Desarrollo de aplicaciones con tecnología JEE5

Alfonso Rodríguez*, Reinaldo Jaimes, Oscar Pereira, Edwin Pérez, y Jhon Vásquez **
* Ingeniero de Sistemas. Especialista en Tecnología Avanzadas para el Desarrollo de Software.
Docente Programa Ingeniería de Sistemas. Co-investigador Grupo HYDRA.
** Estudiantes VI ivel de Ingeniería de Sistemas. Integrantes
Semillero de Investigación Kronos, Grupo HYDRA.
jhon_new2@hotmail.com

Palabras clave: Servlets, JSP (Java Server Pages), Enterprise Java Beans, Beans de Sesión, Beans de Entidad, API de persistencia

Key words: Servlets, JSP (Java Server Pages), Enterprise Java Beans, Session Beans, Entity Beans, Java Persistence API

Resumen

El semillero de investigación Kronos da a conocer en este artículo, sus principales avances en el aprendizaje y manejo de JEE5, presentando algunas definiciones, características, experiencias y dificultades encontradas en la construcción de aplicaciones con Java Enterprise Edition 5.0.

JEE5 es una plataforma que incluye tecnologías como: EJB, Servlets y JSP, encargada de implementar las diferentes responsabilidades de un sistema de información en cada capa (Presentación, lógica de negocio y datos).

Abstract

The research group Kronos wants to show in this article the main advances in learning and JEE5 management, introducing some definitions, characteristics, experiences and difficulties encountered in building applications with Java Enterprise Edition 5.0.

JEE5 is a platform that includes technologies such as: EJB, Servlets and JSP, in charge of implementing the different responsibilities of an information system in each layer (presentation, business logic and data)

I. INTRODUCCIÓN

Con la evolución de la tecnología, los desarrolladores de software tienen a su disposición un gran conjunto de herramientas para la construcción de sistemas de información robustos, que pueden ser implantados en diversos modelos de negocio.

El semillero de investigación Kronos, perteneciente al grupo de investigación HYDRA de UNISANGIL, ha incursionado en la línea de investigación en construcción de software, utilizando tecnología JEE5, con el fin de participar en la solución de problemas empresariales, dado que, para los desarrolladores actuales es de gran importancia construir sistemas de información que sean portables, distribuidos, transaccionales, que ayuden a incrementar la velocidad, seguridad y fiabilidad de las tecnologías usadas del lado del servidor.

JEE 5 es una plataforma que proporciona un conjunto de API (Interface de Programación de Aplicaciones) que reduce el tiempo de desarrollo y la complejidad, y mejora el rendimiento de las aplicaciones, a través de un modelo de programación simplificado.

II. JEE 5 (JAVA ENTERPRISE EDITION 5.0)

JEE 5 está soportada por el lenguaje de programación Java y es ejecutado sobre una máquina virtual. Esta tecnología fue desarrollada para soportar servicios empresariales; estos tipos de aplicaciones son complejos, acceden a diversas fuentes de datos y son usados desde diversos tipos de clientes.

La plataforma JEE utiliza un modelo de aplicación multicapas. La lógica está dividida en componentes que, dependiendo de su función y responsabilidad, son instalados en diferentes nodos computacionales dependiendo de la capa a la que pertenezca [1].

Un componente es una unidad de software autocontenida que es ensamblada en una aplicación JEE con sus clases, archivos y que se comunican con otros componentes [2]. La especificación Java define que las aplicaciones *cliente* (stand alone) y *applets* se ejecutan en el cliente; los componentes Servlets, JavaServer Faces (JSF) y JavaServer Pages (JSP) son componentes web y se ejecutan en el servidor de aplicaciones dentro del contenedor web. Los Enterprise JavaBeans implementan la lógica de negocio y se ejecutan en el servidor de aplicaciones dentro del contenedor de EJB [3].

Cada uno de los componentes se comunica entre sí dependiendo de la capa en la que se encuentren. Por ejemplo, el cliente (browser) hace una petición (request) a la capa web y ésta a su vez se comunica con la capa de negocio, los EJB se encargan de satisfacer la petición de la capa anterior y retorna una respuesta a la capa web; ésta, entonces, responde (response) al cliente.

Una cliente *stand alone* no se comunica con la capa web si no que lo hace directamente con la capa de negocio, tal como se indica en la siguiente figura. (En la figura no se muestra la comunicación desde un cliente móvil, la cual puede ser con la capa web o directamente con la capa de negocio) [4].

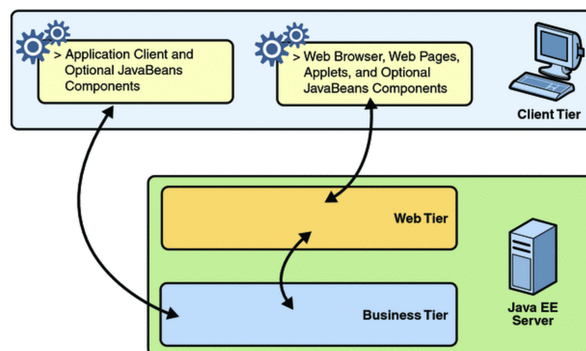

Fig. 1. Comunicación entre capas [3]

JEE ofrece soporte a una amplia gama de API. A continuación se presenta un resumen para ilustrar algunas tecnologías que soporta: EJB, JavaServer Faces, Java Message Service, Java Transaction API, Java API for XML Processing, Java API for XML Web Services, Java Architecture for XML Binding, Java DataBase Connectivity, Java Persistence API, Java Naming and Directory Interface, Java Authentication and Authorization Service, entre otros [3].

Una de las características más importantes que tiene JEE son los EJB 3.0; están compuestos por Beans de Sesión, Beans de Entidad y Beans de Mensaje y fueron desarrollados para simplificar la construcción de aplicaciones empresariales.

Los Session Beans son componentes de negocio que implementan servicios ofrecidos por la aplicación, definidos en una interface de servicio (POJI), esta interface puede ser accedida de forma local (@Local) o remota (@Remote). Los Session Beans representan un cliente de la aplicación o una sesión de un cliente, no son compartidos por los clientes ni persistentes. Los Session Bean pueden ser sin estado (@Stateless) o con estado (@Stateful).

Un Bean (@Stateless) conserva su estado únicamente durante la ejecución de un método invocado. El contenedor de EJB elige el bean que va a ejecutar un método; son apropiados para brindar escalabilidad de aplicaciones, ya que puede soportar múltiples clientes.

Un Bean con estado (@StateFul) representa la interacción con un cliente específico y el Bean. El estado del Bean (atributos) se mantiene mientras dure la sesión. Al terminar la sesión el estado no se conserva; el contenedor puede almacenarlos en memoria secundaria y pueden implementar servicios web [1], [3].

Almacenar la información de una aplicación en un medio físico, ya sea en una base de datos, un archivo de texto, etc., se entiende como persistencia de datos. Para este propósito existe el API de persistencia de Java

(JPA), que es un mecanismo de persistencia de datos que ha sido introducido por EJB para proporcionar Plain Old Java Object (POJO). Los POJOs permiten implementar el modelo del dominio en términos de objetos y realizar el mapeo Objeto/Relacional, facilitando a los desarrolladores el manejo y almacenamiento de datos relacionales [5], [6].

El API de persistencia de Java está compuesto por tres partes: el API de persistencia, el lenguaje de consultas de persistencia y los metadatos para el mapeo Objeto/Relacional [5].

El API de Persistencia de Java usa el término de entidad para definir clases persistentes; se denotan con la anotación @Entity. Estas clases serán mapeadas a la base de datos relacional en el transcurso del programa.

Las entidades son administradas por el administrador de entidades (javax.persistence.EntityManager). El administrador de entidades es un API que ofrece los servicios para trabajar con una entidad; es asociado con un contexto persistente (@PersistenceContext), el cual define el ámbito en el que las instancias de la entidad se crearán, guardarán y eliminarán.

El API de EntityManager crea y elimina las instancias persistentes de la entidad, encuentra las entidades por la clave principal de la entidad, y permite que las consultas se ejecuten en las entidades [5].

Las consultas se implementan a través de Java Persistence Query Languaje (JPQL), un lenguaje que permite consultar entidades, no tablas y no requiere conocer la implementación de la base de datos. Las consultas se ejecutan por medio de los métodos EntityManager.createQuery y EntityManager.createNamedQuery [1], [5].

El Mapeo Objeto-Relacional permite reducir la brecha entre el modelo de objetos del dominio y la base de datos, transformándolo de forma automática, dependiendo de las anotaciones que se agreguen a la entidad y de las relaciones entre ellas.

Las entidades se construyen a partir de POJOs; estos consisten en objetos java que no implementan interfaces especiales, ni tampoco dependen de frameworks. Esta programación es muy utilizada en modelos EJB, por sus múltiples beneficios, dentro de los cuales podemos destacar que un POJO simplifica el desarrollo, porque en lugar de que un programador piense en todo (lógica de negocio, persistencia, transacciones, etc.), puede concentrarse en desarrollar una cosa a la vez; un POJO también acelera el desarrollo, ya que un programador puede probar su lógica de negocio sin desplegar el servidor de aplicaciones [1], [5], [6].

III. SERVIDOR DE APLICACIONES

Un servidor de aplicaciones, básicamente, es un servidor en una o más computadoras que ejecutan ciertas aplicaciones. Usualmente es un dispositivo de software que proporciona servicios de aplicación a la capa cliente; la mayor parte de las funciones de la lógica de negocio son gestionadas por un servidor de

aplicaciones. La centralización y la disminución de la complejidad en el desarrollo de aplicaciones son unos de los beneficios que se obtiene al usar esta tecnología [3].

En la arquitectura JEE son muy importantes los Enterprise JavaBeans; el servidor EJB facilita el desarrollo de aplicaciones y permite que sean escalables, transaccionales y portables, al tiempo que se encarga de la gestión de transacciones y demás requerimientos no funcionales, como seguridad, concurrencia, acceso remoto, manejo de persistencia, entre otros. La única preocupación para el desarrollador es construir la lógica de negocio [3].

JBOSS es un ejemplo de servidor de aplicaciones, es una plataforma de servicios java para el desarrollo de aplicaciones, el cual permite desplegar aplicaciones y servicios web en una arquitectura orientada a servicios.

IV. ARQUITECTURA POR CAPAS

Una arquitectura es el resultado de un proceso de diseño de un sistema específico que tiene en cuenta las funciones de los componentes, sus interfaces, interacciones y sus limitaciones. Las aplicaciones modernas se desarrollan por capas, logrando así que cada capa tenga sus respectivas responsabilidades y que haya cohesión entre las mismas [4].

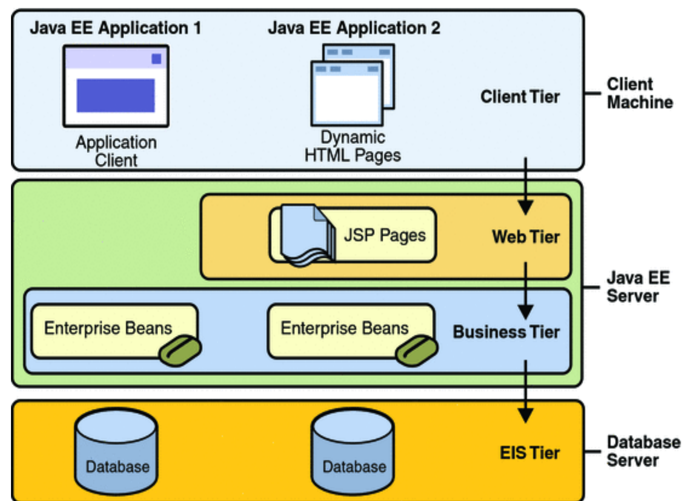

Fig. 2. Arquitectura por capas [3]

La capa de negocio es la encargada de recibir los datos de la capa de presentación; utiliza los servicios de la capa de datos durante la ejecución del proceso para consultar y actualizar la información que requiera; tiene la lógica de negocio; recolecta información y la procesa; transforma datos; determina qué funcionalidades pueden ejecutarse sobre un conjunto de datos, dependiendo del estado del proceso.

La plataforma JEE implementa la lógica de negocio a través de Componentes EJB. Esta especificación permite al desarrollador concentrarse en la lógica de negocio y el servidor EJB se encarga de requerimientos no funcionales [1].

La capa de Base de Datos es donde se almacenan los datos, se encarga de acceder a los mismos; está formada por uno o más gestores de bases de datos. Esta capa recibe solicitudes de almacenamiento o recuperación de información desde la capa de negocio [1], [5].

La capa de Presentación presenta el sistema al usuario, le comunica la información y captura datos y ordenes del usuario. Esta capa se comunica únicamente con la capa de negocio; también es conocida como interfaz gráfica —GUI— (Interfaz Gráfica de Usuario). Hay que tener en cuenta que existen muchas tecnologías con las cuales podemos acceder a una aplicación. Más adelante explicaremos la capa de presentación aplicada en la web [7].

La capa de presentación está compuesta por componentes web y aplicaciones web, que están conformadas por dos tipos de aplicación: orientada a la presentación y orientada a los servicios.

Las aplicaciones orientadas a la presentación generan las páginas web interactivas, crean un contenido dinámico a las respuestas y peticiones hechas por el usuario. Los componentes web son objetos del lado del servidor que manejan eventos del cliente, entrada de datos y tienen un ciclo de vida. Las aplicaciones orientadas a servicios implementan el punto final de un servicio web [5], [8].

Los componentes web más usados son:

- JSP (Java Server Pages)

Es una tecnología Java que permite generar contenido dinámico web, en forma de documentos HTML, XML o de otro tipo. JSP permite la utilización de código Java mediante scripts. Además, es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Los JSP son traducidos a Servlets y su ciclo de vida es el mismo al del Servlet [3].

- Servlets

Los Servlets son una tecnología de componentes diseñados para servir a las necesidades de los desarrolladores web que responden a peticiones HTTP y para generar contenido dinámico. Los Servlets son la tecnología más antigua y popular.

Se presentó como parte de la plataforma JEE; es especialmente diseñado para ofrecer contenido dinámico desde un servidor web, generalmente HTML y es también la base de las tecnologías JSP y JSF.

Al implementar una interfaz, el Servlet es capaz de interpretar los objetos de tipo HttpServletRequest y HttpServletResponse, los cuales contienen la información de la página que invocó al Servlet [3].

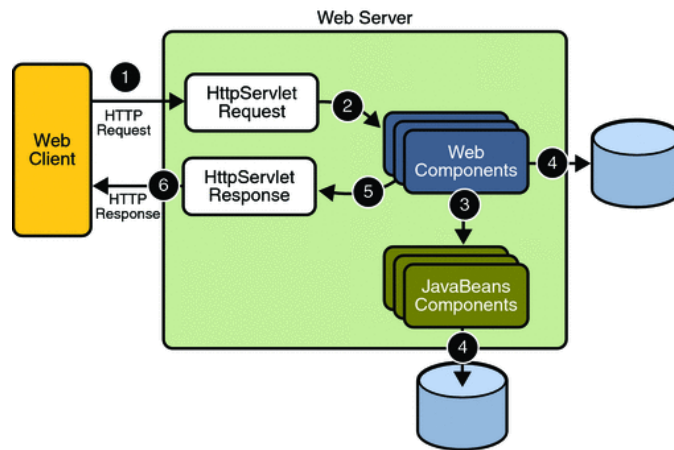

Fig. 3. Ciclo de vida de una petición Web [3].

El ciclo de vida de un Servlet está compuesto por las siguientes actividades: cargar el Servlet, crear una instancia del Servlet, inicializar el Servlet, invocar el método init, manejar la interacción del cliente con el Servlet, destruir el Servlet.

V. EXPERIENCIAS

El semillero Kronos inició su proceso de investigación con una revisión bibliográfica y adquisición de las herramientas necesarias; después se realizaron pruebas de las tecnologías más importantes de JEE, por lo tanto fue necesario seleccionar un contexto de negocio apropiado para poner en práctica lo aprendido. Para este caso, se tomó parte del modelo de objetos de negocio de una biblioteca, e implementó la persistencia. La lógica de negocio fue correspondiente a: servicio de consultas, servicios CRUD (Create, Retrive, Update, Delete) para cada uno de los objetos del modelo, componentes web para ingresar información usando JavaServer Faces y pruebas unitarias de los servicios a través del FrameWork JUnit.

A continuación se presenta el diagrama de clases, el cual ilustra el modelo de objetos del domino.

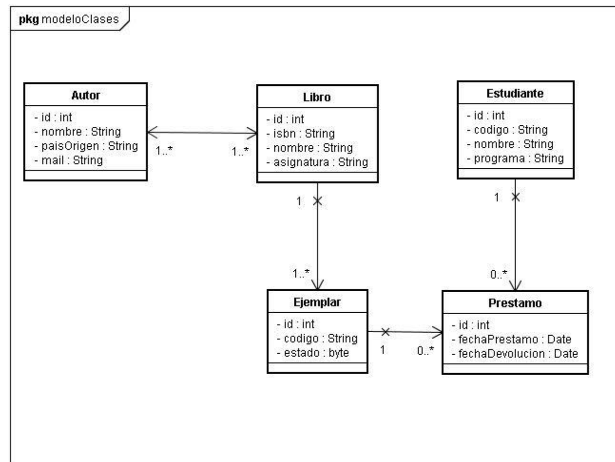

Fig. 4. Modelo de Clases

En el diseño de la arquitectura se definieron tres capas: datos, lógica de negocio y presentación. Además se utilizaron dos patrones de diseño: el patrón Bussines Object para pasar objetos entre las diferentes capas de la aplicación y el patrón Delegate para conectar la capa web con la capa de negocio y poder realizar una conexión remota a través de JDNI y obtener referencia al EntityManager por inyección utilizando PersistenceContext [8].

Cada clase (POJO) fue creada en dos paquetes: el primer paquete, *biblioteca.entities* (Entidades persistentes) y el segundo, *biblioteca.bo* (Business Object).

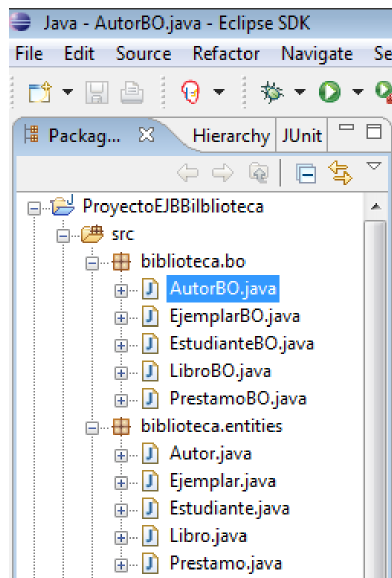

Fig. 5. Explorador de Paquetes Eclipse

El siguiente paso fue incluir las anotaciones de persistencia a las clases que se encuentran en el paquete

biblioteca.entities, que serán mapeadas de forma automática, entre las que podemos destacar: @Entity, indica la entidad de persistencia; @Table, crea una tabla en la base de datos; @Id, indica un identificador único para cada instancia de una entidad; @Column, crea las columnas dentro de una tabla en la base de datos; @OneToOne, esta anotación se usa en relaciones uno a uno; @OneToMany, se usa en relaciones de uno a muchos; @ManyToOne, se usa en relaciones de muchos a uno; @ManyToMany, se usa en relaciones muchos a muchos.

A continuación se presenta un fragmento de código en Java que permite mostrar la implementación de una entidad persistente.

```
/**
 * Representa el Entity Autor.
 */
@Entity
@Table( name = "autores" )
@EntityListeners( AutorListener.class )
public class Autor implements Serializable{


    //--------------------------------
    //Constantes.
    //--------------------------------


    /**
     * Constante de Serialización.
     */
    private static final long serialVersionUID = -7344814398448919lL;
```

```
/**
 * Retorna el Id del Autor.
 * @return int
 */
@Id
@GeneratedValue
@Column( name = "idAutor" )
public int getId( ) {
    return id;
}


/**
 * Cambia el Id del Autor
 * @param id Nuevo Id del Autor.
 */
public void setId( int id ) {
    this.id = id;
}


/**
 * Retorna el nombre del Autor.
 * @return String
 */
@Column( name = "nombreAutor", nullable = false )
public String getNombre( ) {
    return nombre;
}
```

```java
/**
 * Retorna los prestamos del ejemplar.
 * @return Collection<Prestamo>
 */
@OneToMany( cascade = javax.persistence.CascadeType.ALL,
        fetch = FetchType.LAZY, mappedBy = "ejemplar" )
public Collection<Prestamo> getPrestamo( ) {
    return prestamo;
}
```

```java
/**
 * Retorna el ejemplar del préstamo.
 * @return Ejemplar
 */
@ManyToOne
@JoinColumn( name = "idEjemplar", nullable = false )
public Ejemplar getEjemplar( ){
    return this.ejemplar;
}
```

```java
/**
 * Retorna los autores del Libro.
 * @return Collection<Autor>
 */
@ManyToMany( cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        fetch = FetchType.LAZY )
@JoinTable( name="Libros_Autor",
    joinColumns=
        @JoinColumn( name="idLibro",
        referencedColumnName="idLibro", updatable = false ),
        inverseJoinColumns=
        @JoinColumn( name="idAutor",
        referencedColumnName="idAutor", updatable = false )
)
public Collection<Autor> getAutores( ) {
    return autores;
}
```

Fig. 6. Ejemplo Entity Bean

Para convertir las clases Entities a BO (Business Object) se implementó el método toBO() en cada clase de entidad, como se muestra a continuación.

```java
/**
 * Convierte el Entity Ejemplar a un BO
 * @return EjemplarBO
 */
public EjemplarBO toBO( ){
    EjemplarBO ejemplarBO = new EjemplarBO( );

    ejemplarBO.setId( this.getId( ) );
    ejemplarBO.setCodigo( this.getCodigo( ) );
    ejemplarBO.setEstado( this.getEstado( ) );

    return ejemplarBO;
}
```

Fig. 7. Ejemplo de método para transformar a Bussines Object

Luego se construyeron los @NamedQueries y @NamedQuery, para las consultas dinámicas, junto con las excepciones (clases BibliotecaException y PersistenciaException) necesarias para la gestión de errores de la aplicación.

```
@NamedQueries( value = { @NamedQuery
        (
            name = "findAutorByPK",
            query = "SELECT a FROM Autor a WHERE a.id = :AutorId"
        ),

        @NamedQuery
        (
                name = "findAllAutores",
                query = "SELECT a FROM Autor a"
        )
})


/**
 * Representa el Entity Autor.
 */
@Entity
@Table( name = "autores" )
@EntityListeners( AutorListener.class )
public class Autor implements Serializable{
```

Fig. 8. Ejemplo de consulta con Lenguaje JPQL

A continuación se creó el paquete biblioteca.services; en este paquete se crearon dos interfaces IServiciosBiblioteca e IServiciosConsultasBiblioteca con la definición abstracta de los servicios de la aplicación. Además, se construyeron dos clases (ServiciosBiblioteca y ServiciosConsultasBiblioteca) que implementan estos métodos, anotadas con @Stateless y a las interfaces se les agregó la anotación @Remote.

```
package biblioteca.services;

import java.util.Date;

 * FUNDACIÓN UNIVERSITARIA DE SAN GIL - UNISANGIL.

/**
 * Interface que expone los servicios que prestará el bean de sesión Biblioteca.
 */
@Remote
public interface IServiciosBiblioteca {
```

```
package biblioteca.services;

import java.util.Collection;

 * FUNDACIÓN UNIVERSITARIA DE SAN GIL - UNISANGIL.

/**
 * Bean de sesión sin estado que maneja los servicios asociados al manejo de Biblioteca.
 */

@Stateless
public class ServiciosBiblioteca implements IServiciosBiblioteca{
```

Fig. 9. Ejemplo de anotaciones @Remote y @Stateless.

Después se creó el paquete biblioteca.test y en su interior se implementaron las clases TestServiciosBiblioteca y TestServiciosConsultasBiblioteca, que permiten hacer las pruebas unitarias a la lógica de negocio.

A continuación se creó el paquete biblioteca.web.beans, donde se implementó —por ejemplo— la clase AutorBean, con la cual podemos obtener la información recibida del JSF y convertirla en un objeto que va al interior del sistema. Luego, se creó el paquete biblioteca.web.delegate, con las clases

ServiciosBibliotecaDelegate y ServiciosConsultasBibliotecaDelegate. Estas clases son las intermediarias entre la capa de presentación y la lógica de negocio.

Por último, se empaquetó y desplegó la aplicación utilizando JBoss como servidor de aplicaciones.

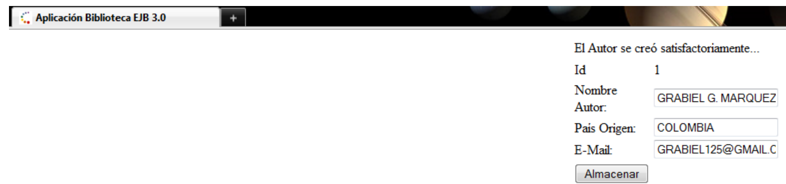

Fig. 10. Interfaz gráfica de usuario para ingresar un autor

Además, podemos verificar el mapeo a la base de datos y si realmente se almacenaron los datos del autor ingresados por el usuario.

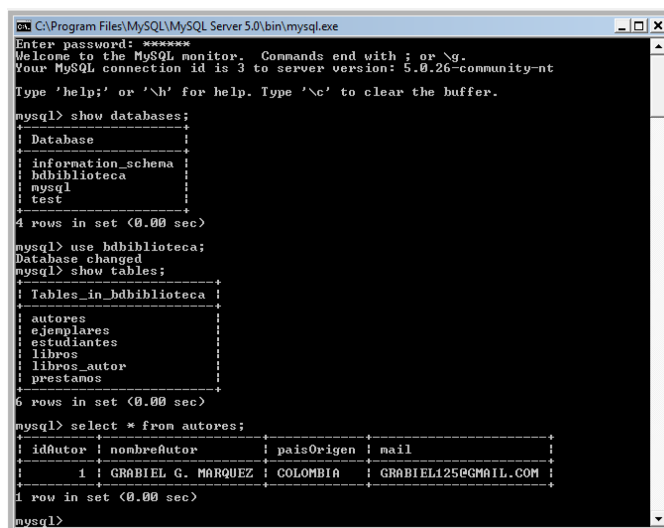

Fig. 11. Consola SGBD MySQL

VI. CONCLUSIONES

Con JEE se puede escribir código eficiente, reduciendo a la mitad el número de líneas de código respecto a otras tecnologías como J2EE.

Es posible modelar en términos de entidades persistentes y no de tablas de la base de datos; desaparece JDBC y otros patrones burocráticos.

Es viable construir aplicaciones orientadas a objetos y no procedimentales, integrando gran diversidad de frameworks. Es una herramienta que trabaja con una arquitectura por capas y distribuida, cada una con responsabilidades bien definidas, proporcionando alta cohesión y bajo acoplamiento.

Del mismo modo, el API de Persistencia permite hacer un mapeo objeto-relacional entre las clases y la base de datos de la aplicación de forma automática; mejora el lenguaje EJB-QL; y añade soporte para SQL nativo. Se añade el EntityManager para la ejecución de operaciones sobre entidades.

Finalmente, los EJB facilitan las tareas de desarrollo, evitando la implementación de interfaces Home y de Componente, pasando a ser un POJO, por lo que simplifican la configuración a través de valores por defecto y el uso de anotaciones de metadatos en lugar de descriptores de despliegue. Además, soporta la inyección de dependencias para usar EJB y variables de entorno, la búsqueda e invocación de métodos es simplificada y los requerimientos no funcionales son responsabilidad del Servidor de Aplicaciones.

REFERENCIAS

[1] S. Haines. *Pro Java EE 5 Performance Management and Optimization*. New York, Estados Unidos: Apress, 2006.

[2] K. Zaman y C. Umrysh. *Developing Enterprise Java Applications with J2EE and UML*. Indianapolis, Estados Unidos: Addison Wesley, 2001.

[3] Sun Microsystems, Inc. *The Java EE 5 Tutorial for Sun Java System Application Server 9.1.* Santa Clara, Estados Unidos: Network Circle, 2008.

[4] P. R. Allen y J. J. Bambara. *SCEA Sun Certified Enterprise Architect for Java EE Study Guide*. Nueva York: McGraw-Hill, 2007.

[5] M. Keith y M. Schincariol. *Pro EJB 3Java Persistence API*. Nueva York, United States: Apress, 2006.

[6] B. Sam-Bolden. *Beginning POJOs from Novice to Professional*. Nueva York, Estados Unidos: Apress, 2006.

[7] J. Tulach. *Practical API Design. Confessions of a Java Framework Architect*. Nueva York, Estados Unidos: Apress, 2008.

[8] A. Bien. *Real World Java EE Patterns Rethinking Best Practices,* http://press.adam-bien.com/ Revisado: 11-Mayo-2010.