11/03

Primer problema: Correr la aplicación

En estas instancias la información que poseo es escasa, inclusive consultando videos sobre el tema, por lo cual me dispuse a realizar un curso de Spring Boot en el cual se muestra cómo utilizar annotations para crear API Rest y controladores junto con servicios.

11/06

Primer Logro: Corre el inicio del programa

Luego de investigar y ponerme al tanto con el curso que estoy realizando, logré configurar correctamente un proyecto Java con Maven utilizando Spring Boot, con las dependencias correspondientes para poder utilizar las annotations que requiera.

Procedimiento

Cree los packages correspondientes para mantener la estructura del proyecto ordenada, dentro del package "controllers" creé el controlador de "inicio" (InicioController.java), el cual solo es una clase que mapea mi JSP de la pagina de inicio a través de la annotation @GetMapping("/"). Esto vendría a asemejarse a lo que en la versión anterior de mi proyecto era el Servlet de inicio "InicioServlet".

En el package "models" ubiqué hasta el momento solo las clases Usuario, Cliente y Prestador, que por ahora no se utilizan.

Decidí implementar además de Spring Boot, el framework conocido como Hibernate, para aprender a utilizar con las annotations correspondientes a JPA.

18/06

Segundo Problema: Implementar Hibernate

Procedimiento

Al agregar los packages de "repositories" y "services" junto con las clases e interfaces para poder proceder con el registro de algún tipo de usuario, aparecieron más conflictos a la hora de buildear el proyecto. Consulté el registro de errores del proyecto a ChatGPT y me sugirió cambiar la versión de dialecto de Hibernate:

"El error que estás experimentando indica que Hibernate no puede cargar la clase org.hibernate.dialect.MySQL5Dialect.

Esto se debe a que parece que la clase no se encuentra en el classpath.

Hay varias cosas que debes verificar y corregir:

1. Dialecto de Hibernate

A partir de Hibernate 5.3, los dialectos de MySQL fueron renombrados.

Si estás utilizando una versión más reciente de Hibernate, deberías

usar org.hibernate.dialect.MySQL8Dialect en lugar de org.hibernate.dialect.MySQL5Dialect.

2. Configuración del Dialecto

Asegúrate de que el dialecto está correctamente configurado en tu application.properties o application.yml.

3. Dependencia de Hibernate

Verifica que tienes la dependencia correcta de Hibernate en tu pom.xml."

- Corrección:

En application.properties cambie org.hibernate.dialect.MySQL5Dialect por org.hibernate.dialect.MySQL8Dialect, verifiqué además las dependencias correspondientes a Hibernate, y estaban correctamente colocadas dentro del archivo Pom.

1/07

Tercer problema: Clases abstracta Usuario y sus tipos Cliente y Prestador (Subclases) ---> Error de mapeo de herencias

Con el agregado de las clases : Usuario y Cliente (Usuario como superclase abstracta y Cliente como subclase) tuve un inconveniente en la configuración para mapear a Cliente como subclase, visto que ambas tienen una relación de este tipo mediante clave foránea o FK en las distintas tablas de la base de datos.

Al intentar correr la aplicación, el stack trace me confirmaba que no estaba utilizando bien las etiquetas o annotations dentro de la clase Cliente, indagando, pude localizar la falla en el código para poder mapear correctamente la relación de herencia que tenían esta clases. A todo esto, tuve que cambiar la configuración de Hibernate, ya que no solo verificaba las tablas de mi base de datos, sino que ademas las modificaba y agregaba campos extra, esto lo hice dentro de applicationProperties cambiando de:

spring.jpa.hibernate.ddl-auto=update

a

spring.jpa.hibernate.ddl-auto=validate.

Procedimiento:

- Consulté dos videos sobre mapeo de relaciones con JPA(Java Persistence Api) a través de annotations:

links:

https://www.youtube.com/watch?v=hNG8Xq5ypuM&t=1098s

https:/www.youtube.com/watch?v=g3z9lZukLEI

A la par también consulté con GPT aunque se prestaba mas para confusiones que otra cosa. Gracias a los videos pude ilustrarme sobre como hacer el mapeo de herencia correctamente, para poder buildear y poder correr la aplicación como se debía, cosa que al final pude lograr, y al hacerlo descubrí que tenía que corregir varias cosas, como el uso de @PrimaryKeyJoinedColumn, @Inheritance y los tipos de estrategia(yo utilicé JOINED debido a que utilizo dos tablas para diferenciar el tipo de usuario), el agregado de la clase Administrador fue clave, ya que no podía instanciar Usuarios si no existía un administrador, el uso de las annotations @OneToMany y @ManyToOne para las relaciones de administrador con usuarios, @Autowired @Service @Repository @Entity @Column junto con sus propiedades.

6/07

Segundo logro: Registro de Usuarios (Clientes y Prestadores)

Hasta este momento, no podía instanciar un Cliente o Prestador a través del uso del formulario JSP, y esto se debía a que, en el controlador de registro instanciaba Cliente y Prestador sin antes instanciar un Administrador o buscando el administrador en la base de datos, así que procedí a instanciar uno para probar a ver si el conflicto desaparecía, y así ocurrió. Una vez que pude por fin instanciar un usuario de esta forma, hice algunos cambios:

Creé un package "configurations" y allí una clase de configuración inicial llamada "DataInitializer" en la cual a través de las annotations @Configuration y @PostConstruct junto con el método "init()" verifico si en la base de datos existe algún administrador, y en caso de que no, se instancia uno y se guarda en la BDD. De esta forma para instanciar algún Usuario, buscamos solo al admin en la BDD y utilizamos el atributo necesario (según la clase que lo requiera).

PD: Todo esto lo hice con tutoría profesional, ya que no tenía idea de cómo encarar esta problemática por mi cuenta.

08/07

Tercer Logro: Inicio y Cierre de sesión

A través de un único controlador, se logró iniciar sesión a partir de un usuario creado y luego cerrar la sesión y volver a la página de inicio.

18/07

Cuarto Logro: Se crea y se almacena una sucursal

Se agregaron JSP que incluyen el formulario para controlar Sucursales y el form para controlar Salas. Se refactoreó el código perteneciente a la clase LoginController, y se configuró para cargar información pertinente luego de iniciar sesión, además agregamos el controlador para Sucursales, en el cual se encuentran los algoritmos pertenecientes al CRUD de las mismas, hasta ahora pudimos lograr crear una 'sucursal' e ingresarla en la base de datos, no sin antes resolver bastantes cuestiones.

Luego de muchas pruebas intentando crear a través del formulario una sucursal, probe la búsqueda por ID de una entidad dentro de la base de datos, en particular la de un Prestador, haciendo unos cambios en el repositorio y servicios de Prestador. El método que debería buscar el Prestador por ID, devuelve Optional<Prestador> o sea, prácticamente me devuelve el prestador o null, la cuestión fue que Hibernate por defecto, buscaba el Prestador por ID pero se basaba en el ID de Usuario, ya que la herencia se entiende de esa manera por como están mapeadas las entidades, por lo cual tuve que modificar la solicitud SQL mediante la annotation @Query que Spring framework provee, esto permite realizar otra solicitud que nosotros queramos, allí la configuré para que busque al Prestador por su propia ID

"SELECT p FROM Prestador p WHERE p.idPrestador = :idPrestador"

De esta manera, realizando esos cambios y agregando excepciones a la hora de realizar el llamado a la función que realiza la búsqueda por ID pude recorrer mejor la BDD para buscar el dato que necesitaba para la creación de la Sucursal.

Otra complicación que se tuvo, fue que luego de crear una Sucursal, intente editarla y eliminarla, pero esto resulto trabajoso de resolver.

Analicé una y otra vez el CRUD dentro de SucursalController realizado pura y exclusivamente con @GetMapping y @PathVariable, refactoricé varias veces el controlador y obtuve errores 500, 404 y 405. Finalmente verifiqué el JSP que realizaba la consulta hacia este controlador y entendí que la Query String no se debía enviar como lo hacía cuando utilizaba servlets, sino que debía apuntar al mapeo del controlador y enviar directamente el número de id de la sucursal que necesitaba modificar.

22/07

Quinto Logro: implementación de Spring Security - Encriptado de passwords

Una vez ya terminado el controlador de sucursales, me dispuse a refactorear el controlador de inicio para poder actualizar la lista de recursos requerida por cada tipo de usuario, ya que una vez creada una sucursal hasta no volver a iniciar sesión, no volvía a actualizarse la lista en la página de inicio. Al mismo tiempo también me di cuenta que sin iniciar sesión podía acceder a ciertos controladores mapeados vía URL, por lo cual busque información sobre autenticación de usuarios para restringir el acceso a los mismos sin haber antes iniciado sesión con algún tipo de usuario, consulté con chatGPT y me sugirió utilizar Authentication de Spring Security, que recopila los datos del nombre de usuario, con ese

dato busco al usuario y seteo los atributos que necesite. Por otro lado, también busque videos informativos en los que se realizaban estos procedimientos:

-"https://www.youtube.com/watch?v=AImMo4QFiqE"

-"https://www.youtube.com/watch?v=bgKmkgVuek8"

y en ellos también descubrí que para implementar la autenticación debía crear una configuración de seguridad, y en ella el gestor de la autenticación en cuestión, allí se le presentan los detalles del usuario a autenticar (Por convención se utilizan el nombre de usuario, su password y su autoridad / rol) la clase mapeada como configuración la llamé SecurityConfig, donde hay métodos mapeados como beans que se encargan de realizar el trabajo de permitir o autorizar ciertos mapeos y acceso a determinadas rutas del proyecto anteriores a la autenticación del usuario, e incluso posteriores, ya que también se configuran las rutas a las cuales determinados tipos de usuarios tienen acceso o no según su rol. El rol/autoridad de cada usuario también se pudo configurar dentro de SecurityConfig y CustomUserDetailsService (Clase nueva que extiende de UserDetailsService que utilizo para configurar el objeto de tipo User necesario para realizar una autenticación como se debe) de esta manera, teniendo lo requerido ya se puede iniciar sesión con cualquiera de los usuarios registrados de manera segura, con encriptado y codificado de password. <-- Commit 26/07.

27/07

Sexto Logro: CRUD de Salas de ensayo completo

Se agregó el controlador para la clase Sala, en él está configurado el CRUD personalizado para la visualización de las salas de ensayo acorde a la sucursal que se seleccionó, y para usuarios cuyos roles son Prestador o Administrador, la creación, edición y eliminación de estas.

Lo laborioso en este caso fue ajustar los atributos de modelo y peticiones en los botones y links de los JSP correspondientes a las consultas mapeadas para el controlador mismo, ya que para algunos casos utilicé @PathVariable y en otros @RequestParam, a futuro me gustaría cambiar en todos los controladores el uso de @PathVariable por @RequestParam, ya que se adecuaba mejor a lo aprendido sobre QueryString durante las clases referidas al proyecto en la segunda parte de la cursada de esta asignatura.

30/07

Séptimo Logro: CRUD de Reservas completo

Se agregó el controlador para la clase Reserva, donde configuré el CRUD también personalizado para poder visualizar reservas realizadas por clientes mediante un jsp preparado para ver lo necesario de cada tipo de usuario. Estas reservas pueden ser creadas/modificadas/eliminadas por Clientes (solo las suyas), Prestadores (solo reservas hechas en alguna de sus sucursales), y el Administrador. Acá también hubo que ajustar los atributos de modelo de los jsp pero no las peticiones, ya que en los métodos con @GetMapping utilicé @RequestParam.

Mi mayor conflicto para lograr que el controller funcione en su totalidad, fue dentro del método mapeado con @PostMapping "deleteReserva", en el cual se obtenía por request el id de la reserva realizada y con ese dato realizaba la peticion de eliminación con el service creado para Reserva así mismo, dentro del service se usaba el repository con los metodos disponibles para crear, editar, eliminar u obtener todas las entidades, en este caso empecé utilizando "deleteById" pero después de varias pruebas Hibernate no realizaba la petición a la base de datos. Luego probé realizando otro método "deleteByIdReserva" tanto en el service como en el repository pero allí el log de Spring generaba un error que no pude terminar de entender, así que decidí averiguar cómo configurar el método del repository para cambiar/forzar la utilización de la consulta que necesito, antes lo había logrado solo con @Query, pero en esta ocasión no funcionó, por lo cual opté por buscar más información y encontré dos videos:

-"https://www.youtube.com/watch?v=RNmiRbWvFRc"

-"https://www.youtube.com/watch?v=Z6p0RkHPwmY"

Con la información que obtuve de los videos, decidí utilizar las annotations @Modifying que indica que la consulta es una operación de modificación (INSERT, UPDATE,DELETE), @Transactional marca a las acciones del método para que se ejecuten como una transacción, @Query que especifica la consulta JPQL(Java Persistence Query Languaje) para eliminar la reserva con el id que se proporciona y @Param dentro del argumento del método para vincular el parámetro de la @Query al método creado. De esta forma refactoricé esa porción del repository, y luego volvi a correr la aplicación y a loguearme como cliente, allí realicé una reserva correctamente, la pude editar luego, y finalmente pude eliminarla del registro, el log de hibernate mostró la ejecución de la query "DELETE" como se debía.

Adicionalmente, dentro de la base de datos edite las relaciones mediante FK entre las entidades o tablas, para que se actualicen o eliminen en cascada, de esta forma por ejemplo al eliminar una sucursal, se eliminaran respectivamente todas las reservas realizadas en la sucursal, y todas las salas de ensayo dentro de la sucursal, esto evita de cierta forma tener que realizar código extra e iteraciones para eliminar recursos dentro de otros recursos.

03/08

Octavo Logro: Listas de reservas específica para Administrador y para Prestador

Dentro del controlador de Reserva, cree dos métodos que utilizan @GetMapping para la visualización de las listas de reservas. En el caso de los prestadores verán las que se hayan realizado en alguna de las salas de sus sucursales, y en el caso del Administrador, verán las reservas totales realizadas en todas las sucursales.

06/08

Personalización de mensajes de error o de proceso exitoso completa.

Se logró mediante el manejo de excepciones utilizando “try – catch” dentro de los métodos utilizados en los controladores, y configurando la visualización en los JSP.

11/08

Noveno Logro: CRUD de Usuarios completo

Estaba buscando darle una vuelta más de rosca al proyecto, así que a diferencia de la versión anterior, en donde los usuarios solo podían visualizarse y eliminarse, decidí agregar la posibilidad de que cada usuario (salvo el Administrador, por ahora) puedan editar sus atributos (menos el ID), para eso cree el controlador UsuarioController, en donde se encuentran los métodos que se encargan de crear y eliminar (para los Administradores), y editar usuarios. Se refactoreó la navbar agregando un botón "Editar Cuenta" para poder acceder al controlador y realizar el Update para las entidades.

12/08

Décimo Logro: Validación de Reservas

Teniendo completo el CRUD de reservas hace ya unos días, decidí ultimar detalles como validaciones para el final, por si llegaban a ocurrir problemas o confusiones. Así que en cuanto terminé con el grueso, me enfoqué en la lógica para realizar los 'throw exception' si se cumplían determinadas condiciones al tener acceso a todos los objetos que se relacionan, fue sencillo esta vez pensar bajo qué concepto una reserva puede existir, así que dicté los términos para mandar excepciones. Luego hice el llamado a esta función en los métodos @PostMapping correspondientes ("/edit","/create"), de esta manera, antes de realizar la reserva, la misma se valida y si todo esta bien, se realiza correctamente, sino, las excepciones se harán vigentes en el jsp de reservas.

26/08

Problema: vista de form de Login con sesión iniciada

Testeando la aplicación, nunca me percaté de un error que estuvo a la vista pero que pasé por alto, sucede cuando un usuario entra en sesión y a su vez, sigue teniendo acceso al formulario de login, cosa que no debería ocurrir a menos que se de por finalizada la sesión, por ello, me dispuse a averiguar sobre qué puedo hacer ante esta situación ya que en el pasado cuando vimos Servlets, esto lo solucionábamos utilizando filtros y utilizando el método "doFilter()" y en esta nueva ocasión utilizamos Controllers que generamos a través del uso de frameworks (en este caso Spring e Hibernate), hasta el momento, encontré información sobre una interfaz "HandlerInterceptor" que contiene Spring, que luego veré cómo funciona.

30/08

Actualización: Información ---> "https://refactorizando.com/handlerinterceptor-vs-filter-en-spring-2/"

La interfaz HandlerInterceptor Opera a nivel del framework Spring MVC y se centra en interceptar y modificar el procesamiento de las solicitudes entrantes antes y después de que se ejecute el controlador y se renderice la vista. En esta ocasión pude implementarla perfectamente para evitar el acceso a el formulario de login y de registro del proyecto una vez iniciada la sesión con un usuario mediante la creación de la clase "LoginInterceptor", alli se sobreescribe el metodo "preHandle" el cual se ejecuta antes del interceptor, esto quiere decir que toma acción previa a interceptar la url que debe tomar el interceptor como tal, en este caso, adquiere desde la sesión un atributo fundamental, el usuario (solo existe una vez que se inicia la sesión), y si el mismo esta presente, se lo redirige a la pagina de inicio de la aplicación.

Ademas, para efectivizar la ejecución del interceptor se le dió vida a una nueva clase llamada "WebMvcConfig" que implementa otra interfaz llamada "WebMvcConfigurer" propia de Spring Framework, para asi sobreescribir su método "addInterceptors()" y configurarlo para que posterior al inicio de la sesión intercepte a las URL pertenecientes al registro y el login de la aplicación ("/login", "/registrarse"), de esta manera evitamos posibles errores en la aplicación.

@Autowired

Descripción: Inyecta automáticamente las dependencias marcadas con esta anotación.

Uso: Se utiliza en campos, constructores o métodos para inyectar dependencias gestionadas por Spring.

@Controller

Descripción: Marca una clase como un controlador de Spring MVC.

Uso: Se utiliza en clases que manejan solicitudes web y devuelven vistas.

@ExceptionHandler

Descripción: Define un método que maneja excepciones específicas lanzadas por los métodos del controlador.

Uso: Se utiliza en métodos dentro de un controlador para manejar excepciones y proporcionar respuestas personalizadas.

@GetMapping

Descripción: Maneja solicitudes HTTP GET.

Uso: Se utiliza en métodos dentro de un controlador para mapear solicitudes GET a métodos específicos.

@PathVariable

Descripción: Vincula una variable de ruta a un parámetro de método.

Uso: Se utiliza en métodos de controlador para extraer valores de la URL.

@PostMapping

Descripción: Maneja solicitudes HTTP POST.

Uso: Se utiliza en métodos dentro de un controlador para mapear solicitudes POST a métodos específicos.

@RequestMapping

Descripción: Mapea solicitudes HTTP a métodos de controlador.

Uso: Se utiliza en clases y métodos de controlador para definir rutas y métodos HTTP.

@RequestParam

Descripción: Vincula un parámetro de solicitud a un parámetro de método.

Uso: Se utiliza en métodos de controlador para extraer parámetros de la solicitud.

@RestController

Descripción: Combina @Controller y @ResponseBody.

Uso: Se utiliza en clases de controlador para manejar solicitudes REST y devolver datos JSON o XML.

@Service

Descripción: Marca una clase como un servicio de Spring.

Uso: Se utiliza en clases que contienen lógica de negocio.

@WebMvcTest

Descripción: Configura un entorno de prueba para controladores Spring MVC.

Uso: Se utiliza en clases de prueba para probar controladores Spring MVC.

@Entity

Descripción: Marca una clase como una entidad JPA.

Uso: Se utiliza en clases que representan tablas en la base de datos.

@GeneratedValue

Descripción: Indica que el valor de un campo debe ser generado automáticamente.

Uso: Se utiliza en campos de entidades JPA para especificar la estrategia de generación de valores.

@Id

Descripción: Marca un campo como la clave primaria de una entidad JPA.

Uso: Se utiliza en campos de entidades JPA para definir la clave primaria.

@JoinColumn

Descripción: Especifica la columna de unión para una asociación de entidad.

Uso: Se utiliza en campos de entidades JPA para definir la columna de unión en relaciones.

@ManyToOne

Descripción: Define una relación de muchos a uno entre dos entidades.

Uso: Se utiliza en campos de entidades JPA para definir relaciones.

@OneToMany

Descripción: Define una relación de uno a muchos entre dos entidades.

Uso: Se utiliza en campos de entidades JPA para definir relaciones.

@Repository

Descripción: Marca una clase como un repositorio de Spring.

Uso: Se utiliza en clases que acceden a la base de datos.

@Service

Descripción: Marca una clase como un servicio de Spring.

Uso: Se utiliza en clases que contienen lógica de negocio.

@Transactional

Descripción: Gestiona transacciones en métodos o clases.

Uso: Se utiliza en métodos o clases para definir el comportamiento transaccional.

@Valid

Descripción: Activa la validación de un objeto.

Uso: Se utiliza en parámetros de método para validar automáticamente los objetos de entrada.

@Value

Descripción: Inyecta valores de propiedades en campos.

Uso: Se utiliza en campos para inyectar valores desde el archivo de propiedades.