

Creación de la Clase de Usuarios



Creación de la Clase de Usuarios en MySql

Vamos a crear la clase de **Usuario** y **Persona** de manera automática utilizando los **Wizards de NetBeans**. Esto nos permitirá generar las clases de entidad desde la base de datos y configurarlas correctamente con JPA.

1 Eliminar la Clase Persona Existente

Antes de generar las nuevas clases de entidad, eliminamos la clase `Persona` que habíamos creado previamente en el paquete `domain`.

◆ **Pasos:**

1. En **NetBeans**, ve al paquete `sga.domain`.
2. Haz **clic derecho** sobre `Persona.java` y selecciona `Delete`.
3. Confirma la eliminación.

2 Generar las Clases de Entidad desde la Base de Datos

◆ Pasos:

1. Haz clic derecho en el paquete `sga.domain`.
 2. Selecciona "Entity Classes from Database".
 3. En **DataSource**, selecciona `Local Data Source` y elige la **conexión a MySQL** que creamos previamente.
 4. Deberían aparecer las tablas disponibles (`persona` y `usuario`).
 5. Selecciona ambas tablas (`persona`, `usuario`) y haz clic en "Add".
 6. Presiona "Next".
-

3 Opciones de Generación

- ◆ En esta sección, NetBeans nos ofrece varias opciones para la generación de clases.

✓ Se deja activado:

- **Generate Named Query Annotations for Persistent Fields** → Esto genera automáticamente consultas `@NamedQuery` en cada clase de entidad, lo que nos permite realizar búsquedas predefinidas en la base de datos sin necesidad de escribir JPQL manualmente.

✗ ✗ Se deselecciona:

- **Generate JAXB Annotations** → No lo usaremos por ahora, ya que esta opción se usa para convertir objetos en XML. Se puede habilitar más adelante si necesitamos integración con servicios web.

✗ ✗ No seleccionamos:

- **Generate mappedSuperclasses instead of Entities** → No se selecciona porque esta opción genera una clase padre abstracta en lugar de entidades completas, lo cual no es necesario en nuestro caso.
7. Haz clic en "Next".
-

4 Configurar Opciones de Mapeo

Dejamos las opciones por defecto:

- **Collection Type:** `java.util.List` → Se usa `List` para manejar colecciones de datos relacionados, lo cual es estándar en JPA.
- **✗ Use Column Names in Relationships (se deselecciona)** → Esto evita que las relaciones usen los nombres de columna de la base de datos, permitiendo una nomenclatura más clara en el código.
- **✓ Use Defaults if Possible (se deja activado)** → Permite que NetBeans **use valores predeterminados** cuando sea posible, facilitando la configuración automática.

8. Haz clic en "Finish".
-

5 Revisar las Clases Generadas

NetBeans generará automáticamente las clases **Persona.java** y **Usuario.java** en `sga.domain`.

✗ Modificamos la clase `Persona.java`:

- **Agregamos un constructor** con los campos nombre, apellido, email y telefono (sin `idPersona` ni `usuarioList`).
- **Regeneramos el método `toString()`**, omitiendo la lista de usuarios, ya que cargar una colección grande puede afectar el rendimiento.
- **Simplificamos los imports** con `import jakarta.persistence.*;`.

```
package sga.domain;

import jakarta.persistence.*;
import jakarta.validation.constraints.Size;
import java.io.Serializable;
import java.util.List;

@Entity
@NamedQueries({
    @NamedQuery(name = "Persona.findAll", query = "SELECT p FROM Persona p"),
    @NamedQuery(name = "Persona.findByIdPersona", query = "SELECT p FROM Persona p
WHERE p.idPersona = :idPersona"),
    @NamedQuery(name = "Persona.findByNombre", query = "SELECT p FROM Persona p WHERE
p.nombre = :nombre"),
    @NamedQuery(name = "Persona.findByApellido", query = "SELECT p FROM Persona p
WHERE p.apellido = :apellido"),
    @NamedQuery(name = "Persona.findByEmail", query = "SELECT p FROM Persona p WHERE
p.email = :email"),
    @NamedQuery(name = "Persona.findByTelefono", query = "SELECT p FROM Persona p
WHERE p.telefono = :telefono")})
public class Persona implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
```

```
@Column(name = "id_persona")
private Integer idPersona;
@Size(max = 45)
private String nombre;
@Size(max = 45)
private String apellido;
// @Pattern(regexp="[a-zA-Z0-9!#$%&'*+/=?^_-`{|}~-]+(?:\\.[a-zA-Z0-9!#$%&'*+/=?^_-`{|}~-]+)*@(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?\\.)+[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?", message="Invalid email")//if the field contains email address consider using this annotation to enforce field validation
@Size(max = 45)
private String email;
@Size(max = 45)
private String telefono;
@OneToMany(mappedBy = "persona")
private List<Usuario> usuarioList;

public Persona() {
}

public Persona(String nombre, String apellido, String email, String telefono) {
    this.nombre = nombre;
    this.apellido = apellido;
    this.email = email;
    this.telefono = telefono;
}

public Persona(Integer idPersona) {
    this.idPersona = idPersona;
}

public Integer getIdPersona() {
    return idPersona;
}

public void setIdPersona(Integer idPersona) {
    this.idPersona = idPersona;
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getApellido() {
    return apellido;
}

public void setApellido(String apellido) {
    this.apellido = apellido;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
```

```

    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public List<Usuario> getUsuarioList() {
        return usuarioList;
    }

    public void setUsuarioList(List<Usuario> usuarioList) {
        this.usuarioList = usuarioList;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (idPersona != null ? idPersona.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Persona)) {
            return false;
        }
        Persona other = (Persona) object;
        if ((this.idPersona == null && other.idPersona != null) || (this.idPersona != null && !this.idPersona.equals(other.idPersona))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "Persona{" + "idPersona=" + idPersona + ", nombre=" + nombre + ", apellido=" + apellido + ", email=" + email + ", telefono=" + telefono + '}';
    }
}

```

❖ No incluimos `usuarioList` en `toString()`, porque cargar muchas relaciones puede hacer que imprimir un objeto sea lento o cause problemas de rendimiento.

❖ Modificamos la clase `Usuario.java`:

- **Agregamos un constructor** con los campos `username` y `password`.
- **Regeneramos el método `toString()`**, esta vez **incluyendo la relación `persona`**, ya que es una sola entidad y no una lista.

```
package sga.domain;

import jakarta.persistence.*;
import jakarta.validation.constraints.Size;
import java.io.Serializable;

@Entity
@NamedQueries({
    @NamedQuery(name = "Usuario.findAll", query = "SELECT u FROM Usuario u"),
    @NamedQuery(name = "Usuario.findByIdUsuario", query = "SELECT u FROM Usuario u
WHERE u.idUsuario = :idUsuario"),
    @NamedQuery(name = "Usuario.findByUsername", query = "SELECT u FROM Usuario u
WHERE u.username = :username"),
    @NamedQuery(name = "Usuario.findByPassword", query = "SELECT u FROM Usuario u
WHERE u.password = :password")})
public class Usuario implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id_usuario")
    private Integer idUsuario;

    @Size(max = 45)
    private String username;
    @Size(max = 45)
    private String password;

    @JoinColumn(name = "id_persona", referencedColumnName = "id_persona")
    @ManyToOne
    private Persona persona;

    public Usuario() {
    }

    public Usuario(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public Usuario(Integer idUsuario) {
        this.idUsuario = idUsuario;
    }

    public Integer getIdUsuario() {
        return idUsuario;
    }

    public void setIdUsuario(Integer idUsuario) {
        this.idUsuario = idUsuario;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

```
public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public Persona getPersona() {
    return persona;
}

public void setPersona(Persona persona) {
    this.persona = persona;
}

@Override
public int hashCode() {
    int hash = 0;
    hash += (idUsuario != null ? idUsuario.hashCode() : 0);
    return hash;
}

@Override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the id fields are not set
    if (!(object instanceof Usuario)) {
        return false;
    }
    Usuario other = (Usuario) object;
    if ((this.idUsuario == null && other.idUsuario != null) || (this.idUsuario != null && !this.idUsuario.equals(other.idUsuario))) {
        return false;
    }
    return true;
}

@Override
public String toString() {
    return "Usuario{" + "idUsuario=" + idUsuario + ", username=" + username + ", password=" + password + ", persona=" + persona + '}';
}
}
```

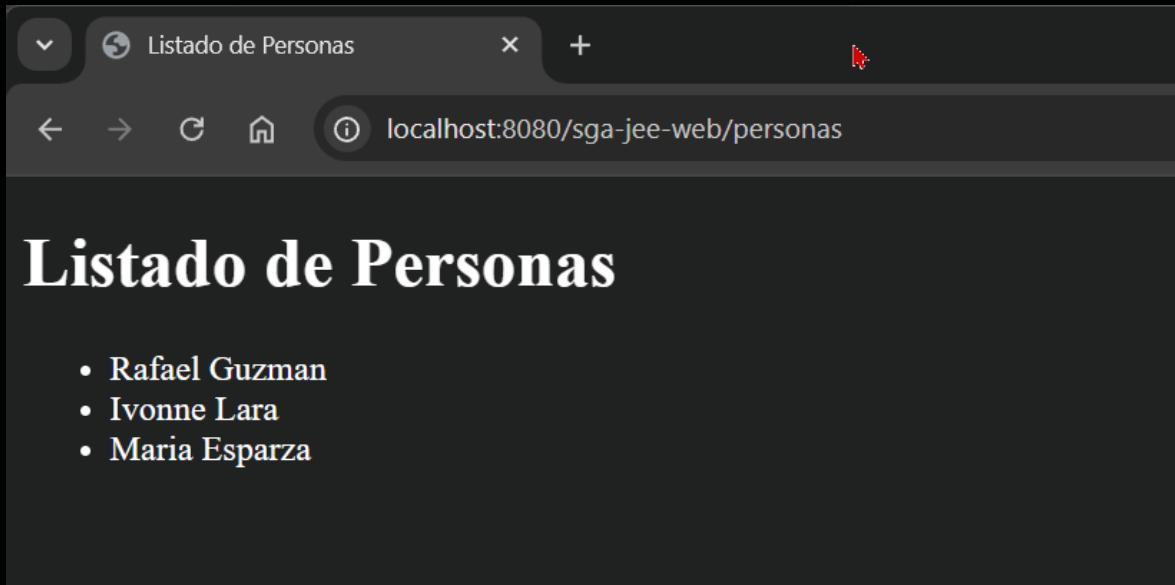
★ Aquí sí incluimos `persona` en `toString()`, porque es un solo objeto y no una lista grande de elementos.

6 Ejecutar el Proyecto

1. Hacemos **Clean and Build** en el proyecto (`sga-jee-web`).
2. Si hay errores, detenemos **GlassFish** y lo reiniciamos.
3. Ejecutamos el proyecto (`Run`).
4. Abrimos el navegador y verificamos la lista de personas:

<http://localhost:8080/sga-jee-web/personas>

- Si todo está bien, la lista de personas debería mostrarse correctamente.



📌 Conclusión

- ✓ Generamos automáticamente las clases de **Persona** y **Usuario** desde la base de datos.
- ✓ Configuramos las relaciones **@OneToMany** y **@ManyToOne** correctamente.
- ✓ Agregamos validaciones y **NamedQueries** para consultas optimizadas.
- ✓ Modificamos los métodos `toString()` para evitar problemas de rendimiento con listas grandes.
- ✓ Ejecutamos el proyecto y verificamos la conexión con MySQL y GlassFish.

Con esto, nuestro **sistema SGA** está listo para manejar **múltiples entidades y sus relaciones con JPA**. 

Saludos!

Ing. Ubaldo Acosta

Fundador de GlobalMentoring.com.mx