



(P7) Aplicación backend con tecnología Java en servidores de aplicaciones

PRODUCTO 2

Persistencia de datos

(Conv_0921)

Consultor: Prof. Paco Gómez Arnal

GRUPO - THE BINARIES

Asier Uruñuelas Sánchez
Francisca M. Rodríguez Vázquez
Pau Egea Cortés
Sofía Figueroa Arocha

ÍNDICE

1. **Diagrama de clases que describa su estructura estática a partir de la descripción del programa de sopa de letras descrito en el Producto 1.**
2. **Diagrama modelo-relacional, a partir del diagrama de clases, que permita la persistencia de las clases en una base de datos relacional. Hay que tener en cuenta que los jugadores estarán almacenados en LDAP.**
3. **Instalar la Base de datos MariaDB o MySQL en caso de no tenerla instalada.**
4. **Crear la base de datos que se ha diseñado en el diagrama modelo-relacional en la Base de datos.**
5. **Crear las clases y estructura la MVC que se utilizará para la realización del programa Sopa de letras teniendo en cuenta que el programa se debe realizar en JSP para las vistas y Servlets para los controladores.**
6. **Crear la persistencia de los datos del programa utilizando el patrón de diseño DAO mediante Java y JDBC.**

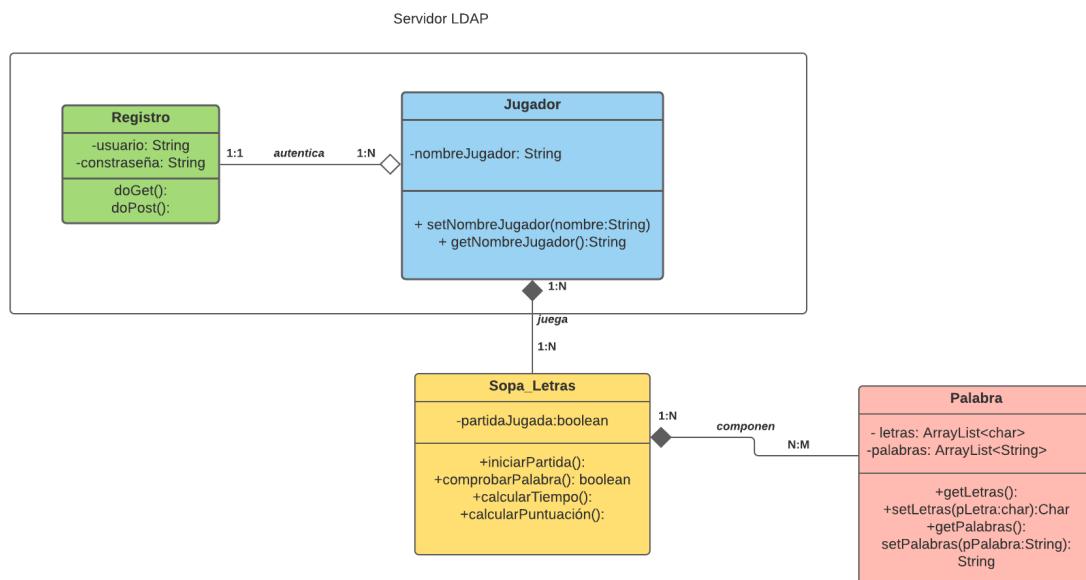
1. Diagrama de clases que describa su estructura estática a partir de la descripción del programa de sopa de letras que se ha descrito en el Producto 1.

En el diagrama de clases que hemos realizado se detallan las clases que se usarán para la implementación de nuestra aplicación. Para ello, hemos creído oportuno, incorporar una clase que recoja la autenticación o logueo del jugador, almacenados en un servidor LDAP.

Las otras dos clases son las siguientes:

- ★ Clase Sopa_Letras: Cuenta con arrays bidimensionales o matrices y las funciones principales de la aplicación (iniciar partida, comprobar palabra, calcular tiempo y calcular puntuación).
- ★ Clase Palabra: Elegirá las palabras para cada juego. Para ello, consideramos que necesita una lista de caracteres, que normalmente es el tipo primitivo que se suele usar para el establecimiento de una letra, y así crear un marco de búsqueda y restringirlo a sólo letras: después la idea sería buscar una función para llamar al formato UTF-8, o más concretamente la idea es restringirlo a la especificación ASCII (que es el estándar para el alfabeto latino).

Por consiguiente, hemos especificado las relaciones de asociación con el fin de esquematizar con la mayor aproximación posible, la intensidad de relación que hay entre las distintas clases. Más concretamente, cabe destacar que la relación entre Sopa de Letras y Palabra es más intensa (de composición, que es el rombo negro), ya que la clase palabra no tendría sentido sin la existencia de la clase Sopa de Letras, ni la Sopa de Letras tendría razón de existencia sin la existencia de la clase jugadores. Las relaciones de cardinalidad son las que entendemos a partir de la realidad de abstracción que observamos en el mundo real: mírese la siguiente imagen.



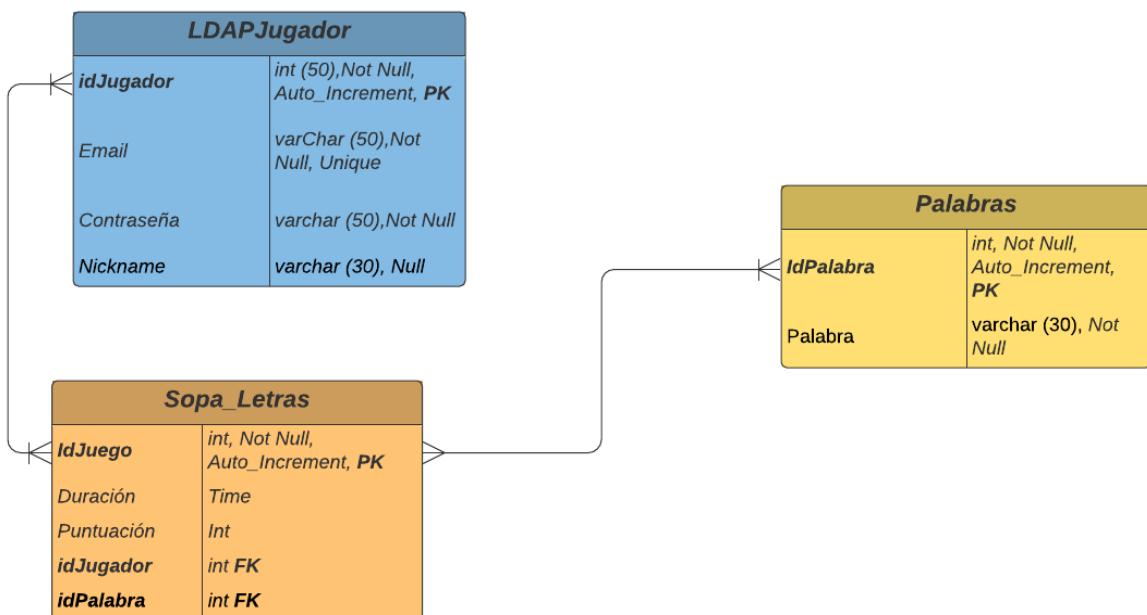
2. Diagrama modelo-relacional, a partir del diagrama de clases, que permita la persistencia de las clases en una base de datos relacional. Hay que tener en cuenta que los jugadores estarán almacenados en LDAP.

Para realizar el diagrama modelo-relacional contamos con una serie de tablas que parten del diagrama de clases. Y éstas son las siguientes:

- ★ Tabla Jugador: Se encuentra almacenada en el servidor LDAP.
- ★ Tabla Juego: Contiene los datos de configuración del juego.
- ★ Tabla Palabra: Conforma el conjunto de palabras que se utilizarán en cada juego.

Cabe destacar, en primera instancia, que el diseño de las relaciones del Modelo ER pueden experimentar cambios, según las necesidades que vayamos experimentando durante el proyecto. La idea principal es clara, hay 3 entidades necesarias que serán las tablas con sus respectivos atributos y que se relacionarán entre sí. Sin duda, los identificadores entendidos como Claves Primarias (PK), se guardarán con un número entero y autoincremental para ordenar cada partida, jugador y palabras existentes en la Base de Datos.

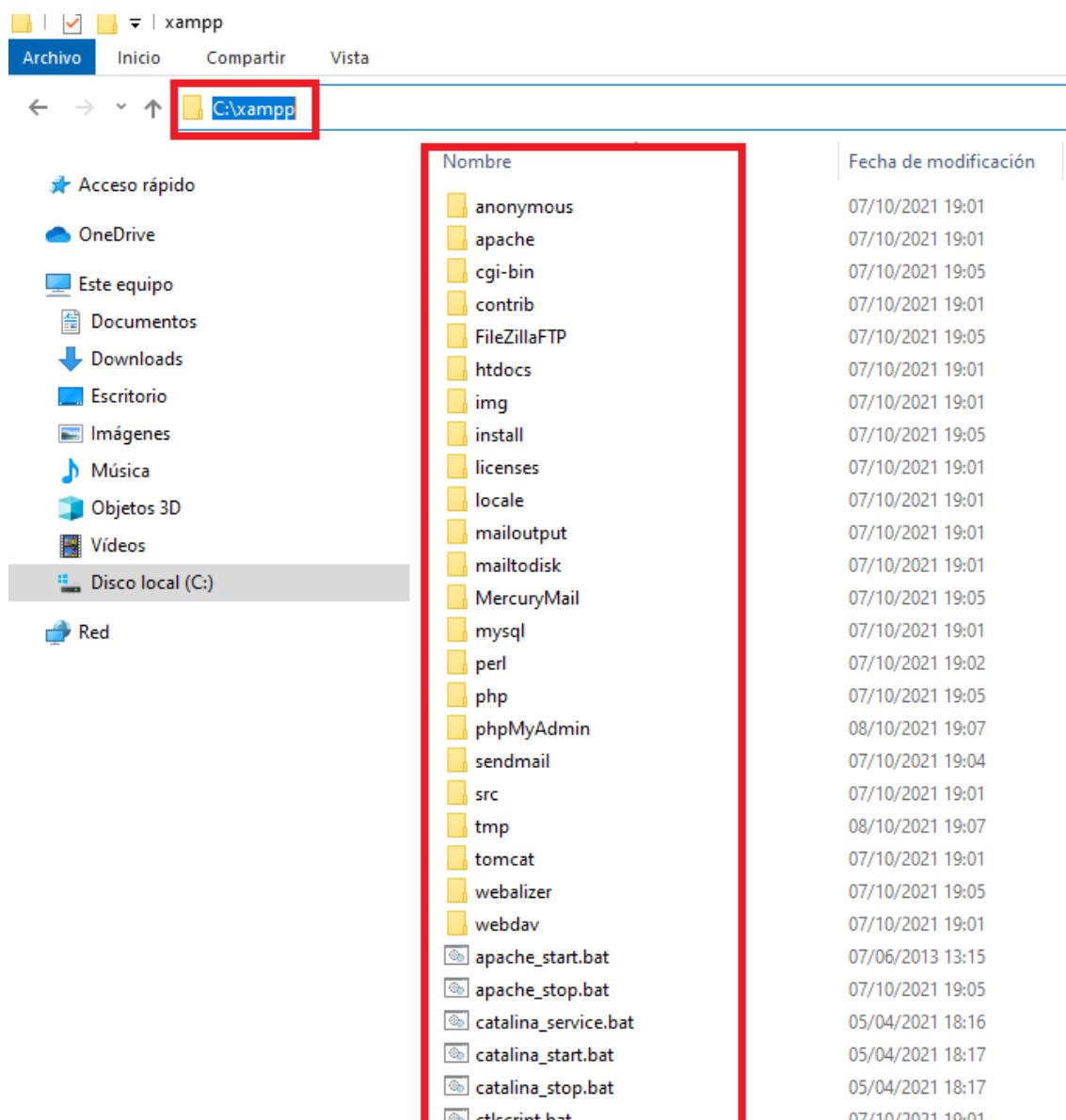
No obstante, hay otro elemento que consideramos que podría ser interesante de cara a la construcción de la Base de Datos, que es el uso de Claves Foráneas (FK). El motivo que nos impulsa a pensar en las llaves foráneas es que las claves primarias estén conectadas. ¿Para qué serviría? Pues la idea sería guardar un registro en el que se pueda observar de manera ordenada las partidas jugadas por un jugador y las palabras que le han aparecido en una partida determinada: pero tal y como explicamos anteriormente, si bien esto es una idea que pretendemos desarrollar, el proceso de programación y las necesidades del grupo podrían alterar esta estructura.



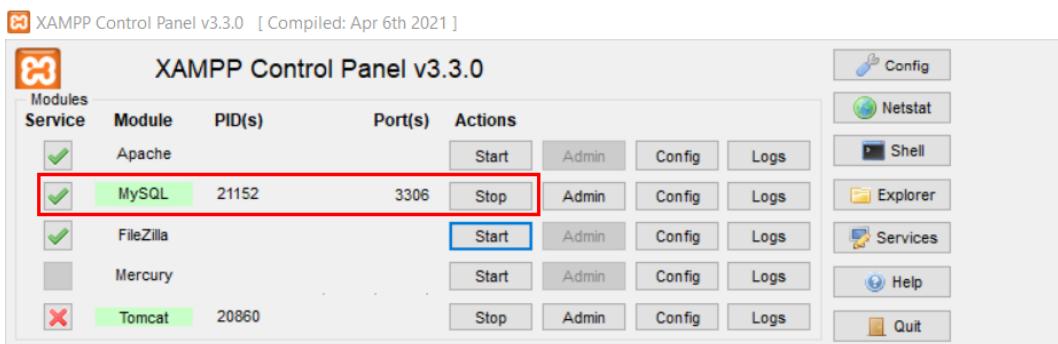
3. Instalar la Base de datos MariaDB o MySQL en caso de no tenerla instalada.

Comenzamos con la descarga e instalación de XAMPP (server independiente de plataforma que consiste principalmente en la base de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script: PHP y Perl).

<https://www.apachefriends.org/es/index.html>

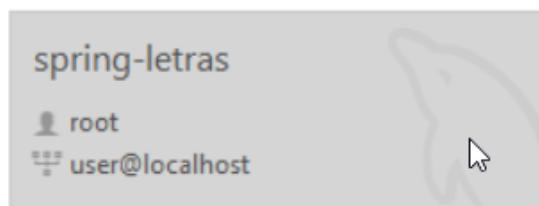


En este punto, comprobamos que XAMPP conecta con MySQL



Creamos la conexión a nuestra base de datos de MySQL, que en nuestro caso lo hemos situado en el fichero *Resources>Application.properties*.

```
# DATABASE
spring.datasource.url=jdbc:mysql://localhost/spring-letras?useSSL=false
spring.datasource dbname=spring-letras
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

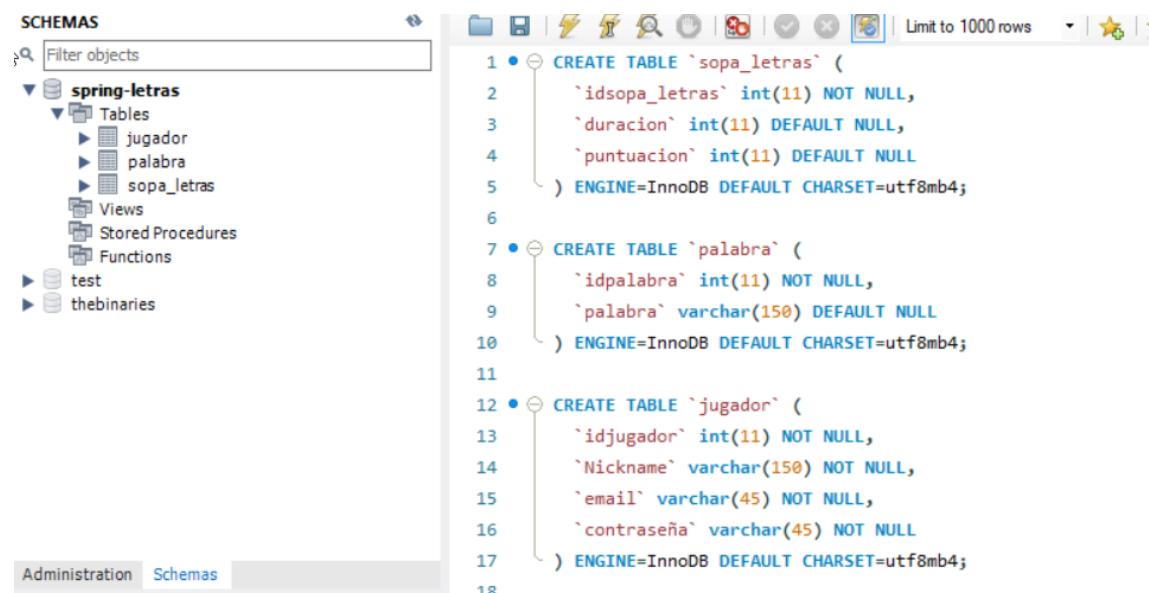


Y a continuación desde el SGBD, en nuestro caso con Workbench, creamos las tablas con sus respectivos atributos en la base de datos.



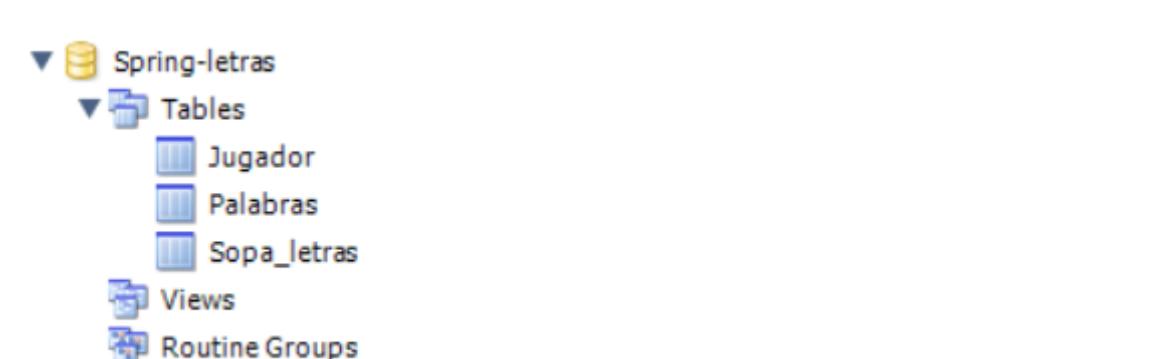
4. Crear la base de datos que se ha diseñado en el diagrama modelo-relacional en la Base de datos.

Creación de las tablas.



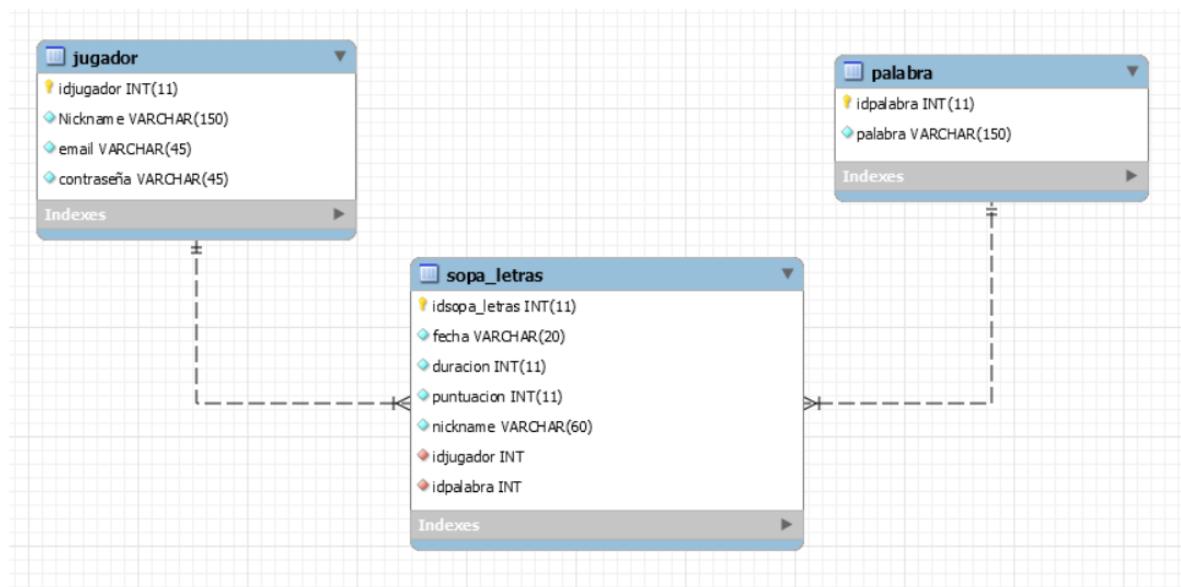
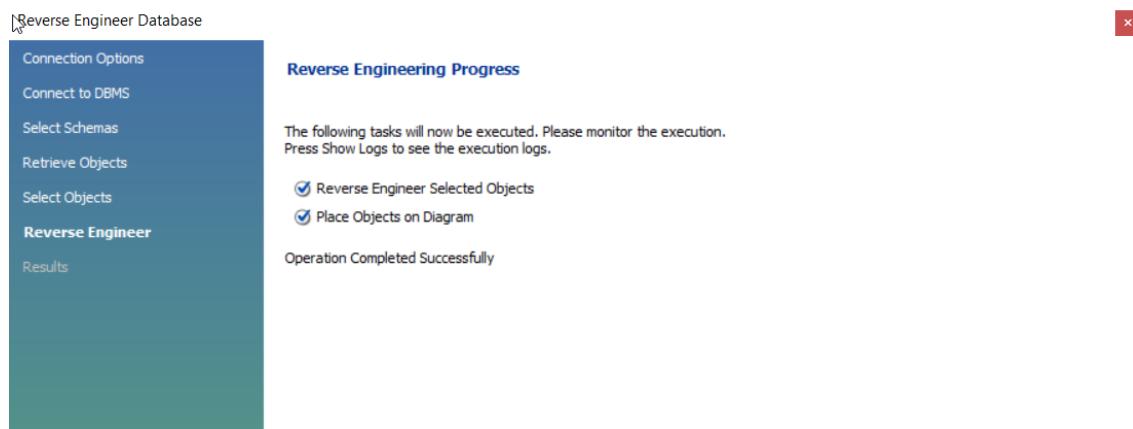
The screenshot shows the MySQL Workbench interface with the 'Schemas' tab selected. On the left, the 'spring-leturas' schema is expanded, showing its structure. On the right, the SQL editor displays the CREATE statements for three tables:

```
1 • CREATE TABLE `sopa_letras` (
2     `idsopa_letras` int(11) NOT NULL,
3     `duracion` int(11) DEFAULT NULL,
4     `puntuacion` int(11) DEFAULT NULL
5 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
6
7 • CREATE TABLE `palabra` (
8     `idpalabra` int(11) NOT NULL,
9     `palabra` varchar(150) DEFAULT NULL
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
11
12 • CREATE TABLE `jugador` (
13     `idjugador` int(11) NOT NULL,
14     `Nickname` varchar(150) NOT NULL,
15     `email` varchar(45) NOT NULL,
16     `contraseña` varchar(45) NOT NULL
17 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
18
```



The screenshot shows the MySQL Workbench interface with the 'Schemas' tab selected. On the left, the 'Spring-leturas' schema is expanded, showing its structure. It contains three tables: Jugador, Palabras, and Sopa_letras, along with Views and Routine Groups.

Y creamos el diagrama de nuestra base de datos.



5. Crear las clases y estructura la MVC que se utilizará para la realización del programa Sopa de letras teniendo en cuenta que el programa se debe realizar en JSP para las vistas y Servlets para los controladores.

Para levantar un servidor LDAP en spring hacemos uso de **Spring Security**, que es un framework que sirve para gestionar y controlar accesos, es decir, para verificar las autenticaciones. Con esto, mediante la configuración de éste con los datos de toda la configuración del LDAP, lo guardamos con un archivo llamado **Idap-data.ldif**, que se puede visualizar en nuestro github.

Mediante el uso de **WebSecurityConfigurerAdapter**, nos permite levantar el servidor ldap y comparar las contraseñas. En nuestro caso le mandamos nuestro login personalizado.

```

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable().authorizeRequests()
            .and().formLogin().loginPage("/login") // Manda a nuestra vista login y recibe de esta los datos.
            .and().permitAll();
    }
}

```

Este paso se puede realizar porque hemos agregado una dependencia a nuestro **pom.xml**.

```

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-ldap</artifactId>
</dependency>

```

Para que spring pueda hacer uso de las **vistas jsp**, se usa **application properties**, añadiendo el siguiente código e indicando la ruta de los jsp para su correcto funcionamiento.

```

#Views configuration
spring.mvc.view.prefix=/WEB-INF/vistas/

```

```
spring.mvc.view.suffix=.jsp
```

En el caso de querer recuperar los datos del usuario, agregamos a nuestro proyecto la siguiente dependencia:

```
<!--  
https://mvnrepository.com/artifact/org.springframework.security/spring-s  
ecurity-taglibs -->  
<dependency>  
    <groupId>org.springframework.security</groupId>  
    <artifactId>spring-security-taglibs</artifactId>  
    <version>5.5.2</version>  
</dependency>
```

Ésta nos permite hacer llamadas a Spring Security. Además, mediante esta dependencia podemos recuperar el nombre del usuario aunque se encuentre logueado mediante ldap, ahorrando mucho tiempo y haciendo más cómodo a los usuarios el uso de la aplicación.

Agregamos a nuestra vista jsp las siguientes líneas para hacer uso de la dependencia.

```
<%@ taglib prefix="security"  
uri="http://www.springframework.org/security/tags" %>
```

Para recuperar el nombre y se pueda visualizar en nuestro html hacemos uso del siguiente código.

```
<security:authorize access="isAuthenticated()>  
    <security:authentication property="principal.username" />  
</security:authorize>
```

Para la **persistencia de datos en la bbdd** agregamos las siguientes dependencias.

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>  
  
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>8.0.26</version>
```

```
</dependency>
```

Para que sea más fácil el código y la persistencia agregamos las siguientes dependencias a pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.22</version>
    <scope>provided</scope>
</dependency>
```

Con la librería de java **Lombok** nos permite ahorrar mucho tiempo en **getters y setters** mediante anotaciones para acceder a los modelos y bbdd.

The screenshot shows a Java code editor with the following code:

```
import lombok.Getter;
import lombok.Setter;
import javax.persistence.*;
import lombok.*;

@Entity
@Table(name = "users")
public class Users {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Getter
    @Setter
    @Column(name = "id")
    private int id;

    @Getter @Setter @Column(name = "nombre")
    private String nombre;
```

Annotations used in the code:

- `@Entity`
- `@Table(name = "users")`
- `@Id`
- `@GeneratedValue(strategy= GenerationType.IDENTITY)`
- `@Getter`
- `@Setter`
- `@Column(name = "id")`
- `@Getter @Setter @Column(name = "nombre")`

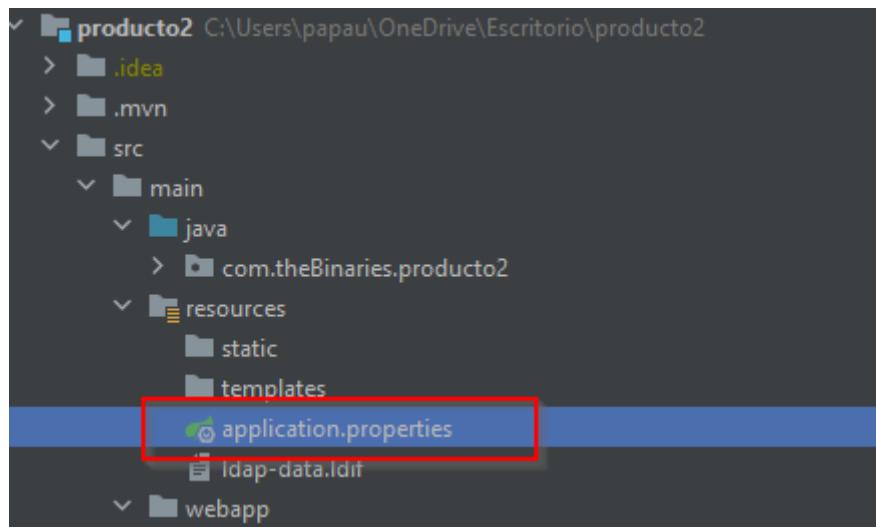
Annotations highlighted with red boxes:

- `@Getter` and `@Setter` in the `@Id` annotation
- `@Getter @Setter` in the `@Column(name = "nombre")` annotation

Annotations explained with callouts:

- `Importacion de la libreria` (Importation of the library) points to the `import lombok.*;` line.
- `Uso de anotaciones` (Use of annotations) points to the `@Getter` and `@Setter` annotations within the `@Id` annotation.

Configuración del archivo **properties** donde configuramos la **bbdd**.



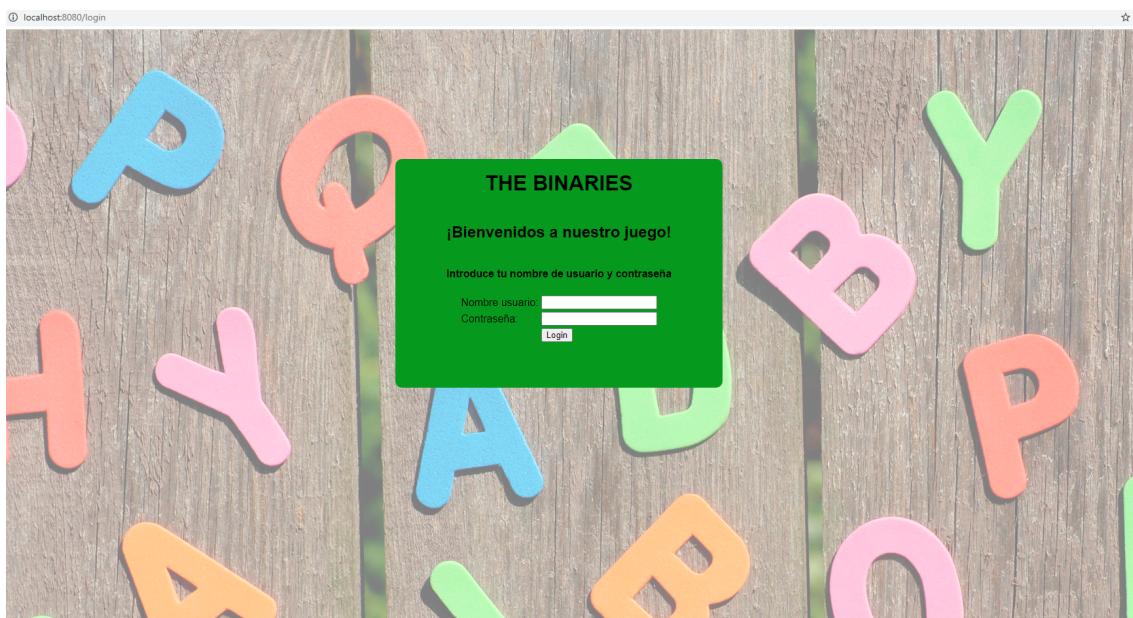
```
# DATABASE
spring.datasource.url=jdbc:mysql://localhost/spring-leturas?useSSL=false
spring.datasource dbname=spring-leturas
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

The configuration parameters highlighted with red boxes are:

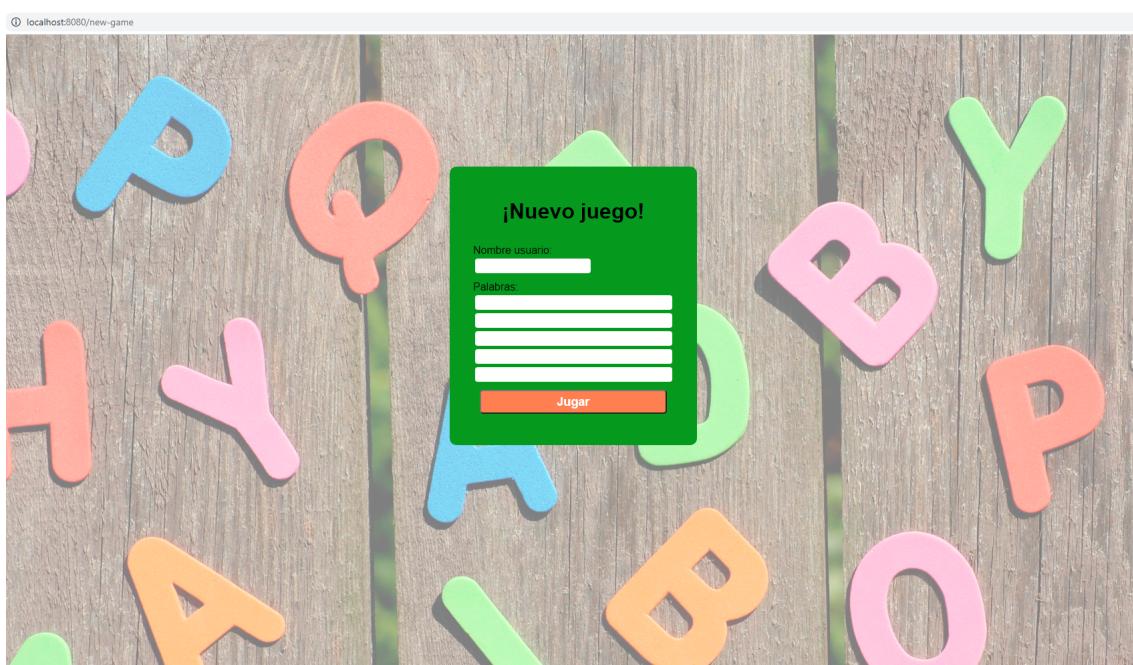
- spring.datasource.url (highlighted as "Url de la bbdd")
- spring.datasource.username (highlighted as "nombre de la bbdd")
- spring.datasource.password (highlighted as "contraseña")
- spring.datasource.driver-class-name (highlighted as "driver de mysql")

VISTAS DEL JUEGO

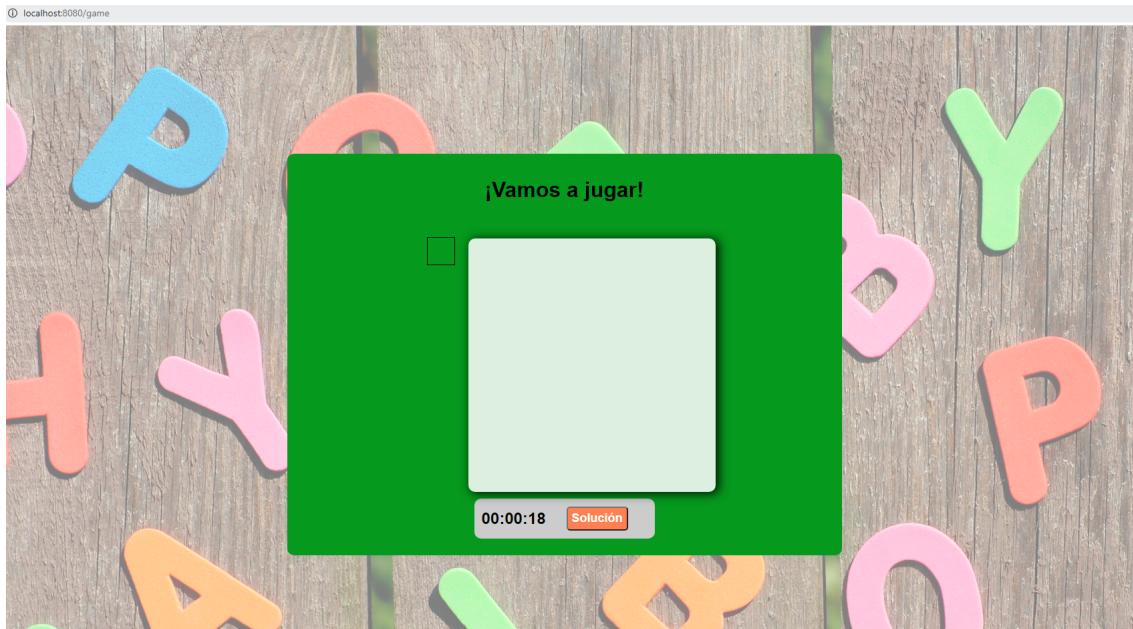
Login



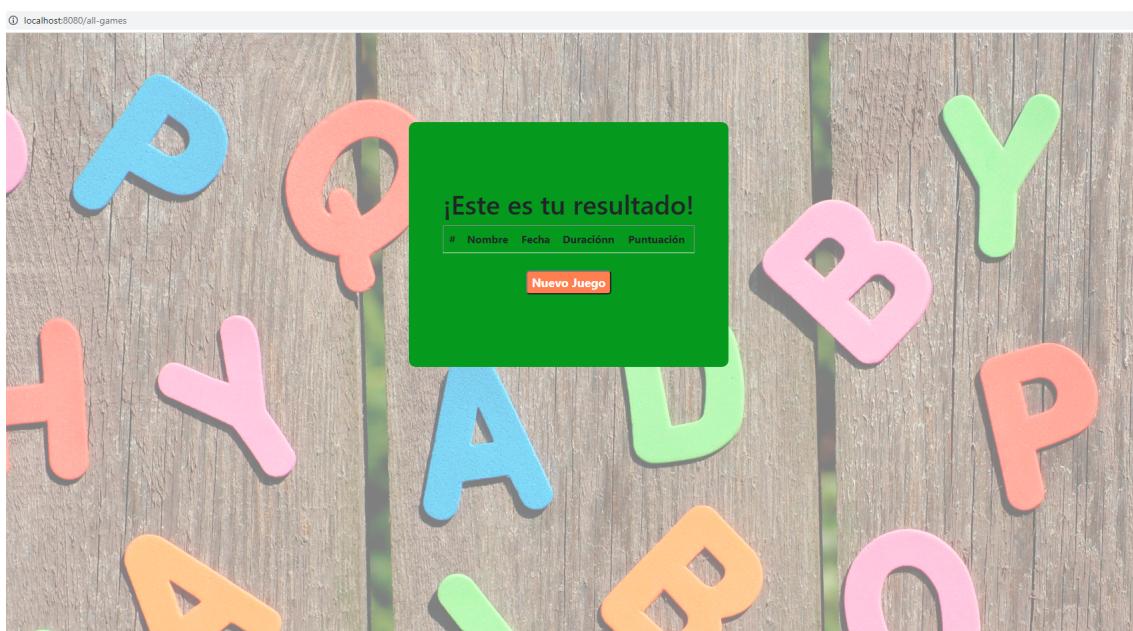
Nuevo juego



Vamos a jugar

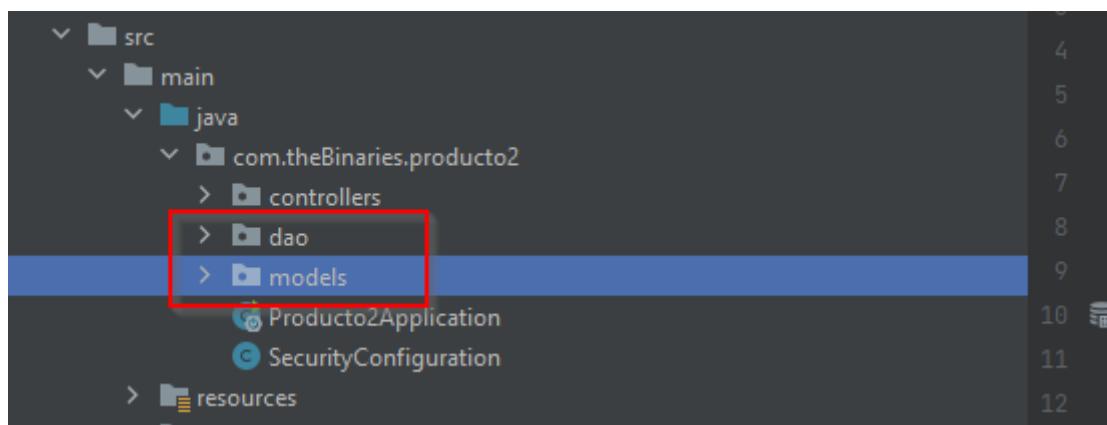


Resultado

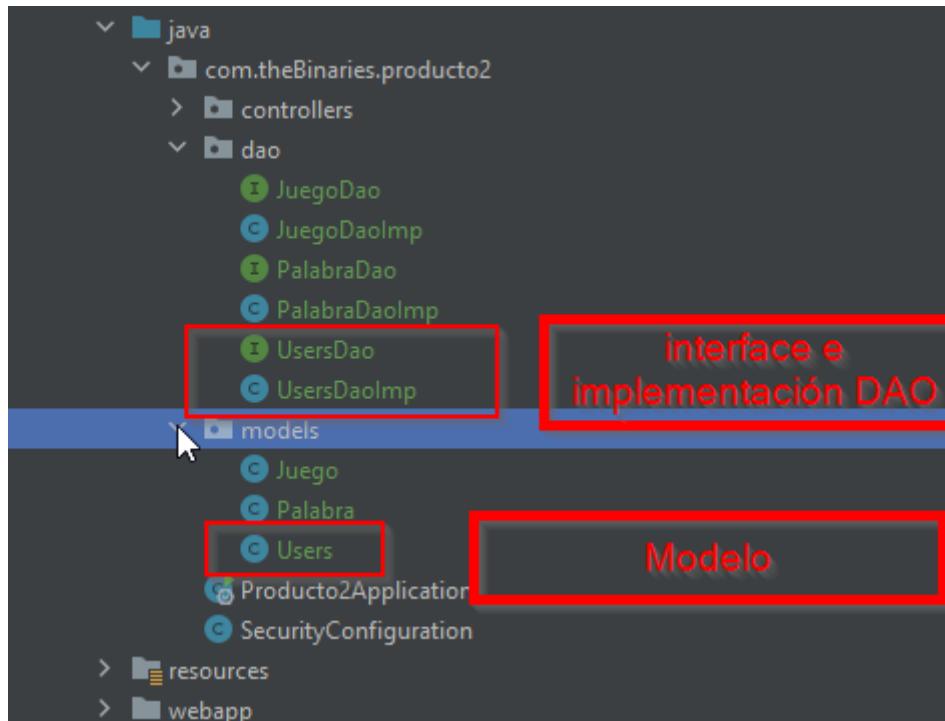


6. Crear la persistencia de los datos del programa utilizando el patrón de diseño DAO mediante Java y JDBC.

Una vez realizada toda la configuración tenemos todo lo necesario para crear nuestro MVC con modelos, DAO y la implementación. Para ello, creamos dos paquetes con los nombres DAO y Modelo.



Dentro de cada uno crearemos los diferentes tipos, tanto modelos como interfaces e implementaciones.



En este caso, solo vamos a explicar cómo hemos hecho el usuario porque el resto se hace del mismo modo.

```
package com.theBinaries.producto2.models;

import lombok.Getter;
import lombok.Setter; Libreria Lombok

import javax.persistence.*;
libreria de persistencia...

@Entity Anotacion que es una entidad
@Table(name = "users") Apunta a la bbdd a la tabla users
public class Users {

    @Id Clave primaria y se genera automaticamente
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Getter getter y setter creados por lombok
    @Setter
    @Column(name = "idUser") columna que apunta la id de la clase
    private int id;

    @Getter @Setter @Column(name = "nombre") Valor apunta a la columna nombre de la bbdd
    private String nombre;

    public Users( String nombre) {
        this.nombre = nombre;
    }
}
```

Para su implementación hacemos uso de interfaces **UserDao**.

```
package com.theBinaries.producto2.dao;

import com.theBinaries.producto2.models.Users;

import java.util.List;

public interface UsersDao {

    void registrar(Users usuario);
}
```

I

En este interfaces se deciden los métodos de la clase que implementa esta interface, en nuestro caso **usersDaoImplements**.

```
package com.theBinaries.producto2.dao;
import com.theBinaries.producto2.dao.UsersDao;
import com.theBinaries.producto2.models.Users;
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.Transactional;

@Repository
@Transactional
public class UsersDaoImp implements UsersDao{

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    @Transactional
    public void registrar(Users usuario) { entityManager.merge(usuario); }

}
```

En este caso, únicamente registramos los usuarios, ya que estos ya están registrados en Idap; y lo que haremos, será guardar su nombre en bbdd, la cual se relaciona con el juego para después hacer un tabla con la puntuación y los usuarios.