Guía control check 2 LAB

Test 1 – Creación de le entidad Care

Modificar la clase “Care” alojada en el paquete “org.springframework.samples.petclinic.care” para que sea una entidad. Esta entidad debe tener los siguientes atributos y restricciones:

- Un atributo de tipo entero (Integer) llamado “id” que actúe como clave primaria en la tabla de la base de datos relacional asociada a la entidad.
- Un atributo de tipo cadena (String) llamado “name” que no puede ser vacío, de valor único (es decir, que no puede haber en la base de datos otro Care con el mismo nombre[1]), y cuya longitud mínima son 3 caracteres y la máxima 40.
- Un atributo de tipo entero (Integer) llamado “careDuration” que debe contener un valor entre 1 y 120. Este atributo representará la duración del tratamiento o cuidado aplicado a la mascota.
- Una relación N a N unidireccional con la clase PetType (tipo de mascota) que describa el tipo de mascota al que se puede aplicar ese tipo de cuidado. El atributo que representa la relación debe ser de tipo Set. Este conjunto no puede ser vacío, puesto que todo cuidado debe poder aplicarse al menos a un tipo de mascota. Además, debe habilitar la ejecución en cascada de todas las operaciones de JPA sobre este atributo.

```
26
27  @Entity
28  @Getter
29  @Setter
30  @Table(name = "care")
31  public class Care extends BaseEntity {
32      @NotBlank
33      @Size(min = 3, max = 40)
34      @Column(name = "name", unique = true)
35      String name;
36
37      @Min(2)
38      @Max(120)
39      @Column(name = "care_duration")
40      int careDuration;
41
42      @NotEmpty
43      @ManyToMany(cascade = CascadeType.ALL)
44      Set<PetType> compatiblePetTypes;
45
46  }
47
```

Test 2 – Creación de la Entidad CareProvision y su repositorio asociado

Modificar la clase "CareProvision" para que sea una entidad. Esta clase está alojada en el paquete "org.springframework.samples.petclinic.care", y debe tener los siguientes atributos y restricciones:

- El atributo de tipo entero (Integer) llamado "id" actuará como clave primaria en la tabla de la base de datos relacional asociada a la entidad.
- Crear una relación de N a 1 unidireccional desde "CareProvision" hacia "Care" obligatoria, usando como nombre del atributo "care" en la clase "CareProvision". Esta relación indicará el cuidado realizado a la mascota durante la visita (corte de pelo, desparasitación, etc.).
- El atributo de tipo cadena llamado "userRating", donde los owners nos dejarán un comentario que debe comenzar con una valoración de entre 0 y 9 estrellas. Por tanto, la cadena de la descripción debe comenzar con el texto "Care rated with X stars", donde X es un valor entero entre 0 y 9. La cadena puede contener cualquier otro texto libre que deseen los usuarios después de la valoración, para poder recibir sugerencias o quejas, por ejemplo, "Care rated with 9 stars, I am quite happy with the service!". La expresión regular que modela dicho comportamiento es `^Care rated with [0-9] stars.*$`
- Debe tener una relación N a 1 unidireccional (**no** obligatoria) desde "CareProvision" hacia "Visit" utilizando un atributo llamado "visit". Esta relación indicará la visita durante la que se presta el servicio/cuidado/tratamiento.

Modificar el interfaz "CareProvisionRepository" alojado en el mismo paquete para que extienda a CrudRepository y permita gestionar cuidados realizados a mascotas durante las visitas (CareProvision).

Además, se debe descomentar y anotar el método "List<Care> findAllCares ()" en dicho repositorio para que permita obtener todos los cuidados existentes.

```java
@Entity
@Getter
@Setter
@Table(name = "care_provision")
public class CareProvision extends BaseEntity {
    @Column(name = "user_rating")
    @Pattern(regexp = "^Care rated with [0-9] stars.*$")
    String userRating;

    @NotNull
    @ManyToOne(optional = false)
    Care care;

    @ManyToOne(optional = true)
    Visit visit;

}
```

```java
@Repository
public interface CareProvisionRepository extends CrudRepository<CareProvision, Integer> {
    List<CareProvision> findAll();

    Optional<CareProvision> findById(int id);
    CareProvision save(CareProvision p);

    @Query("SELECT c FROM Care c")
    List<Care> findAllCares();
    //List<Care> findCompatibleCares(PetType petType, Care additionalCare);
    //Care findCareByName(String name);
    //List<CareProvision> findCaresProvidedByVisitId(Integer visitId);
    //Page<CareProvision> findAllPaginatedCareProvisions(Pageable p);
}
```

Test 3 – Modificación del script de inicialización de la base de datos para incluir dos cuidados, asociados a perros, hamsters y gatos, y dos prestaciones de cuidados en visitas

Modificar el script de inicialización de la base de datos, para que se creen los siguientes cuidados (Care) y prestaciones de cuidados (CareProvision):

Care 1:

- id: 1
- name: "Hair cut"
- careDuration: 30

Nota: Este cuidado debe estar asociado a perros (PetType con id 2) y hamsters (PetType con id 6), para ello deberá insertar las filas que sean necesarias en la tabla intermedia de la relación N a N. Tenga en cuenta que el orden en que se especifican las inserciones es importante para que la inicialización sea correcta.

Care 2:

- id: 2
- name: "Exotic shampoo cleaning"
- careDuration: 15

Nota: Este cuidado debe estar asociado a hamsters (PetType con id 6) y gatos (PetType con id 1), para ello deberá insertar las filas que sean necesarias en la tabla intermedia de la relación N a N. Tenga en cuenta que el orden en que se especifican las inserciones es importante para que la inicialización sea correcta.

CareProvision 1:

- id: 1
- visit_id: 1
- user_rating: Care rated with 8 stars
- care_id:1

CareProvision 2:

- id: 2
- visit_id: 2
- user_rating: Care rated with 9 stars, I am quite happy!
- care_id: 2

Primero inserto solo los valores que figuran en el enunciado del ejercicio y ejecuto los tests.

```
INSERT INTO care(id,name,care_duration) VALUES(1,'Hair cut',30);
INSERT INTO care(id,name,care_duration) VALUES(2,'Exotic shampoo cleaning',15);

INSERT INTO care_provision(id,visit_id,user_rating,care_id)
    VALUES(1,1,'Care rated with 8 stars',1);
INSERT INTO care_provision(id,visit_id,user_rating,care_id)
    VALUES(2,2,'Care rated with 9 stars, I am quite happy!',2);
```

Nos va a dar error, me voy al log de los test para ver cual es el nombre de la tabla auxiliar autogenerada por Hibernate para la relación ManyToMany:

```
Hibernate:

    create table care_compatible_pet_types (
       cares_id integer not null,
        compatible_pet_types_id integer not null,
        primary key (cares_id, compatible_pet_types_id)
    )
```

Siguiendo la estructura de la tabla y sus atributos podemos insertar los valores necesarios para la relación que se especifican en el enunciado

```
INSERT INTO care(id,name,care_duration) VALUES(1,'Hair cut',30);
INSERT INTO care(id,name,care_duration) VALUES(2,'Exotic shampoo cleaning',15);

INSERT INTO care_compatible_pet_types(cares_id,compatible_pet_types_id) VALUES(1,2);
INSERT INTO care_compatible_pet_types(cares_id,compatible_pet_types_id) VALUES(1,6);
INSERT INTO care_compatible_pet_types(cares_id,compatible_pet_types_id) VALUES(2,1);
INSERT INTO care_compatible_pet_types(cares_id,compatible_pet_types_id) VALUES(2,6);

INSERT INTO care_provision(id,visit_id,user_rating,care_id)
    VALUES(1,1,'Care rated with 8 stars',1);
INSERT INTO care_provision(id,visit_id,user_rating,care_id)
    VALUES(2,2,'Care rated with 9 stars, I am quite happy!',2);
```

Es importante insertar los datos en el orden adecuado

Test 4 – Creación de un Servicio de gestión de los cuidados para mascotas

Modificar la clase "CareService", para que sea un servicio Spring de lógica de negocio, que permita obtener todos los cuidados realizados (CareProvision). No modifique por ahora la implementación de los demás métodos del servicio.

La clase del servicio debe estar anotada con @Service. Además debemos crear un atributo con el repositorio que contenga los métodos con las consultas necesarias.

```
@Service
public class CareService {

    @Autowired
    CareProvisionRepository cpr;
```

Anotamos el método con @Transactional. Hacemos que el método haga una llamada al método del repositorio que contiene la consulta deseada y lo devuelva.

```
@Transactional(readOnly = true)
public List<CareProvision> getAllCaresProvided(){
    return cpr.findAll();
}
```

Test 5– Creación de un Formatter de cuidados

Implementar los métodos del *formatter* para la clase Care (usando la clase llamada "CareFormatter" que está ya alojada en el mismo paquete "care" con el que estamos trabajando). El formatter debe mostrar los cuidados usando la cadena de su nombre, y debe obtener un cuidado dado su nombre, buscándolo en la BD (para esto debe hacer uso del servicio de gestión de cuidados, usando el método del servicio que busca por nombre "getCare(String nombre)", y no del repositorio, aunque quizás necesite crear o descomentar un método en el repositorio e invocarlo desde el servicio). Recuerde que, si el formatter no puede obtener un valor apropiado a partir del texto proporcionado, debe lanzar una excepción de tipo "ParseException".

Se crea un atributo tipo 'CareService' y se inicializa el objeto 'CareFormatter' pasando como parámetro el 'CareService'

```java
@Component
public class CareFormatter implements Formatter<Care>{

    private CareService careService;
    @Autowired
    public CareFormatter(CareService careService) {
        this.careService = careService;
    }

    @Override
    public String print(Care object, Locale locale) {
        return object.getName();
    }

    @Override
    public Care parse(String text, Locale locale) throws ParseException {
        Care care = careService.getCare(text);
        if (care == null) {
            throw new ParseException(s: "Care not found", errorOffset: 0);
        }
        return care;
    }

}
```

El primer método devuelve el nombre de un cuidado pasado por parámetro, lo hacemos fácilmente usando el 'getter'.
Para el segundo método creamos un objeto de tipo Care y lo inicializamos con una llamada al método del servicio que obtiene el cuidado recibiendo un nombre. Se devuelve este cuidado salvo en el caso de que no lo encuentre, en el que se lanzará una excepción tipo 'ParseException'.

```java
@Query("SELECT c FROM Care c WHERE c.name = ?1")
Care findCareByName(String name);
```

En el repositorio tenemos descomentar y añadir la Query al método para que encuentre el cuidado

```java
public Care getCare(String careName) {
    return cpr.findCareByName(careName);
}
```

En el servicio llamamos al método del repositorio

Test 6 - Crear una relación reflexiva N a N en la entidad Care

Cree una relación N a N reflexiva (con la propia entidad Care) unidireccional, que represente el conjunto de cuidados incompatibles con el cuidado actual en la misma visita. Por ejemplo, si se realiza una sesión de cardado y corte de pelo, no es recomendable realizar una sesión de tratamiento capilar desinfectante con productos fuertes en la misma visita, puesto que podemos irritar el cuero cabelludo de la mascota. Pare ello, añada un atributo llamado "incompatibleCares" de tipo Set<Care>. Debe habilitar la ejecución de todas las operaciones de JPA en casada para este atributo.

Modifique el script de inicialización de los datos para hacer que el cuidado 1 sea incompatible con el cuidado 2 y viceversa.

Se añade en la clase 'Care' el atributo que se indica en el enunciado:

```
@ManyToMany(cascade = CascadeType.ALL)
Set<Care> incompatibleCares;
```

Consulto el log de los tests para saber qué nombre y atributos tiene la tabla auxiliar para la relación reflexiva:

```
Hibernate:

    create table care_incompatible_cares (
       care_id integer not null,
        incompatible_cares_id integer not null,
        primary key (care_id, incompatible_cares_id)
    )
```

Se modifica el script de inicialización de la base de datos con los registros indicados:

```
INSERT INTO care_incompatible_cares(care_id,incompatible_cares_id) VALUES(1,2);
INSERT INTO care_incompatible_cares(care_id,incompatible_cares_id) VALUES(2,1);
```

Test 7 – Anotar el repositorio para obtener los cuidados (Care) disponibles para un tipo de mascota, que no sean incompatibles con otro tipo de cuidado dado

Crear una consulta personalizada que pueda invocarse a través del método “getAllCompatibleCares” del servicio de gestión anterior. La consulta personalizada debe obtener los cuidados disponibles para un tipo de mascota especificada mediante un parámetro, que no sean incompatibles con otro tipo de cuidado que se proporcionará también como parámetro (y que se supone que ha sido proporcionado ya a la mascota durante la visita).

Creo la consulta en el repositorio:

```
@Query("SELECT c FROM Care c WHERE :type MEMBER OF c.compatiblePetTypes AND NOT (:care MEMBER OF c.incompatibleCares)")
List<Care> findCompatibleCares(@Param("type") PetType petType, @Param("care") Care additionalCare);
```

Los parámetros son objetos con tipos definidos, por lo que debemos usar la notación @Param para indicar su nombre al realizar la consulta

Modificamos el método del servicio para llamar al método del repositorio:

```
public List<Care> getAllCompatibleCares(PetType petTypeName, Care additionalCareName){
    return cpr.findCompatibleCares(petTypeName, additionalCareName);
}
```

Test 8– Registro de cuidados realizados a una mascota durante una visita.

Implementar el método “save” del servicio de gestión de cuidados. Si durante la visita ya se ha prestado un cuidado que es incompatible con el que se desea grabar, se debe lanzar una excepción de tipo “NonCompatibleCaresException” (esta clase está ya creada en el paquete cares). Para poder hacer esto deberá crear una consulta personalizada en el repositorio que obtenga los cuidados proporcionados durante una visita, y exponerla a través del método “getCaresProvidedInVisitById(Integer visitId)” del servicio de gestión de cuidados. Además, el servicio debe asegurarse de que el cuidado es compatible con el tipo de mascota al que se pretende prestar el cuidado (no debemos permitir registrar cuidados de corte de pelo a peces, por ejemplo). Si el tipo de la mascota que viene a la clínica durante la visita no está entre las compatibles con el cuidado, deberá lanzarse una excepción de tipo “UnfeasibleCareException”. Además, en caso de lanzarse cualquiera de las dos excepciones, la transacción asociada a la operación de guardado del cuidado debe echarse atrás (hacer rollback).

Creo en el repositorio la consulta con el conjunto de cuidados proporcionados durante una visita:

```
@Query("SELECT c FROM CareProvision c WHERE c.visit.id = ?1")
List<CareProvision> findCaresProvidedByVisitId(Integer visitId);
```

Llamo a dicha consulta en el método del servicio indicado:

```
public List<CareProvision> getCaresProvidedInVisitById(Integer visitId){
    return cpr.findCaresProvidedByVisitId(visitId);
}
```

Anoto el método con @Transactional y añado al atributo rollbackFor las dos excepciones indicadas en el enunciado.
Creo una lista con todos los cuidados proporcionados en la visita utilizando el método anterior. También obtengo el cuidado a guardar del parámetro recibido en el método.
Creamos dos ‘if’ en el que se comprueban las dos condiciones del enunciado (dos cuidados incompatibles entre sí, y cuidados incompatibles para el tipo de mascota). En caso de que se cumpla alguna condición, se lanzará la excepción correspondiente.

```
@Transactional(rollbackFor = {NonCompatibleCaresException.class, UnfeasibleCareException.class})
public CareProvision save(CareProvision p) throws NonCompatibleCaresException, UnfeasibleCareException {
    List<CareProvision> caresProvided = cpr.findCaresProvidedByVisitId(p.getVisit().getId());
    Care care = p.getCare();
    if(caresProvided.stream().anyMatch(provided->care.getIncompatibleCares().contains(provided.getCare()))) {
        throw new NonCompatibleCaresException();
    }

    if(!care.getCompatiblePetTypes().contains(p.getVisit().getPet().getType()))
        throw new UnfeasibleCareException();
    return cpr.save(p);
}
```

Test 9 – Creación de método que permita obtener todos los cuidados proporcionados con paginación

Se pide implementar un método que permita devolver un listado de cuidados proporcionados paginado. El método se deberá implementar en el servicio de gestión de cuidados (se ha creado un método vacío llamado ”getPaginatedCareProvisions” para ello en dicha clase) y habrá que modificar el repositorio creando un método que devuelva los datos paginados, invocándolo desde el servicio.

Método del repositorio:

```
@Query("SELECT c FROM CareProvision c")
Page<CareProvision> findAllPaginatedCareProvisions(Pageable p);
```

Método del servicio:

```
public Page<CareProvision> getPaginatedCareProvisions(Pageable pageable){
    return cpr.findAllPaginatedCareProvisions(pageable);
}
```

Test 10 – Creación de Controlador para la creación de nuevos cuidados prestados durante las visitas.

Crear un método de controlador en la clase anterior que responda a peticiones tipo post en la url:

http://localhost:8080/visit/<X>/cares/create

Donde <X> es el identificador de la visita al que se asociará el cuidado.

Este método debe encargarse de validar los datos del cuidado administrado, mostrar los errores encontrados si existieran a través del formulario proporcionado en el fichero "cares/createOrUpdateProvidedCareForm", y si no existen errores, almacenar el nuevo cuidado prestado a través del servicio de gestión de cuidados prestados. Tras grabar el cuidado proporcionado a la mascota, en caso de éxito, se usará redirección para volver a la página de inicio de la aplicación ("/").

En caso de que el servicio de gestión de cuidados lance alguna de las dos excepciones se debe mostrar la página de error personalizada "cares/InvalidCare.jsp". Para ello debe hacerse uso del objeto de configuración de controladores proporcionado en la clase "ExceptionHandlingControllerAdvice", y anotar el método de gestión de las excepciones.

En primer lugar creo un atributo para el servicio y una instancia del controlador en la que inicializo el servicio.
Como quiero que el método que voy a crear responda a peticiones post, lo anoto con @PostMapping y añado la URL como atributo
El método, que devuelve un objeto ModelAndView, recibe como parámetros:

- Un 'id' que corresponde a la visita anotado con @PathParam, que es el dato que se incluye en la URL
- Un objeto 'CareProvision' anotado con @Valid para la validación
- Un objeto 'BindingResult' que contiene el resultado del proceso de binding

Primero creo un objeto tipo ModelAndView vacío.
A continuación compruebo si el BindingResult tiene errores. Si se encuentran errores establezco la vista del formulario en el ModelAndView. En caso de que no hayan errores, llamo al método save del servicio pasando el 'CareProvision' como parámetro y añado la vista de inicio al modelo.
Por último devuelvo el modelo, que llevará a la vista que se ha establecido.

```java
@Controller
public class CareController {

    CareService service;
    @Autowired
    public CareController(CareService service) {
        this.service = service;
    }

    @PostMapping("/visit/{visitId}/cares/create")
    public ModelAndView process(@PathParam("visitId") Integer visitId, @Valid CareProvision care, BindingResult br)
                                throws UnfeasibleCareException, NonCompatibleCaresException {
        ModelAndView res = new ModelAndView();
        if(br.hasErrors()) {
            res.setViewName("cares/createOrUpdateProvidedCareForm");
        } else {
            service.save(care);
            res.setViewName("redirect:/");
        }
        return res;
    }

}
```

Por último modificamos el método en la clase de gestión de controladores 'ExceptionHandlingControllerAdvice' añadiendo la anotación @ExceptionHandler con las dos excepciones requeridas, y para que redirija a la vista indicada en el enunciado.

```
@ExceptionHandler({UnfeasibleCareException.class, NonCompatibleCaresException.class})
public String handleInvalidCareException(HttpServletRequest request, Exception ex){
    return "cares/InvalidCare";
}
```

Anotaciones de la clase

- Anotar con @Getter y @Setter
- Anotar siempre con @Entity
- Anotar con @Table(name = "xxxxx")

Anotaciones de los atributos

- @NotNull comprueba que un atributo no es null. Es decir, comprueba que un atributo no tiene ningún valor.
- @NotEmpty comprueba que un atributo, ya sea un String o una lista no está vacío.
- @NotBlank comprueba que un atributo no es null y tampoco está vacío.
- @Size(min = x, max = y) nos permite especificar la longitud máxima y mínima para un String
- @Min(x) y @Max(y) nos permite especificar el valor máximo y mínimo que puede tomar un valor numérico
- Pattern(reg_exp = "xxxx")
- @ManyToMany(cascade = CascadeType.xxx)
  Set<ClaseRelacionada> nombreClase;