ANEXO II

SPORTIA
CLASES RECREATIVAS Y DEPORTIVAS

MÉTODOS, MODELOS Y PRUEBAS

G2-AJSW
Aplicación Java Sobre Web, Instituto de Ingeniería y Agronomía, UNAJ.

VERSION 1.0

PREPARED BY	FERNANDO PEREYRA	DATE	11/07/2022

TABLA DE CONTENIDO

1 MÉTODOS

A continuación, se describe a cada uno de los métodos del proyecto según servicio implementado.

Servicio: UserService

Públicos

CRUD

findUserById(ObjectId id) -> User : Busca al usuario de la base de datos que coincida con el parámetro *id* ingresado.

findUserByEmail(String email) -> User : Método encargado de buscar un usuario de la base de datos por medio del email ingresado por parámetro.

findUserByName(String name) -> User : Este método se encarga de buscar una lista de usuarios de la base de datos que coincidan con el nombre ingresado por parámetro.

findAllUsers() -> List<User>: Método encargado de retornar una lista completa con todos los usuarios de la base de datos.

deleteUserById(ObjectId id) -> void : Elimina al usuario del sistema que coincida con el Id ingresado por parámetro.

saveUser(User user) -> User : Método encargado de almacenar un nuevo usuario a la base de datos del sistema. Previo a su almacenamiento se realiza una serie de pasos:

1. Se aplica un hash a la password
2. Se setea la lista que contiene los roles que por defecto tendrá todo nuevo usuario
3. Se lo habilita para utilizar el sistema poniendo la propiedad *enabled* en *true*
4. Mediante la interfaz *UserRepository* se lo almacena en la base de datos

UTILITIES

isEmailExist(String email) -> boolean : Método encargado de determinar sin un email pertenece a un usuario almacenado en el sistema retornando *true* en caso de que así fuera o *false* en caso contrario.

registerNewUserAccount(DataUser dataUser, BindingResult bindingResult) -> User : Se encarga de la validación de los campos del formulario se rechaza manualmente los campos con errores al objeto BindingResult enlazado al objeto de análisis, en este caso, *DataUser*. El método retorna al usuario registrado exitosamente o *null* en caso de existir error en la validación de los datos ingresados por el usuario.

Autenticación y autorización

loadUserByUsername(String email) -> UserDetails : Implementación del método de la interfaz *UserDetailsService*, encargado de verificar y validar una correcta autenticación del usuario y determinar el nivel de autorización que este posee en el sistema. Retorna un objeto *UserDetails* perteneciente al *package org.springframework.security.core.userdetails*.

Privados

getUserAuthority(Set<UserRole> userRoles) -> List<GrantedAuthority> : Con este método A partir de una lista de tipo Set de Roles del usuario, establecidos con un *enum*, se obtiene la lista de objetos de tipo *GrantedAuthority* (interfaz perteneciente al paquete de Spring Security) necesaria para realizar el proceso de autenticación del usuario.

buildUserForAuthentication(User user, List<GrantedAuthority> authorities) -> UserDetails : Este método se encarga de verificar coincidencia entre los datos ingresados por el usuario y la fuente de datos correspondiente. A partir del @param *user* se válida asociación lícita entre email y password para autenticación y con el @param *authorities* se determina el nivel de acceso del usuario a los recursos del sistema para autorización.

Servicio: LessonService

Públicos

CRUD

FindLessonById(String id) : Busca a la representación del objeto Lesson (una clase brindada por la empresa) en la base de datos a partir del *id* ingresado por parámetro.

findLessonsByListId(Set<ObjectId> lessonsId) -> List<Lesson> : Método encargado de buscar una lista de *Lesson* en la base de datos que contenga alguno de los identificadores definidos en la lista de tipo *Set* de *ObjectId* pasada por parámetro.

findAll() -> List<Lesson> : Retorna una lista con todas las clases de tipo *Lesson* almacenadas en la base de datos.

findLessonsByType(String type) -> List<Lesson> : Busca en la base de datos una lista de Lesson que sean del tipo ingresado por parámetro.

findLessonByTypeAndStartTimeBetween(String type, String startTime, String endTime) -> List<Lessons> : Busca a las clases de tipo Lesson que coincidan con el parámetro *type* ingresado, y su horario de inicio esté situado dentro del rango de tiempo establecido entre los parámetros *startTime* y *endTime*.

saveLesson(Lesson lesson) -> void : Método encargado de almacenar una Lesson a la base de datos mediante @Bean definido con la interfaz *lessonRepository*.

filter(String type, String range_time, String zone) -> List<Lesson> : Método encargado de **f**iltrar la búsqueda a la base de datos de las clases tipo Lesson por tipo, rango de tiempo y zona.

UTILITIES

getTimeRange(int startTime, int endTime) -> List<String> : Retorna una lista de tipo *String* que define el rango de tiempo permitido para filtrar las clases del sistema. Esta lista se mostrará al usuario en la interfaz correspondiente.

Servicio: MPService

Este servicio es utilizado para dotar al sistema de un servicio de sandbox de pago de Mercado Pago.

Se utilizó la Api Checkout Pro implementándose el método descripto a continuación.

createPreference(PreferenceDTO preferenceDTO) -> ResponseEntity : Método encargado de crear la preferencia de la API de Mercado Pago (Checkout Pro) a partir de la preferencia definida específicamente por el sistema para la venta de sus servicios (*PreferenceDTO*) y según la elección de estos para la compra por parte del usuario. En la preferencia creada por Checkout Pro se encuentra el link por medio del cual se redirigirá al usuario hacia Mercado Pago para que continue allí el proceso de compra.

Servicio: EmailSenderService

Servicio utilizado para el envío de correos electrónicos. Para que esto sea posible se agregó la dependencia:

```
<dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
    <version>1.4.7</version>
</dependency>
```

sendEmail(DataMail dataMail, List<File> attachedFiles) -> ResponseEntity : Método encargado de enviar correo electrónico en la aplicación.

Primero se inicializa la sesión SMTP a partir de las propiedades correspondientes: *smpt.host, smtp.port, smtp.sender, smpt.user, smtp.auth,* etc.

Se crea un nuevo mensaje para la sesión iniciada previamente mediante la clase *MimeMessage*.

Se indican en este el emisor, el remitente, el asunto y el cuerpo del correo.

Para construir el cuerpo del mensaje se utiliza la clase *MimeBodyPart* que permite agregarle al *body* las diferentes partes que lo componen, a saber: texto, contenido HTML, archivos adjuntos, etc.

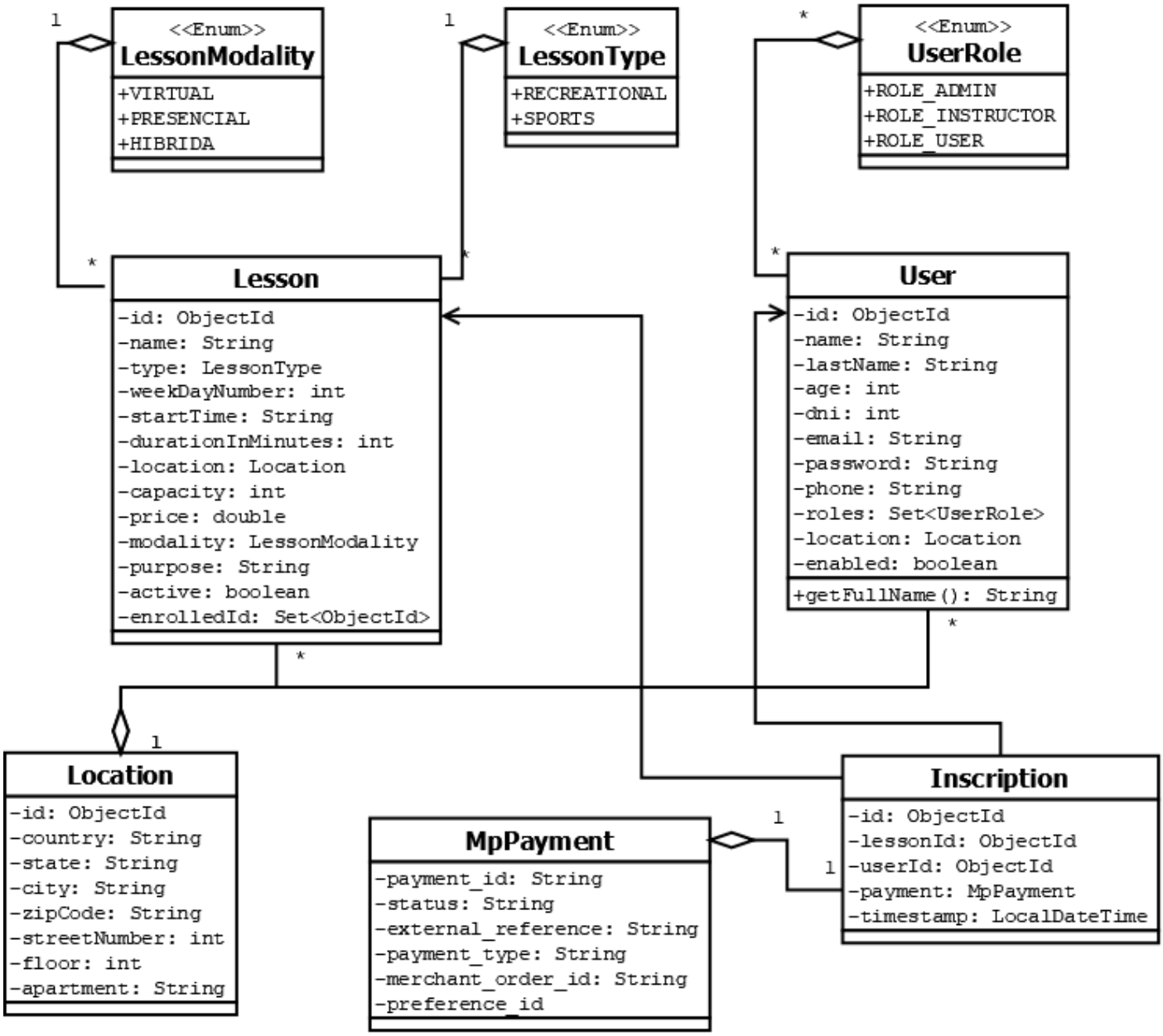
Se realiza la conexión por medio del protocolo de transporte SMTP indicando las credenciales, usuario y contraseña de la cuenta SMTP utilizada para manda el correo, se guardan los datos establecidos en el objeto *MimeMessage* y se envía el mensaje.

Por último, se cierra la conexión de trasnporte SMTP establecida.

2 MODELOS

Definición del modelo de datos utilizado en la aplicación dentro de la estructura MVC.

Diagrama de clases:



3 PRUEBAS

Se indican algunas de las pruebas implementadas en la aplicación y las herramientas utilizadas en cada caso.

findLessonsByRangeTimeTest

Se realizó un test de la funcionalidad de búsqueda de las clases por un determinado rango de tiempo indicado por parámetro del servicio de las clases (*LessonService*).

Se utiliza la dependencia de Mokito para la creación de entornos simulados de prueba. En este caso, se simulo la funcionalidad del repositorio de las clases (*LessonRepository*) indicándole la fuente de datos en la cual basarse para obtener los resultados de los test al aplicarle las funciones puestas a prueba del servicio indicado.

```
@Test
void findLessonsByRangeTimeTest(){
    List<Lesson> lessons = Arrays.asList(
            Lesson.builder().name("Billar").type(LessonType.RECREATIONAL).startTime("14:30").build(),
            Lesson.builder().name("Fútbol").type(LessonType.SPORTS).startTime("19:00").build(),
            Lesson.builder().name("Poker").type(LessonType.RECREATIONAL).startTime("22:45").build()
    );

    when(lessonRepository.findAll()).thenReturn(lessons);
    when(lessonRepository.findLessonByTypeAndStartTimeBetween( type: "recreational",
            startTime: "12:00",  endTime: "16:00")).thenReturn(Arrays.asList(lessons.get(1)));

    assertEquals( expected: 3, lessonService.findAll().size());
    verify(lessonRepository, times( wantedNumberOfInvocations: 1)).findAll();

    assertEquals( expected: "Billar", lessonService.findLessonByTypeAndStartTimeBetween(
            type: "recreational",  startTime: "12:00",  endTime: "16:00"));
    verify(lessonRepository, times( wantedNumberOfInvocations: 1)).findLessonByTypeAndStartTimeBetween(
            type: "recreational",  startTime: "12:00",  endTime: "16:00");
}
```

saveNewUser_WithCorrectInput_thenSuccess

Se pone a prueba a través de este método el correcto funcionamiento del endpoint */signup* en método POST, encargado del registro del usuario, en una entrada correcta de datos para posterior almacenamiento exitoso del nuevo usuario del sistema.

```
@Test
void saveNewUser_WithCorrectInput_thenSuccess() throws Exception {
    this.mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/signup")
            .accept(MediaType.TEXT_HTML)
            .param( name: "name",  ...values: "Juan")
            .param( name: "lastName",  ...values: "Peréz")
            .param( name: "email",  ...values: "jp@email.com")
            .param( name: "password",  ...values: "pass")
            .param( name: "matchingPassword",  ...values: "pass"))
            .andExpect(view().name( expectedViewName: "login"))
            .andExpect(status().isOk())
            .andDo(print());
}
```