Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Carrera: Ingeniería en ciencias y sistemas
Catedrático: Floriza Ávila
Auxiliares: Benaventi Fuentes - Byron Hernández
Curso: Practicas Iniciales
Sección: C

Manual técnico

Practica #4

Programadores:

Christopher Miguel Angel Ramos Ascencio

Carne 202200057

Correo personal: kelltusad@gmail.com

Correo institucional: 3312427451801@ingenieria.usac.edu.gt

Joel Alexander Guzaro Tzunun

Carne 202201395

Correo personal: alexguz039@gmail.com

Correo institucional: 3013146910101@ingenieria.usac.edu.gt

Descripción del Proyecto

Como estudiantes de la carrera de Ciencias y Sistemas uno de los ámbitos donde más campo para aplicar que existe en la actualidad es el desarrollo de aplicaciones web y anteriormente los estudiantes ya han tenido un acercamiento con este tipo de aplicaciones de una manera mucho más general. Es necesario que los estudiantes se empiecen a familiarizar con herramientas que van a utilizar a lo largo de la carrera y que les puede servir en un ambiente laboral, estas herramientas son mayormente conocidas como Frameworks.

Para fomentar el uso de estas herramientas, se requiere que los grupos de estudiantes sean capaces de realizar una aplicación que se describirá más adelante, siempre enfocándose en la correcta distribución del trabajo y por la manera de trabajar, se requiere que todo este trabajo este almacenado en un repositorio para que los alumnos puedan acceder al código de una manera más eficiente.

Objetivos

- Relacionar al estudiante con Frameworks de desarrollo para aplicaciones web.
- Fomentar el uso de repositorios para administración de las aplicaciones y sus avances.
- Administrar información en un gestor de base de datos.

Manual técnico

En este manual podrás encontrar todos los aspectos técnicos que se abordaron para realizar la "Practica 4" donde abordaremos las clases y funciones, bases de datos. En general el apartado Backend de la práctica. Así como la funcional dad que realiza cada componente.

Backend en VisualStudio Code:

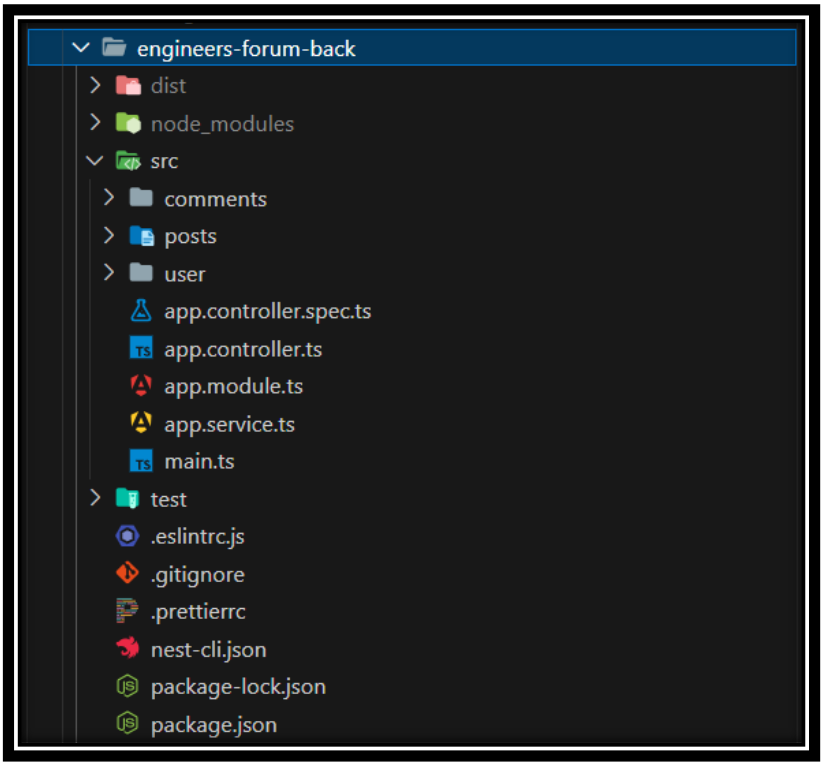
En VisualStudio contamos con una carpeta que lleva en su interior todo lo que se refiere al back con la implementación de Node para programar.

En la carpeta del backend encontramos los apartados de cada módulo diferenciado en la carpeta de “src”.

User: Es la carpeta que contiene las funcionalidades de uso y manejo de usuarios en la aplicación. Para iniciar sesión, registro etc.

Posts: Es la carpeta que contiene las funcionalidades para que los usuarios ingresados y Logeados puedan crear publicaciones y los demás usuarios también sean capaces de verlas.

Comments: Es la carpeta que contiene las funcionalidades del apartado de los comentarios en las publicaciones

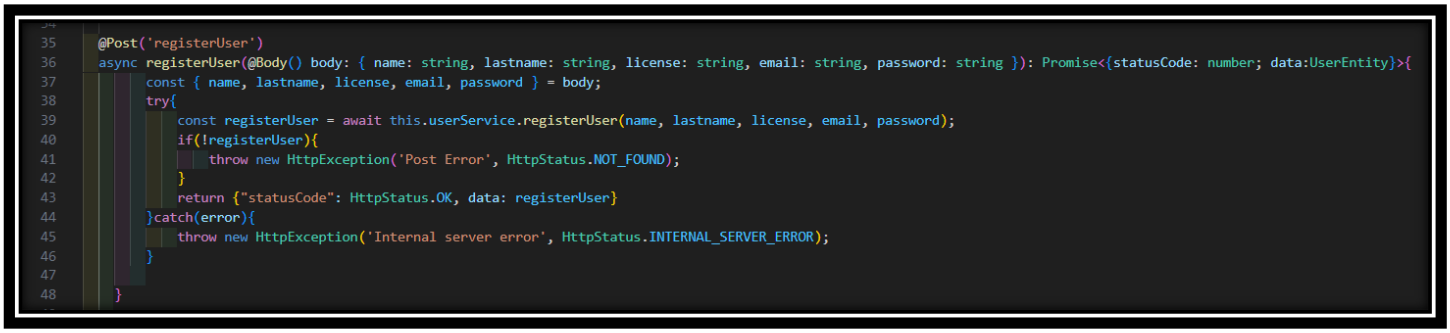


Funciones:

Cada carpeta dentro de src cuenta con sus funciones individuales para ser implementadas en la aplicación. Sus controladores, entity y servicios propios. A continuación se abordará en más detalle sobre las funcionalidades en el backend.

User

registerUser:
Es un post que solicita los campos que el usuario debe llenar para ser registrado



```
@Post('registerUser')
async registerUser(@Body() body: { name: string, lastname: string, license: string, email: string, password: string }): Promise<{statusCode: number; data:UserEntity}>{
    const { name, lastname, license, email, password } = body;
    try{
        const registerUser = await this.userService.registerUser(name, lastname, license, email, password);
        if(!registerUser){
            throw new HttpException('Post Error', HttpStatus.NOT_FOUND);
        }
        return {"statusCode": HttpStatus.OK, data: registerUser}
    }catch(error){
        throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
    }

}
```

getUsers:
Es un get que busca todos los usuarios en la base de datos

```
@Get('getUsers')
async findAll(): Promise<{ statusCode: number; data: UserEntity[] }> {
  try {
    const user = await this.userService.findAll();
    if (!user) {
      throw new HttpException('Users not found', HttpStatus.NOT_FOUND);
    }
    return { "statusCode": HttpStatus.OK, data: user };
  } catch (error) {
    throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

profile:
Es un post que publica el carnet del usuario logeado actualmente

```
@Post('profile')
async getUserById(@Body('license') license: string): Promise<{ statusCode: number; data: UserEntity[] }>{
  try {
    const user = await this.userService.findUserById(license);
    if (!user) {
      throw new HttpException('User not found', HttpStatus.NOT_FOUND);
    }
    return { "statusCode": HttpStatus.OK, data: [user] };
  } catch (error) {
    throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

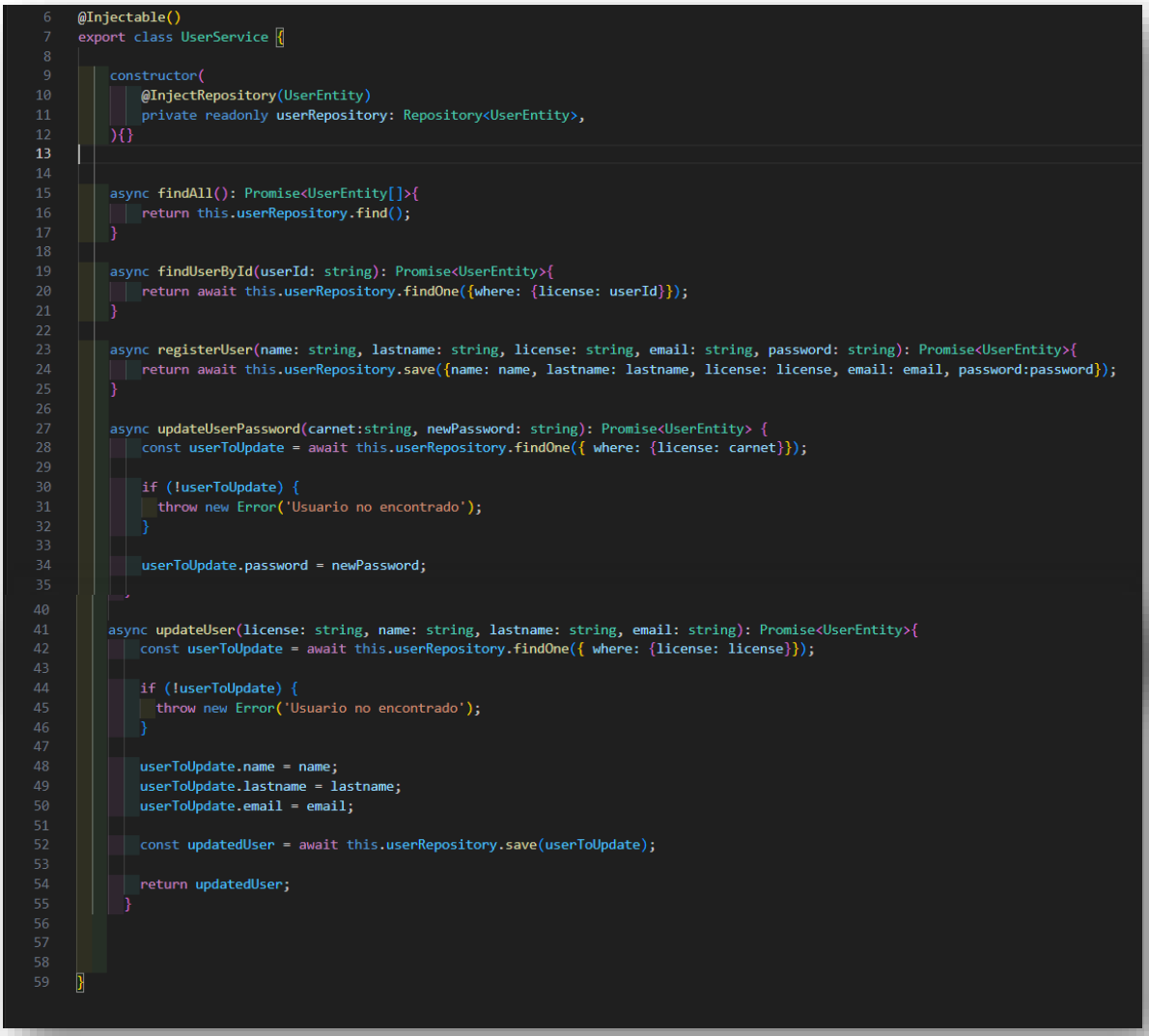
update- password:
Es un put que actualiza la contraseña del usuario

```
@Put('update-password')
async updateUserPassword(
  @Body('password') newPassword: string,
  @Body('carnet') carnet: string,
): Promise<{statusCode:number, data: UserEntity}> {
  try{
    const updateUser = await this.userService.updateUserPassword(carnet, newPassword);
    if(!updateUser){
        throw new HttpException('update Error', HttpStatus.NOT_FOUND);
    }
    return {"statusCode": HttpStatus.OK, data: updateUser}
  }catch(error){
    throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

update-user:
Es un put que modifica los datos principales del usuario (nombre, apellido, email)

```
@Put('update-user')
async updateUser(
  @Body('email') email: string,
  @Body('lastname') lastname: string,
  @Body('name') name: string,
  @Body('license') license: string,
): Promise<{statusCode:number, data: UserEntity}> {
  try{
    const updateUser = await this.userService.updateUser(license, name, lastname, email);
    if(!updateUser){
        throw new HttpException('update Error', HttpStatus.NOT_FOUND);
    }
    return {"statusCode": HttpStatus.OK, data: updateUser}
  }catch(error){
    throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

En los servicios de user encontramos las funciones
findAll: Busca todos los usuarios
findById: Busca el usuario por su número de Id
registerUser: función que registra a los usuarios
updateUserPassword: actualiza la contraseña del usuario actual



```
@Injectable()
export class UserService {

    constructor(
        @InjectRepository(UserEntity)
        private readonly userRepository: Repository<UserEntity>,
    ){}

    async findAll(): Promise<UserEntity[]>{
        return this.userRepository.find();
    }

    async findUserById(userId: string): Promise<UserEntity>{
        return await this.userRepository.findOne({where: {license: userId}});
    }

    async registerUser(name: string, lastname: string, license: string, email: string, password: string): Promise<UserEntity>{
        return await this.userRepository.save({name: name, lastname: lastname, license: license, email: email, password:password});
    }

    async updateUserPassword(carnet:string, newPassword: string): Promise<UserEntity> {
        const userToUpdate = await this.userRepository.findOne({ where: {license: carnet}});

        if (!userToUpdate) {
          throw new Error('Usuario no encontrado');
        }

        userToUpdate.password = newPassword;

    async updateUser(license: string, name: string, lastname: string, email: string): Promise<UserEntity>{
        const userToUpdate = await this.userRepository.findOne({ where: {license: license}});

        if (!userToUpdate) {
          throw new Error('Usuario no encontrado');
        }

        userToUpdate.name = name;
        userToUpdate.lastname = lastname;
        userToUpdate.email = email;

        const updatedUser = await this.userRepository.save(userToUpdate);

        return updatedUser;
      }


}
```

En la base de datos de SQL los usuarios constan con los siguientes atributos:

```
@Entity({name: 'Users'})
export class UserEntity {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    name: string;

    @Column()
    lastname: string;

    @Column()
    email: string;

    @Column()
    license: string;

    @Column()
    password: string;
}
```

posts

createPost:
Es un post que toma los datos del carnet, maestro, curso, fecha y un id para crear un post en la aplicación

```
@Post('createPost')
async createPost(@Body() body: {textPost: string, license_user: string, teacher: string, nameCourse: string, idCourse: string, datePost: string}): Promise<{statusCode: number, data: PostEntity}> {
    const { textPost, license_user, teacher, nameCourse, idCourse, datePost} = body;
    try{
        const post = await this.postservice.registerPost(textPost, license_user, teacher, nameCourse, idCourse, datePost);
        if(!post){
            throw new HttpException('Post Error', HttpStatus.NOT_FOUND);
        }
        return {"statusCode": HttpStatus.OK, data: post}
    }catch(error){
        throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

getPosts:
Es un get que busca todos los posts cargados en la base de datos

```
10      @Get('getPosts')
11      async findAll(): Promise<{statusCode: number, data: PostEntity[]}> {
12          try{
13              const posts = await this.postservice.findAll();
14              if(!posts){
15                  throw new HttpException('Post Error', HttpStatus.NOT_FOUND);
16              }
17              return {"statusCode": HttpStatus.OK, data: posts}
18          }catch(error){
19              throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR);
20          }
21      }
22
```

En los servicios de los posts encontramos las funciones:

findAll: busca todas las publicaciones
registerPost: crea un nuevo post

```
@Injectable()
export class PostsService {

    constructor(
        @InjectRepository(PostEntity)
        private readonly postRepository: Repository<PostEntity>,
    ){}

    async findAll(): Promise<PostEntity[]>{
        return this.postRepository.find();
    }

    async registerPost(textPost: string, license_user: string, teacher: string, nameCourse: string, idCourse: string, datePost: string): Promise<PostEntity>{
        return await this.postRepository.save({textPost: textPost, license_user: license_user, teacher: teacher, nameCourse: nameCourse, idCourse: idCourse, datePost: datePost});
    }


}
```

En la base de datos de SQL los posts cuentan con los siguientes atributos:

```
@Entity({name: 'Posts'})
export class PostEntity {
    @PrimaryGeneratedColumn()
    idPost: number;

    @Column()
    textPost: string;

    @Column()
    license_user: string;

    @Column()
    teacher: string;

    @Column()
    nameCourse: string;

    @Column()
    idCourse: string;

    @Column()
    datePost: string;
}
```

Comments

createComment:
Es un post que solicita los campos que el usuario debe llenar para hacer un comentario en una publicación existente

```
@Post('createComment')
async createPost(@Body() body: {textComment: string, idPost: number, license_user: string}): Promise<{statusCode: number, data: CommentsEntity}> {
    const { textComment, idPost, license_user} = body;
    try{
        const comment = await this.commentsservice.registerComment(textComment, idPost, license_user);
        if(!comment){
            throw new HttpException('Post Error', HttpStatus.NOT_FOUND);
        }
        return {"statusCode": HttpStatus.OK, "data": comment}
    }catch(error){
        throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR)
    }
}
```

getCommentsByPostId:
Es post que encuentra el Id de la publicación para realizar el comentario

```
@Post('getCommentsByPostId')
async getCommentsByPostId(@Body() body: { idPost: number }): Promise<{statusCode: number, data: CommentsEntity[]}> {
    const { idPost } = body;
    this.commentsservice.getCommentsByPostId(idPost);
    try{
        const comment = await this.commentsservice.getCommentsByPostId(idPost);
        if(!comment){
            throw new HttpException('Post Error', HttpStatus.NOT_FOUND);
        }
        return {"statusCode": HttpStatus.OK, "data": comment}
    }catch(error){
        throw new HttpException('Internal server error', HttpStatus.INTERNAL_SERVER_ERROR)
    }
}
```

En los servicios de los comments encontramos las funciones:

getCommentsByPostId: identifica el id de la publicación la cual se va a comentar
registerComment: crea un nuevo comentario

```
@Injectable()
export class CommentsService {



    constructor(
        @InjectRepository(CommentsEntity)
        private readonly commentRepository: Repository<CommentsEntity>,
    ){}

    async getCommentsByPostId(idPost: number): Promise<CommentsEntity[]> {
        return this.commentRepository.find({ where: { idPost } });
    }

    async registerComment(textComment: string, idPost: number, license_user: string): Promise<CommentsEntity>{
        return await this.commentRepository.save({textComment: textComment, idPost: idPost, license_user: license_user});
    }



}
```

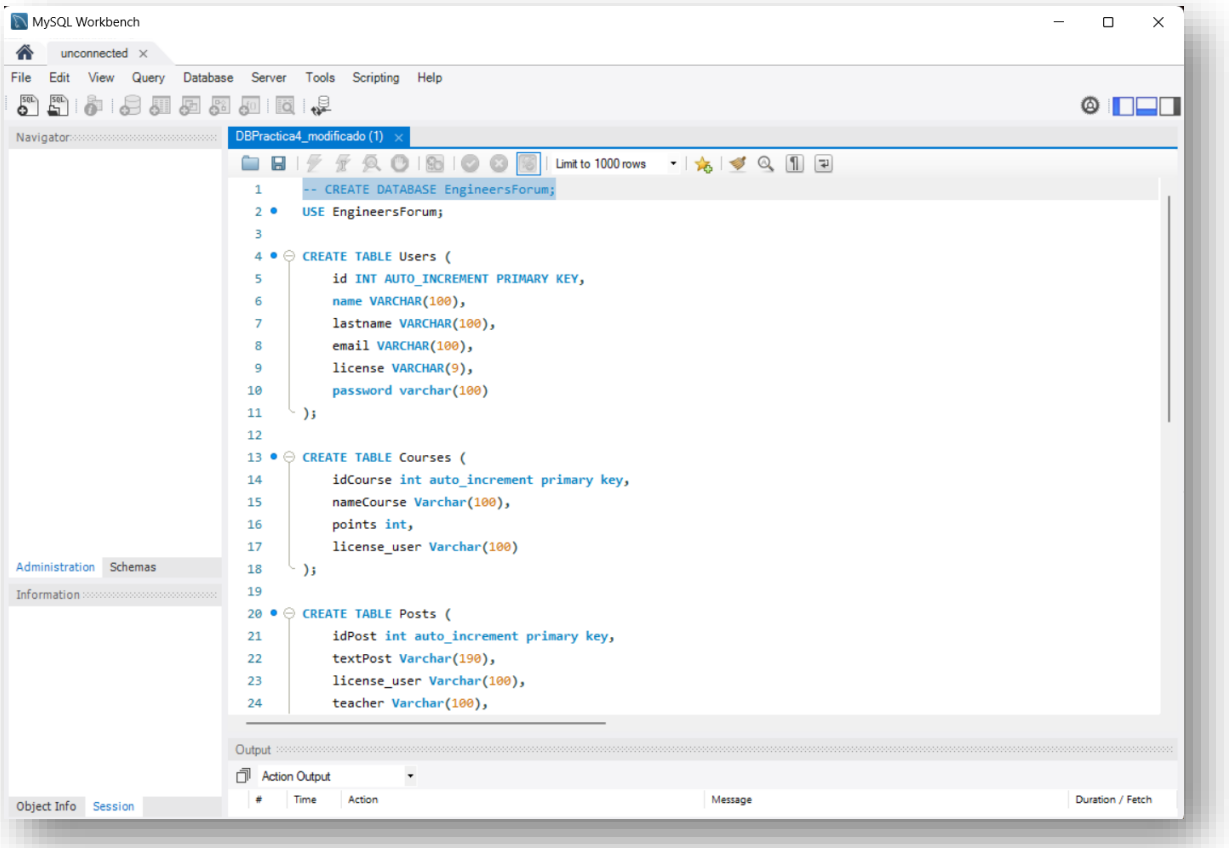
En la base de datos de SQL los comments cuentan con los siguientes atributos:

```
@Entity({name: 'Comments'})
export class CommentsEntity {
    @PrimaryGeneratedColumn()
    idComment: number;

    @Column()
    textComment: string;

    @Column()
    idPost: number;

    @Column()
    license_user: string;
}
```

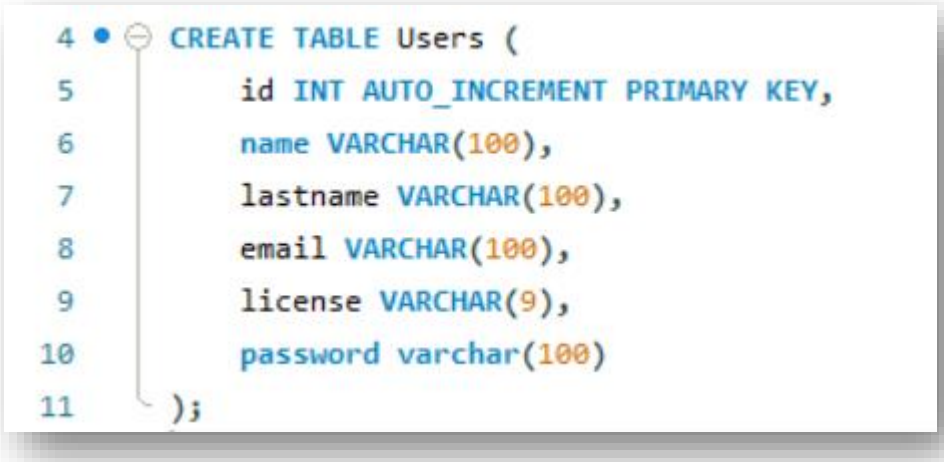
Base de datos

Para la practica4 utilizamos MySQL para crear la base de datos creando la tabla “EngineersForum”.

En la workbench de MySQL tenemos las tablas que almacenan la información de usuarios, cursos, comentarios etc



Base de datos Usuarios:



Base de datos posts:

```
20  CREATE TABLE Posts (
21      idPost int auto_increment primary key,
22      textPost Varchar(190),
23      license_user Varchar(100),
24      teacher Varchar(100),
25      nameCourse Varchar(100),
26      idCourse Varchar(100),
27      datePost Varchar(100)
28  );
```

Base de datos comments:

```
30  CREATE TABLE Comments (
31      idComment int auto_increment primary key,
32      textComment Varchar(190),
33      idPost int,
34      license_user Varchar(100)
35  );
```

Borrar la base de datos:

```
37  Select * from Comments;
38
39  Delimiter //
40  Create procedure InsertUser(name VARCHAR(100), lastname VARCHAR(100), email VARCHAR(100), license VARCHAR(9), password varchar(100))
41  BEGIN
42      Insert Into Users(name, lastname, email, license, password) values (name, lastname, email, license, password);
43  END //
44
45  Call InsertUser('Joel', 'Guzaro', 'alexguz039@gmail.com', '202201395', 'admin');
46  Call InsertUser('Christopher', 'Ramos', 'christopherramos@gmail.com', '202200057', 'admin');
47
48  Select * from Users;
49
50  ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'admin123';
51  flush privileges;
52
53  -- drop database EngineersForum
```