando ExistDB arranque por primera vez**. Es recomendable **dejarlas** para dar los primeros pasos.

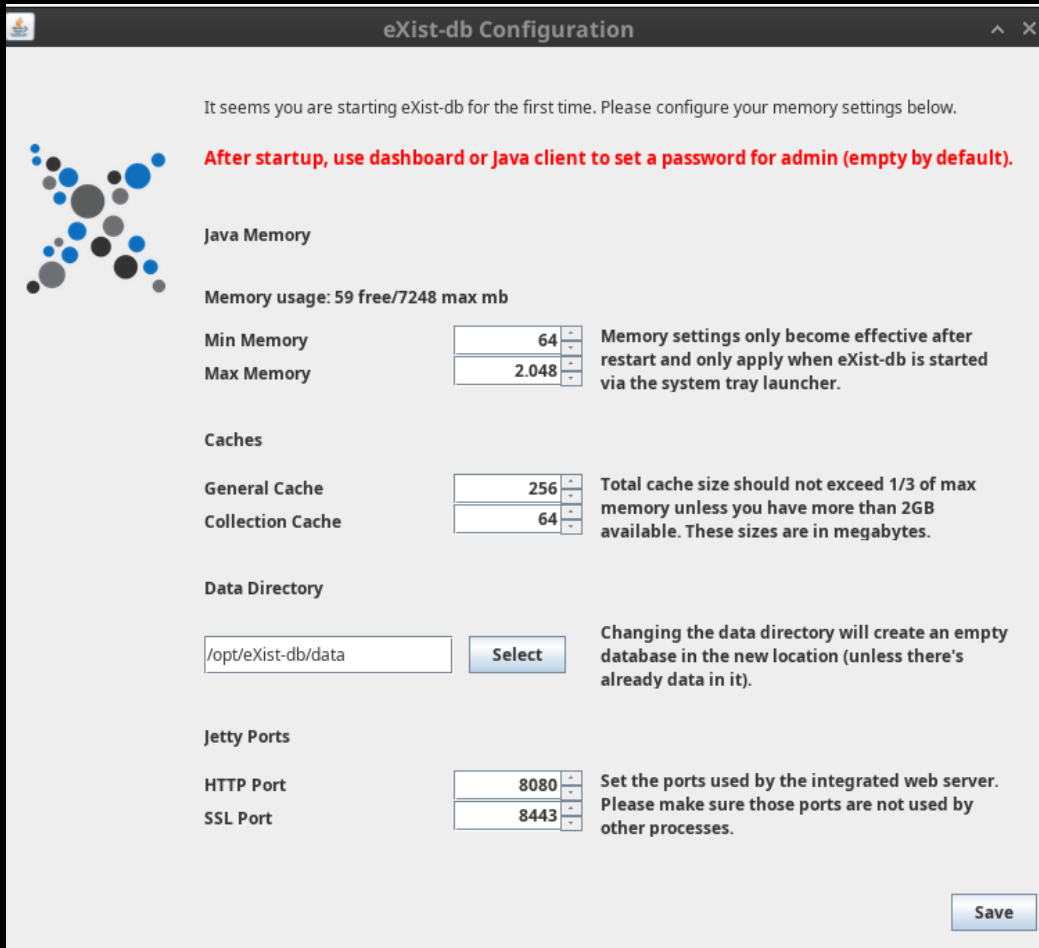
Una vez hechas las selecciones oportunas, hacemos click en Next, y **finalizaremos la instalación**.

Para lanzar la base de datos en Linux o en Windows, simplemente ejecutaremos el **acceso directo** que nos ha generado la instalación en el menú de inicio (En Linux se nos habrá creado un .desktop en la carpeta de instalación). Aparecerá un **logo inicial** con una imagen Splash mientras las aplicaciones seleccionadas previamente se van instalando. Una vez tenemos nuestra base de datos instalada localizaremos un nuevo icono de bandeja del sistema que nos dará acceso a todas las herramientas que nos ofrece ExistDB y nos permitirá **apagar o reiniciar la base de datos**.

Si queremos **lanzar nuestra base de datos desde línea de comandos** nos dirigiremos al **directorio de instalación** de la base de datos y llamaremos a “launcher.sh” o “launcher.bat” dependiendo si usamos Linux o Windows. (Nota: En Linux es: /usr/local/eXist-db/bin/).



## Instalación Exist. Dashboard

The image shows a window titled "eXist-db Configuration". On the left is a logo consisting of a cluster of blue and grey dots. The main area contains configuration options for Java Memory, Caches, Data Directory, and Jetty Ports. Each section has input fields and a "Select" button. A "Save" button is at the bottom right. A red text instruction is at the top right.

It seems you are starting eXist-db for the first time. Please configure your memory settings below.

**After startup, use dashboard or java client to set a password for admin (empty by default).**

**Java Memory**

Memory usage: 59 free/7248 max mb

Min Memory	<input type="text" value="64"/>	Memory settings only become effective after restart and only apply when eXist-db is started via the system tray launcher.
Max Memory	<input type="text" value="2.048"/>	

**Caches**

General Cache	<input type="text" value="256"/>	Total cache size should not exceed 1/3 of max memory unless you have more than 2GB available. These sizes are in megabytes.
Collection Cache	<input type="text" value="64"/>	

**Data Directory**

Changing the data directory will create an empty database in the new location (unless there's already data in it).

**Jetty Ports**

HTTP Port	<input type="text" value="8080"/>	Set the ports used by the integrated web server. Please make sure those ports are not used by other processes.
SSL Port	<input type="text" value="8443"/>	

(Nota de apunte: En Linux, nos aseguramos de ejecutar el `launcher.sh` como `sudo` para poder guardar los cambios de configuración java).

Una vez instalado y lanzado podemos navegar hasta el [Dashboard](#) de la base de datos, abriéndolo desde el **icono de sistema** o accediendo directamente a:

<http://localhost:8080/exist/>

## Integración ExistDB

Vamos a repasar algunas opciones adicionales que ahora tenemos para traer nuestra base de datos ExistDB a nuestro entorno local. Al ingresar en la URL de la base de datos ExistDB, que es nuestra base de datos XML nativa, encontramos el botón de descarga. Si hacemos clic en él y navegamos por la web interactiva, veremos diferentes formas de descarga o soporte. Nos enfocaremos en las tres primeras opciones que nos resultan interesantes:

- La primera opción, que ya discutimos en la unidad, consiste en descargar directamente la **última release estable**, es decir, la última versión disponible de nuestra base de datos. Solo tenemos que hacer clic en el enlace correspondiente y proceder con la descarga.
- La segunda opción es útil si queremos tener un entorno virtualizado localmente. Si deseamos utilizar Docker, crear un **Docker Compose** y levantar la imagen de esta base de datos en nuestro puerto local, podemos descargarla, realizar un pull y utilizarla con Docker. Al levantar la imagen, podemos establecer conexiones directas desde nuestra aplicación.
- Por último, quisiera mencionar **Maven Artifacts**. Al conectarnos a Maven Artifacts, podemos obtener las últimas versiones de los arquetipos creados por esta compañía para esta base de datos. Podemos revisar los cambios más recientes y, específicamente, al explorar uno de ellos, descargar directamente el **JAR** o revisar el archivo **POM** para ver las versiones de los arquetipos, entre otros detalles. Aquí encontraremos las versiones más recientes utilizadas, lo cual puede ser de gran utilidad si queremos incorporar la dependencia Maven directamente en nuestra aplicación para utilizarla con Java.

Quisiera destacar que estas tres opciones también están disponibles para la base de datos, y les recomiendo revisarlas para determinar si son útiles para integrar en sus aplicaciones.

## Estrategias de almacenamiento

Las bases de datos nativas XML pueden ser clasificadas **dependiendo del tipo de almacenamiento que vayan a utilizar**, de este modo, a continuación, veremos los diferentes **modos de almacenamiento** de los que disponemos en una base de datos nativa XML:

- Almacenamiento basado en **texto**.
- Almacenamiento basado en **modelo**.
- Almacenar en **local**.

### Almacenamiento **basado en texto**:

Esta modalidad consiste en **almacenar el documento completo** en base de datos, y dotar a la misma, de algún tipo de **funcionalidad** para que se pueda **acceder fácilmente** a él.

En este tipo de almacenamiento suele:

- realizarse la **compresión** de ficheros.
- añadir algunos **índices** específicos para aumentar la eficiencia.

Las **posibilidades** para esto serían dos:

- Almacenar el fichero binario de tipo **BLOB**:
  - en un **sistema relacional**.
  - en un soporte o **almacén orientado a dicha operación con**:
    - **índices**,
    - soporte de transacciones,
    - etc.

### Almacenamiento **basado en el modelo**:

Para este caso, se utilizaría un **modelo de datos lógico** como por ejemplo puede ser **DOM**, para definir la **estructura y la jerarquía** de los documentos XML que se vayan a almacenar.

Se guardaría el modelo del documento en un **almacén definido** previamente.

Para esto, tenemos algunas **posibilidades** como:

- Traducir desde **DOM** a **tablas** de una base de datos convencional relacional.
- Traducir el objeto **DOM** a **objetos** en una base de datos orientada a objetos.
- Usar un **almacén de datos** creado específicamente para esta utilidad.

Almacenar una forma modificada del documento **en local**:

Podremos usar este tipo de almacenamiento, cuando la **cantidad** de ficheros a almacenar **no** sea muy **elevada**, y **no** se realicen **numerosas actualizaciones** ni **transferencias** de los mismos. Realmente consiste en almacenar una **forma modificada** del fichero XML base completo, directamente en **sistema de archivos**.

Evidentemente tiene diferentes **limitaciones** como **escalabilidad, flexibilidad, recuperación y seguridad**.

## Establecimiento y cierre de conexiones

A continuación, realizaremos una **conexión** con la base de datos ExistDB como ejemplo.

Para ello deberemos de **tener en cuenta** ciertos aspectos:

- `Org.xmlldb.api`: Nos enriquecerá el código con las **Interfaces** y **DatabaseManager**.
- `Org.xmlldb.api.base`: Con esta librería accederemos a:
  - las **colecciones**,
  - los **objetos** Database,
  - **Resource**,
  - **ResourceIterator**,
  - **ResourceSet**,
  - y algunos más.
- `Org.xmlldb.api.modules`: Accederemos a:
  - los servicios de **transacciones**,
  - **XMLResource**,
  - servicios de XpathQueryService,
  - y otros varios relacionados.

Supondremos que tenemos en nuestra **aplicación Java ya desarrollada**, una **clase dedicada al acceso** a capa de datos, donde dicha aplicación:

- **accederá**,
- establecerá **conexión**,
- realizará algunas **consultas**,
- y obtendrá **resultados**.

Para ello iremos **paso a paso**:

1. Primeramente, indicaremos el `driver` a cargar:

Driver

```
String driver = "org.exist.xmlldb.DatabaseImpl";
```

2. Con esta línea cargaremos el driver instanciado:

Carga Driver

```
Class clase = Class.forName(driver);
```

3. Acto seguido, instanciamos la base de datos:

Instancia

```
Database database = (Database) clase.newInstance();
```

4. Con esta línea registraremos la base de datos:

Registro

```
DatabaseManager.registerDatabase(database);
```

5. A continuación, mostraremos algunas líneas de código para acceder a la colección.

Con ellas, **preparamos el código** para la colección que más tarde consultaremos:

Acceso a la colección

```
Collection coleccion =  
DatabaseManager.getCollection(  
    "xmldb:exist://localhost:8080/exist/xmlrpc/db/",  
    "usuario",  
    "contraseña");  
XPathQueryService service =  
(XPathQueryService) coleccion.getService(  
    "XPathQueryService", "1.0");
```

6. Por último, realizaremos las **consultas** a base de datos.

Los resultados obtenidos se almacenarán en la variable “**resultado**” que posteriormente tendremos que **iterar**, y realizar su procesado:

ResultSet

```
ResourceSet resultado = service.query(  
    "for $b in doc('prueba.xml')//a return $b");
```

7. Una vez hechas las consultas tendremos en cuenta en qué consisten los **conceptos** de:

- **Indexación**: Permiten la **creación de índices** para **acelerar** las **consultas** frecuentes.
- **Identificador único**: Cada documento XML está asociado con un identificador único, a través del cual se puede identificar **en el repositorio**.

## Agregar, modificar y eliminar recursos

También es posible **agregar** nuevos **recursos XML y no XML** a la colección (objeto de "Collection").

Para esto, se necesitan las siguientes clases y métodos:

La **clase Collection** representa una **colección de recursos** (resource) almacenados en una base de datos XML. Tiene **métodos** como los siguientes:

- **storeResource** (resource res):

Es el método más relevante para **agregar nuevos recursos** en esta clase. Almacena el **recurso res** proporcionado por el parámetro en la colección.

- **removeResource** (recurso res): **Elimina el recurso** res pasado al recurso a través de parámetros de la colección.
- **listResources** (): Útil para crear y eliminar nuevos recursos; devuelve una **matriz de cadenas** que contiene los ds de todos los recursos que tiene la colección;
- **getResourceCount**() obtiene la cantidad de recursos almacenados en la colección;
- **createResource** (java.lang.String id, java.lang.String type), crea un nuevo recurso vacío en la colección, cuyo **ID y tipo** son pasados por **parámetros**.

Si ya se comprende el proceso de creación de un nuevo recurso, se puede definir el proceso de eliminación más fácilmente. Para esto, la clase Collection vuelve a intervenir. La forma principal de eliminar recursos es: **removeResource** (resource res), que elimina resource res de la colección.



## Creación y borrado de colecciones

A continuación, veremos un **ejemplo de agregación y borrado** de colecciones en código:

Agregación y borrado de colecciones

```
public Collection anadirColeccion (
    Collection contexto,
    String newColec) throws ExcepcionGestorBD {
    Collection newCollection = null;
    try {
        //Creamos un nuevo servicio.
        CollectionManagementService mgtService = (
            CollectionManagementService) contexto.getService(
                "CollectionManagementService", "1.0");
        //Creamos una nueva collection con codificación UTF8
        newCollection = mgtService.createCollection(
            new String(UTF(.encode(newColec))))
    } catch (XMLDBException e) {
        throw new ExceptionGestorDB (
            "Error agregando coleccion: " + e.getMesagget());
    }
    return newCollection;
}
```

Si lo que deseamos es borrar dicha colección simplemente tendremos que llamar al método: `removeCollection(String nombre)` de la clase “CollectionManagementService”

## Modificación de contenidos XML

Estudiaremos distintas **acciones para modificar el contenido** de un árbol DOM.

- Modificaremos el valor del texto asociado a una etiqueta.
- Podremos modificar el valor de un atributo asociado a una etiqueta.

Modificación del **valor del texto** asociado a una etiqueta:

Valor texto

```
Node nodo = raiz.getElementsByTagName("nombreDelTag").item(0);  
nodo.setTextContent("Otro");
```

- primera línea: obteniendo un nodo.
- segunda línea: con el método “setTextContent” estaremos estableciendo y modificando el valor del texto asociado a dicha etiqueta.

Modificación del **valor de un atributo** asociado a una etiqueta:

Imaginemos que poseemos un objeto de la clase “**Node**” que **representa una etiqueta**. La idea es **convertir** dicho objeto a un objeto de la clase “**Element**” para utilizar el método “**setAttribute**(String)”.

Para ello tendremos que realizar un **casting de “Element”** sobre el nodo deseado. A continuación, podemos ver el código:

Valor atributo

```
Element elemento = doc.getDocumentElement();  
elemento.setAttribute("nombre", "valor");
```

Tal y como podemos observar en el código anterior, extraemos el elemento y trabajamos con el método “setAttribute”, indicándole el nombre de la etiqueta, y a continuación, introduciendo el valor deseado. Para obtener dicho contenido simplemente usaríamos:

```
System.out.println(elemento.getAttribute("nombre"));
```

## Transacciones y excepciones

### Transacciones XML

En algunos de los ejemplos expuestos en el tema, existe el concepto de transacción: un conjunto de declaraciones ejecutadas que **forman inseparablemente una unidad** de trabajo, de modo que todas se ejecutan o no se ejecutan.

Al administrar las transacciones, el administrador XML proporciona **acceso simultáneo a los datos** almacenados, mantiene la **integridad y seguridad** de los datos, y proporciona un **mecanismo de recuperación de fallos** en la base de datos.

Exist-db admite transacciones **compatibles con ACID**:

- **Atomicidad**. Se deben completar **todas** las operaciones de la transacción, **o** no realizar **ninguna** operación, no se puede dejar a la mitad.
- **Consistencia**. La **transacción finaliza** solo cuando la base de datos permanece en un **estado coherente**.
- **Aislamiento**. Las transacciones sobre una misma información deben ser **independientes** para que no interfieran con sus operaciones, y no generen ningún tipo de error.
- **Durabilidad**. Al final de la transacción, el **resultado** de la misma **se conservará** y no se podrá deshacer incluso si el sistema falla.

### Tratamiento excepciones

La clase `XMLDBException` captura todos los errores que ocurren cuando se usa XML (DB para procesar la base de datos).

Esta excepción contendría dos **códigos de error**:

- Un código de error **especificado por cada sistema** de procesamiento XML local (fabricante-proveedor).
- Un código de error **definido en la clase ErrorCodes**.

Si el error que ocurrió en un momento dado es **parte del sistema** de administración XML, `ErrorCode` debe tener un valor de `errorCodes`, **VENDOR\_ERROR**.

La clase `ErrorCodes` define los diferentes **códigos de error XML**: DB utilizado por el atributo `errorCodes` de la **excepción XMLDBException**.

## Descarga e instalación de ExistDB

Se requiere instalar una base de datos XML nativa para realizar las distintas conexiones. Al menos descarga e instalación.

Para realizar la instalación de la base de datos ExistDb, en primer lugar, nos dirigiremos a la web oficial de la herramienta, en este caso:

<http://exist-db.org/exist/apps/homepage/index.html>

Como podemos ver tenemos diferentes **formas de integración de la base de datos** en nuestro sistema, las más interesantes:

- Descargar **directamente** la **última versión** de base de datos e instalarla.
- Descargar una **imagen de docker** y montar dicha imagen para usar la herramienta.
- **Dependencia maven**, que nos ayudará a **inyectar la dependencia** de dicha base de datos, en el proyecto en el que estemos trabajando.

Elegiremos para este caso la primera opción y nos descargaremos la última versión “release” estable de la misma.

Realmente lo que nos estamos descargando es un fichero con **extensión .jar**. Simplemente ejecutaremos dicho fichero .jar para realizar la **instalación** tanto para sistemas Unix como para Windows. Con este pequeño proceso nos bastará para tener instalada nuestra base de datos. Más tarde el usuario tendrá que realizar el **arranque de base de datos** y la **integración** con la aplicación.

## ExistDB config file

Actualmente tenemos una base de datos XML nativa asociada a una aplicación Java de registro de clientes.

Se requiere cambiar algunos aspectos de configuración de la base de datos instalada Exist DB:

- Cambiar el tamaño de **caché a 512MB**.
- Cambiar la **ubicación de los ficheros a usuario/Downloads**.

Una vez instalada nuestra base de datos nativa XML nos dispondremos a realizar varias configuraciones:

**En primer lugar**, comentar, que el **fichero principal de configuración** de dicha base de datos es el fichero "**conf.xml**", como suele ser habitual en numerosos sistemas gestores de base de datos.

Dicho fichero se encuentra alojado en el **directorio raíz** de instalación de nuestra base de datos. (Nota de apunte: en mi instalación está en: /etc/ del directorio de instalación de la base de datos).

A continuación, modificaremos los 2 aspectos que se han requerido y veremos algunos atributos más de configuración del mismo.

### Atributos

```
<db-connection
  cacheSize="512M"
  collectionCache="24M"
  database="native"
  files=" ../usuario/Downloads"
  pageSize="4096"
  nodesBuffer="-1">
  <pool
    max="15"
    min="1"
    sync-period="240000"
    wait-before-shutdown="60000" />
  <recovery
    enabled="yes"
```

```
    sync-on-commit="no"
    group-commit="no"
    size="100M"
    journal-dir=" ../data" />
    <watchdog output-size-limit="10000" query-timeout="-1" />
    <default-permissions collection="0775" resource="0775" />
</db-connection>
```

Como podemos comprobar, en este caso simplemente hemos realizado el cambio que se nos requería:

- Atributo “**cacheSize**” hemos seteado a **512 Megabytes**.
- Atributo “**files**” para indicar la **ubicación de los ficheros**.

Como podemos observar en este **bloque de “db-connection”** del **fichero conf.xml**, tenemos numerosos atributos más, sobre los que podremos aprender su funcionalidad, en la página oficial de la base de datos Exist:

<https://exist-db.org/>

## Ejemplo de preparación de acceso a base de datos XML

Se requiere instalar una base de datos XML nativa, y dejar preparada una **capa de acceso a datos** con las líneas esenciales, al menos, para realizar conexión con base de datos.

En primer lugar, descargaremos la última versión estable de la aplicación ExistDB. Para ello, nos dirigiremos a la web oficial:

<http://exist-db.org/exist/apps/homepage/index.html>

Ejecutaremos el fichero .jar con doble clic y seguiremos la instalación con los diferentes pasos que hemos aprendido en las páginas anteriores.

Una vez instalado, nos dispondremos a realizar una nueva **paquetería en nuestra aplicación** donde contendrá la **clase de acceso** a datos. Normalmente podremos crear una clase con nomenclatura parecida a:

*AcessDataDao.java*

dentro de la estructura: "ApplicationName.com.Java.Dao".

Una vez creada la clase añadiremos las siguientes líneas de código para realizar la **conexión con la base de datos**:

```
String driver = "org.exist.xmlldb.DatabaseImpl";
Class clase = Class.forName(driver);
Database database = (Database) clase.newInstance();
DatabaseManager.registerDatabase(database);
```

Con estas líneas de código, estaremos:

- **dando de alta nuestro driver** para la base de datos XML nativa,
- y al mismo tiempo, estaremos **realizando el registro**.

Una vez realizadas estas simples operaciones, a continuación, podríamos preparar diferentes **colecciones** para obtener **resultados realizando consultas**.

<http://exist-db.org/exist/apps/homepage/index.html>

## ACCESO A DATOS

### TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

#### Programación de componentes de acceso a datos



## Concepto y características de un **componente**

Generalmente, un componente es una **unidad de software** que **encapsula** un segmento de **código** con ciertas **funciones**.

Los estilos de estos componentes pueden ser:

- estilos proporcionados por el **entorno de desarrollo** (incluidos en la **interfaz** de usuario)
- o pueden ser **no visuales**, siendo sus funciones similares a las de las bibliotecas remotas.

Los componentes del software tienen las siguientes **características principales**:

- Un componente es una unidad ejecutable que se puede instalar y utilizar de forma independiente.
- Puede interactuar y operar con otros componentes desarrollados por terceros, es decir, empresas o desarrolladores pueden utilizar componentes y agregarlos a las operaciones que se realizan, lo que significa que pueden estar **compuestos por estos componentes**.
- No tienen estado, al menos su estado no es visible desde 'fuera'.
- La programación orientada a componentes o un componente, se puede asociar a los distintos engranajes electrónicos que, en su **conjunto**, forman un **sistema más grande**.

Podemos afirmar que un **componente** será formado por los siguientes elementos:

1. Atributos, operaciones y eventos: Es parte de la interfaz del componente. Todo en su junto representaría el **nivel sintáctico**.
2. Comportamiento: Representa la parte **semántica**.
3. Protocolos y escenarios: Representa la **compatibilidad** de las secuencias de los mensajes con otros componentes, y la **forma de actuar** que tendrá el componente en diferentes **escenarios** donde llegará a ser ejecutado.
4. Propiedades: Las diferentes **características** que puede tener el componente.

## Propiedades y atributos de un componente

### Indexadas y Simples

Las **propiedades** del componente **determinan su estado** y lo distinguen del resto. Se ha adelantado anteriormente que estos componentes **carecen de estado**, pero estos componentes tienen una serie de propiedades a las que se puede **acceder y modificar** de diferentes formas.

Las **propiedades de los componentes se dividen** en:

1. **simples**,
2. **indexadas**,
3. **compartidas**,
4. y **restringidas**.

### Alcance de las propiedades de los beans de Java

Los atributos y las propiedades de los beans de java deben mantener su scope (alcance) **privado**. De esta forma fomentamos la **seguridad y el acceso** a dichas propiedades y atributos, y sólo se propagarán los cambios **llamando al método** determinado de dicha propiedad.

Se puede verificar y modificar las propiedades del componente accediendo al método del componente o la función de acceso.

Hay **dos tipos de estas funciones**:

- El método get se utiliza para **consultar** o leer el valor de un atributo.

La sintaxis general es la siguiente:

```
Tipo getNombrePropiedad()
```

- El método set se utiliza para **asignar** o cambiar el valor de un atributo.

Su sintaxis general en Java es:

```
SetNombrePropiedad(Tipo valor)
```

Centrándonos en las propiedades o atributos indexados y simples, podemos decir, que:

- los **atributos simples** son atributos que **representan un solo valor**.

Por ejemplo, si hay un botón en la interfaz gráfica, los atributos simples serán atributos relacionados con su **tamaño**, **color** de fondo, **etiqueta**, etc.

- Los **atributos o propiedades indexadas**: son propiedades más complejas que son muy similares a un conjunto de valores: los índices. Todos los **elementos de este tipo** de atributo comparten el **mismo tipo** y se puede acceder a ellos **por índice**.

Concretamente, se puede **acceder** mediante el método de acceso que se mencionó anteriormente (**get** / **set**), aunque la llamada de estos métodos variará, ya que **solo se puede acceder a cada propiedad a través de su índice**.

La sintaxis que encontramos en Java sería:

```
setPropertyName (int index, PropertyType value)
```

## Propiedades y atributos de un componente

### Compartidas y Restringidas

Además de los atributos simples y los atributos indexados, los componentes también tienen otros dos tipos de atributos:

- Atributos compartidos.
- Atributos restringidos.

Los **atributos compartidos**:

Se refieren a los datos mediante los que **informa a todos los interesados** sobre el atributo, cuando cambian, por lo que solo se les notifica **cuando cambia el atributo**. El mecanismo de notificación se basa en eventos, es decir, hay un **componente de origen** que notifica al **componente receptor** cuando un atributo compartido cambia a través de un **evento**. Debe quedar claro que estas propiedades no son bidireccionales, por lo que el **componente receptor no puede responder**.

Para que un componente **permita propiedades compartidas**, debe **admitir dos métodos** Java para registrar componentes que estén interesados en cambios de propiedad.

Los métodos y su sintaxis son:

- Para agregar: `addPropertyChangeListener (PropertyChangeListener x)`
- Para eliminar: `removePropertyChangeListener (PropertyChangeListener x)`

Las **propiedades restringidas**:

Buscarán la **aprobación de otros componentes** antes de cambiar su valor.

Al igual que con los atributos compartidos, se deben proporcionar **dos métodos** de registro para el receptor.

Su sintaxis general en Java sería:

- `addVetoableChangeListener (VetoableChangeListener x),`
- `removeVetoableChangeListener (VetoableChangeListener x).`

Las **propiedades** son uno de los **aspectos relevantes de la interfaz** del componente porque son elementos que forman parte de la **vista externa** del componente (representada como una vista **pública**). Desde la perspectiva del control y uso de componentes, estos elementos observables son particularmente importantes y son la **base para caracterizar otros aspectos** de los componentes.

## POJOS y beans

Bienvenidos al ejemplo de un POJO en Java. Ahora vamos a revisar algunos detalles y diferencias entre los "beans" y los POJOs. Todos los "beans" son POJOs, pero no todos los POJOs son "beans". Aunque pueda parecer un poco confuso, así es como funciona.

Este es un ejemplo básico de un [POJO](#). Tenemos la clase 'Empleado', que tiene varios atributos y un constructor público que nos permite crear un objeto a partir de cero, introduciendo estos atributos. También tenemos nuestros getters aquí. Podríamos agregar setters también, pero este es un ejemplo básico.

¿Qué tenemos que tener en cuenta acerca de los "beans"? Bueno, en comparación con los POJOs, todos los "beans", como mencionamos antes, son POJOs. Pero no todos los POJOs son "Java beans". Otro punto importante es que **implementan** la interfaz '**Serializable**'. Todos los campos deben ser **privados** en los "beans". Esto se hace para proporcionar un control completo sobre los campos. Mientras que **en los POJOs no es obligatorio**; podemos tener campos privados o públicos.

Además, todos los campos en los "**beans**" deben tener sus **getters y/o setters**. Otra característica es que pueden tener un constructor sin argumentos. Luego, los campos solo pueden ser accedidos mediante el constructor, los getters o los setters.

Es importante mencionar que los **getters y setters** tienen ciertos **nombres especiales**, dependiendo del nombre del campo. Por ejemplo, si el nombre del campo es 'xProperty' o 'someProperty', preferiblemente el getter será 'getSomeProperty'.

Por último, la **visibilidad de los getters y setters** en general es **pública**, precisamente para acceder a los atributos desde esos métodos. Además, la visibilidad de los atributos mismos se establece como pública, para que cualquier objeto pueda usarlos.

Básicamente, estas son algunas de las diferencias entre los "beans" y los POJOs. Los "beans" son objetos más restrictivos en comparación con los POJOs.

## Ejemplo de POJO en Java

```
package pruebas;

public class Employee {
    // campo por defecto
    String name;
    // campo publico
    public String id;
    // salario privado
    private double salary;
    // constructor para inicializar los campos
    public Employee(String name, String id, double salary) {
        this.name = name;
        this.id = id;
        this.salary = salary;
    }
    // getters
    public String getName() {
        return name;
    }
    public String getId() {
        return id;
    }
    public Double getSalary() {
        return salary;
    }
}
```

## Ejemplo de Implementación de un componente y propagación

Se requiere la realización de un componente que a través de la clase `PropertyChangeSupport` ejecute los métodos propagados de:

- `addPropertyChangeListener`
- `removePropertyChangeListener`

Tal y como hemos visto en los puntos anteriores de la unidad, nos dispondremos a usar la lógica de componentes EJB para así poder propagar los diferentes cambios.

1. En primer lugar deberemos de **importar** la paquetería `java.beans`.
2. Luego crearíamos un objeto de la clase mencionada `PropertyChangeSupport`:

PropertyChange objeto

```
private PropertyChangeSupport objetoCambios = new PropertyChangeSupport(this);
```

Como podemos observar, el objeto del código anterior mantiene una lista de oyentes y por consiguiente, lanzará los diferentes eventos de cambio de propiedad.

3. A continuación, implementaremos los **métodos que nos ofrece `PropertyChangeSupport`**, simplemente envolveremos las diferentes llamadas de los métodos del objeto soportado:

Métodos

```
public void addPropertyChangeListener(  
    PropertyChangeListener listener) {  
    changes.addPropertyChangeListener(listener);  
}  
  
public void removePropertyChangeListener(  
    PropertyChangeListener listener) {  
    changes.removePropertyChangeListener(listener);  
}
```

## Eventos

La **iteración entre componentes** se denomina **control activo**, es decir, el **funcionamiento** de un componente se activa mediante la **llamada de otro componente**.

Además del control activo (la forma más común de invocar operaciones), existe otro método llamado **control reactivo**, que se refiere a **eventos de componentes**, como los permitidos por el modelo de componentes EJB.

En el control reactivo, los componentes pueden generar eventos correspondientes a **solicitudes de invocación de operaciones**. Posteriormente, otros componentes del sistema recolectarían estas solicitudes, y activarían **llamadas a operaciones** para sus propósitos de procesamiento.

Por ejemplo, cuando el puntero del mouse pasa sobre el ícono del escritorio, si la forma del ícono del escritorio cambia, el componente de ícono emitiría eventos relacionados con la operación de cambiar la forma del ícono.



## Introspección

Las herramientas de desarrollo **descubren las características** de los componentes (es decir, propiedades de los **componentes, métodos y eventos**) a través de un proceso llamado introspección. La introspección, por tanto, se puede definir como un mecanismo a través del cual se pueden **descubrir las propiedades, métodos y eventos** contenidos en el componente.

Los componentes admiten la introspección de **dos formas**:

- Mediante el **uso de convenciones de nomenclatura** específicas, que se conocen al nombrar las características de los componentes. Para EJB, la clase **Introspector** comprueba el EJB para encontrar esos **patrones de diseño** y descubrir sus características.
- Proporcionando **información clara sobre atributos, métodos o eventos** de clases relacionadas.

En particular, EJB admite **múltiples niveles de introspección**.

En un **nivel bajo**, esta introspección se puede lograr mediante la posibilidad de **reflexión**. Estas características permiten a los objetos Java descubrir **información** sobre **métodos** públicos, **campos** y **constructores** de clases cargados durante la ejecución del programa. La reflexión permite la introspección de todos los componentes del software, y todo lo que se tiene que hacer es **declarar el método o variable como público** para que pueda ser descubierto a través de la reflexión.

## Persistencia del componente

Para EJB, la persistencia se proporciona con la biblioteca **JPA** de Java. Hoy en día podemos encontrar tal especificación en **Oracle**. Para usar JPA, se requiere el uso al menos del **JDK 1.5** (conocido como Java 5) en adelante, ya que amplía las nuevas especificaciones del lenguaje Java, como son las **anotaciones**. Para utilizar la biblioteca JPA, es esencial tener un conocimiento sólido de **POO** (programación orientada a objetos), **Java**, y **lenguaje de consulta estructurado**.

Las **clases que componen JPA** son las siguientes:

- **Persistence.**

La clase `javax.persistence.Persistence` contiene un **método auxiliar estático**, el cual usamos para tener un **objeto EntityManagerFactory** de forma **independiente** del que se obtiene a través de JPA.

- **EntityManagerFactory.**

La clase `javax.persistence.EntityManagerFactory`: de la cual **extraemos objetos de tipo EntityManager** usando el conocido patrón de Factory, (o patrón de diseño Factory). (Es una fábrica de objetos EntityManager).

- **EntityManager.**

La clase `javax.persistence.EntityManager` es la **interfaz JPA principal** para la persistencia de aplicaciones. Cada EntityManager puede **crear, leer, modificar y eliminar** (CRUD) un grupo de objetos persistentes.

- **Entity.**

La clase Entity es **una anotación Java**. La encontramos al **mismo nivel** que las clases Java **serializables**.

Cada una de estas entidades las representamos como **registros** diferentes en base de datos.

- **EntityTransaction.**

Permite **operaciones conjuntas** con datos persistentes, de tal forma, que pueden crear **grupos para persistir** distintas operaciones. Si todo el grupo realiza las operaciones sin ningún problema se persistirá, de lo contrario, todos los intentos fallarán. En caso de error, la base de datos realizará **rollback** y volverá al estado anterior.

- **Query.**

Cada proveedor de JPA ha implementado la interfaz `javax.persistence.Query` para **encontrar objetos persistentes** procesando ciertas condiciones de búsqueda. JPA utiliza **JPQL y SQL** para estandarizar el soporte de consultas. Podremos obtener un objeto Query **a través del EntityManager**.

Finalmente, aquellas **anotaciones JPA** (o anotaciones EJB 3.0) las encontraremos en **javax.Persistence**.

Muchos **IDE que admiten JDK5** (como Netbeans, Eclipse, IntelliJ IDEA) poseen **utilidades y complementos** para exportar clases de entidad usando las diferentes anotaciones JPA partiendo de la arquitectura de la base de datos.

## Java EE tutorial

Abordamos los componentes Java, los EJBs en general, así como el módulo de persistencia y los POJOs. En la página oficial de documentación de Oracle, encontramos un tutorial que nos enseña cómo crear nuestro propio Java Enterprise Edition desde cero. Este tutorial nos proporciona una visión general del tema y nos guía a través de la instalación del entorno, la preparación de la plataforma e incluso el despliegue con Apache.

Además, aborda los capítulos de Enterprise Beans y persistencia, donde podemos revisar los contenedores y profundizar en conceptos clave.

El módulo de persistencia, por ejemplo, explora la API y nos introduce en su funcionamiento. En la documentación, encontramos información sobre el Entity Manager, su interfaz y cómo utilizarlo. Además de estos temas, el tutorial también ofrece detalles sobre web services y otros servicios relacionados. Es una fuente de información muy completa que merece la pena explorar en detalle.

<https://docs.oracle.com/javaee/5/tutorial/doc/>

## Ejemplo de persistencia de 2 objetos

A modo recordatorio de persistencia se requiere realizar un ejemplo de persistencia de 2 objetos de clase persona cuyos atributos sean según constructor:

- Nombre
- Edad
- Dirección

Se requiere usar la clase EntityManagerFactory.

Crearemos distintos objetos de una clase persona y los persistiremos.

A continuación, mostraremos el código por medio del cual persistiremos 2 objetos de la clase persona.

Primero nos crearemos los diferentes objetos mediante constructor:

### Objetos persona

```
Persona persona1 = new Persona("pedro",25, "calle 1");  
Persona persona2 = new Persona("pedro",25, "calle 7");
```

Y después, procedemos a crear EntityManagerFactory, y a persistir los diferentes objetos:

### Persistiendo

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("FactoryPersonas");  
EntityManager em =  
    emf.createEntityManager();  
try {  
    em.getTransaction().begin();  
    em.persist(persona1);  
    em.persist(persona2);  
    em.getTransaction().commit();  
} catch (Exception e) {  
    e.printStackTrace();  
}finally {  
    em.close();  
}
```

## Empaquetado de componentes

Un **módulo J2EE** consta de **uno o más componentes** J2EE del **mismo tipo** de contenedor, y **descriptores** de implementación de componentes de este tipo.

El **descriptor** de despliegue del módulo EJB especifica los **distintos atributos** de nuestra transacción y la **autorización** de seguridad del EJB.

Los módulos JavaEE **sin descriptores** de implementación de aplicaciones se pueden implementar como **módulos separados**.

Antes de EJB 3.1, todos los **EJB** debían estar **empaquetados** en estos archivos.

Como buena parte de todas las **aplicaciones J2EE**, contienen un **front-end web** y un **back-end EJB**, lo que significa, que se debe crear un **.ear** que contenga una aplicación con **dos módulos**:

- **.war**
- y **ejb-jar**.

Esta es una buena práctica en el sentido de establecer una clara **separación estructural** entre la parte 'delantera' y la 'trasera.' No obstante, esta diferenciación será complicado delimitarla en aplicaciones simples.

Hay **cuatro tipos de módulos J2EE** para aplicaciones web EJB:

1. **Módulos EJB**, que contienen archivos con **clases** EJB y **descriptores** de despliegue EJB. El módulo EJB está empaquetado como un archivo **JAR** con extensión **.jar**.
2. El **módulo Web**, que contiene:
  - archivos con **Servlet**,
  - archivos **JSP**,
  - archivos de **soporte** de clases,
  - archivos **GIF** y **HTML**,
  - y **descriptores** de implementación de aplicaciones web.

El módulo web está empaquetado como un archivo **JAR** con una extensión **.war** (**archivo web**).

3. El **módulo de la aplicación cliente**, que contiene archivos con **clases y descriptores** de la aplicación cliente.

El módulo de la aplicación cliente está empaquetado como un archivo **JAR** con extensión **.jar**.

4. **Módulo adaptador de recursos**, que contiene:
  - todas las **interfaces**,
  - **clases**,
  - **bibliotecas** nativas,

- y otros **documentos** de Java,
- y sus **descriptores** de implementación del adaptador de recursos.

### Herramientas para desarrollo de componentes no visuales

Actualmente, los entornos de desarrollo de componentes más utilizados son:

- **Intelij**,
  - **NetBeans**,
  - y **Eclipse**.
- 
- **Eclipse** se utiliza principalmente para el desarrollo de **componentes Java** (como EJB),
  - y **Microsoft.NET** para el desarrollo de **ensamblajes (COM +, DCOM)**.

Estos entornos proporcionan una alternativa a la **generación automática de código** y facilitan enormemente el desarrollo de aplicaciones que utilizan componentes.

## Ejemplo de uso de Transaction

Trabajaremos sobre el objeto EntityTransaction.

Haremos uso de dicho objeto para realizar la persistencia en la capa de acceso a datos:

### Uso Entity Transaction

```
private static void agregarEntidad {  
    EntityManagerFactory entityManagerFactory =  
        Persistence.createEntityManagerFactory("PERSISTENCIA");  
    EntityManager entityManager = entityManagerFactory.createEntityManager();  
    EntityTransaction entityTransaction = entityManager.getTransaction();  
    entityTransaction.begin();  
    Student estudiante = new Student("Juan", "Lopez", "micorreo@codigos.com");  
    entityManager.persist(estudiante);  
    entityTransaction.commit();  
    entityManager.close();  
    entityManagerFactory.close();  
}
```

Como podemos observar en el código expuesto, englobamos en un **método privado** el método que nos hará dichas **funciones**. En él:

1. en primer lugar, creamos una **variable** de tipo **EntityManagerFactory** donde damos nombre a nuestra factoría.
2. A continuación creamos un **entityManager** del cual extraemos la **transacción** con **getTransaction()** para poder empezar a **realizar operaciones**.
3. **Creamos** un estudiante 'normal', como ejemplo.
4. Lo **persistimos**,
5. y con nuestra transacción hacemos **commit()** para que sea efectiva.

De este modo usaremos nuestro objeto para **persistir** los diferentes **componentes** y objetos Java en general.

