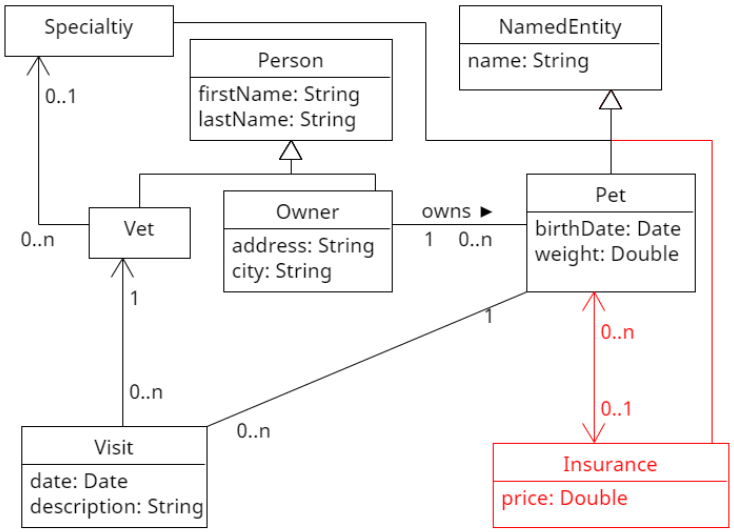
Control práctico 1 de DP1 2023/2024 (Control check 1)

Enunciado

En este ejercicio, añadiremos la funcionalidad de gestión de seguros de mascotas que se venderán en nuestra clínica de mascotas. Para ello realizaremos una serie de ejercicios basados en funcionalidades que implementaremos en el sistema, y validaremos mediante pruebas unitarias. Si desea ver el resultado que arrojarían las pruebas, puedes ejecutar el comando "*.\mvnw test*" en la carpeta raíz del proyecto. Cada prueba correctamente pasada valdrá un punto.



Para comenzar el control debe aceptar la tarea de este control práctico a través del siguiente enlace:

https://classroom.github.com/a/bV7juKN_

Al aceptar dicha tarea, se creará un repositorio único individual para usted, debe usar dicho repositorio para realizar el control práctico. Debe entregar la actividad en EV asociada al control check proporcionando como texto la dirección url de su repositorio personal. Recuerde que además debe entregar su solución del control.

La entrega de su solución al control se realizará mediante un único comando "*git push*" a su repositorio individual. Recuerde que debe hacer push antes de cerrar sesión en la computadora y abandonar el aula, de lo contrario, su intento se evaluará como no presentado. Su primera tarea en este control será clonar (recuerde que si va a usar los equipos del aula para realizar el control necesitará usar un token de autenticación de GitHub como clave, tiene un documento de ayuda a la configuración en el propio repositorio del control). A continuación, deberá importar el proyecto en su entorno de desarrollo favorito y comenzar los ejercicios abajo listados. Al importar el proyecto, el mismo puede presentar errores de compilación. No se preocupe, si existen, dichos errores irán despareciendo conforme usted vaya implementando los distintos ejercicios del control.

Nota importante 1: No modifique los nombres de las clases ni la signatura (nombre, tipo de respuesta y parámetros) de los métodos proporcionados como material de base para el control. Las pruebas que se usan para la evaluación dependen de que las clases y los métodos tengan dicha la estructura y nombres proporcionados. Si los modifica probablemente no pueda hacer que pasen las pruebas, y obtendrá una mala calificación.

Nota importante 2: No modifique las pruebas unitarias proporcionadas como parte del proyecto bajo ningún concepto. Aunque modifique las pruebas en su copia local del Proyecto, éstas serán restituidas mediante un comando git previamente a la ejecución de las pruebas para la emisión de la nota final, por lo que sus modificaciones en las pruebas no serán tenidas en cuenta en ningún momento.

Nota importante 3: Mientras haya ejercicios no resueltos habrá tests que no funcionan y, por tanto, el comando "*mvnw install*" finalizará con error. Esto es normal debido a la forma en la que está planteado el control y no hay que preocuparse por ello. Si se quiere probar la aplicación se puede ejecutar de la forma habitual pese a que "mvn install" finalice con error.

Nota importante 4: La descarga del material de la prueba usando git, y la entrega de su solución con git a través del repositorio GitHub creado a tal efecto forman parte de las competencias evaluadas durante el examen, por lo que no se aceptarán entregas que no hagan uso de este medio, y no se podrá solicitar ayuda a los profesores para realizar estas tareas.

Nota importante 5: No se aceptarán como soluciones válidas proyectos cuyo código fuente no compile correctamente o que provoquen fallos al arrancar la aplicación en la inicialización del contexto de Spring. Las soluciones cuyo código fuente no compile o incapaces de arrancar el contexto de Spring serán evaluadas con una nota de 0.

Test 1 – Creación de la entidad Insurance y su repositorio asociado

Se propone modificar la clase "Insurance" para que sea una entidad. Esta entidad estará alojada en el paquete "org.springframework.samples.petclinic.insurance", y debe tener los siguientes atributos y restricciones:

- Un atributo de tipo Integer llamado "id" que actúe como clave primaria en la tabla de la base de datos relacional asociada a la entidad.
- Un atributo "name" de tipo cadena obligatorio, y cuya longitud mínima son 3 caracteres y la máxima 50. Además, dicho nombre debe ser único**.**
- Un atributo "price" de tipo coma flotante doble (Double) obligatorio, que solo podrá tomar valores positivos (cero incluido).
- Un atributo "pets" de tipo Set<Pet> opcional. No elimine la anotación @Transient por ahora.

Se propone modificar el interfaz "InsuranceRepository" alojado en el mismo paquete, para que extienda a CrudRepository. No olvide especificar sus parámetros de tipo.

Test 2 – Modificación de la Entidad "Pet"

Modificar la clase denominada "Pet" alojarse en el paquete "org.springframework.samples.petclinic.pet" para que incluya el siguiente atributo y restricción además de los que ya posee:

- Un atributo opcional “insurance” de tipo “Insurance” que haga referencia a la relación con la clase Insurance. Incluir las anotaciones necesarias para crear la relación 0..N a 0..1 bidireccional entre Pet e Insurance. Para ello tendrá que actualizar también la entidad Insurance de manera apropiada

Test 3 – Modificación del script de inicialización de la base de datos para incluir dos seguros (y modificar dos de las inserciones de mascotas para asociarle dichos seguros)

Modificar el script de inicialización de la base de datos, para que se creen los siguientes seguros y modificaciones de las mascotas:

Seguro 1:

- Id: 1
- Name: "Premium Insurance"
- Price: 1000.0

Seguro 2:

- Id: 2
- Name: "Medium Insurance"
- Price: 500.0

Además, debe modificar el script de inicialización de la base de datos para que cada mascota pueda o no tener asociado una instancia de Insurance, añadiendo las siguientes asociaciones al script:

- El Pet cuyo id es 3 se asocie con el seguro con id=1
- El Pet cuyo id es 4 se asocie con el seguro con id=1
- El Pet cuyo id es 5 se asocie con el seguro con id=2

Tenga en cuenta que el orden en que aparecen los INSERT en el script de inicialización de la base de datos es relevante al definir las asociaciones.

Test 4 – Creación de un Servicio de gestión de seguros

Modificar la clase “InsuranceService”, para que sea un servicio Spring de lógica de negocio, que permita obtener todos los seguros (como una lista) y grabar los seguros usando el repositorio. No modifique por ahora la implementación de los demás métodos del servicio. Recuerde que los métodos del servicio deberían estar anotados con @Transactional (y los parámetros de anotación adecuados en cada caso).

Test 5 – Creación de un método método asociado en el servicio de gestión de seguros para obtener los seguros por nombre de mascota

Implementar el método “`getInsuranceOfPet(String petName)`” del servicio de gestión de seguros, que obtenga el seguro asociado a una mascota a partir del nombre de la mascota. Para ello, deberá a su vez implementar el método “`getPetByName(String petName)`” del servicio de mascotas (PetService) que permita obtener la mascota con el nombre pasado por parámetro. Para esto último deberá añadir un método apropiado en el repositorio de mascotas (PetRepository). Recuerde que los métodos del servicio deberían estar anotados con @Transactional (y los parámetros de anotación adecuados en cada caso).

Test 6– Creación de consulta personalizada de seguros cuyo precio está delimitado por un rango determinado

Crear una consulta personalizada anotando un método llamado "`findByPriceBetween`" en el repositorio, de manera que tome como parámetros un límite de coste inferior y otro de coste superior (ambos parámetros de tipo Double), que devuelva todos los seguros cuyo precio se encuentra comprendido entre dichos límites. Extender el servicio de gestión de servicios implementando el método llamado "`getInsurancesBetween`" para que invoque al repositorio. Recuerde que los métodos del servicio deberían estar anotados con @Transactional (y los parámetros de anotación adecuados en cada caso).

Test 7 – Creación del Controlador y el método para la creación de nuevos seguros.

Se propone crear un método de controlador en la clase "InsuranceController" que responda a peticiones tipo POST en la URL "/api/v1/insurances" y se encargue de validar los datos del nuevo seguro, mostrar los errores encontrados si existieran como parte de la respuesta (con código de estado 400 en ese caso), y si no existen errores, almacenar el seguro a través del servicio de gestión de seguros y devolver el código de estado 201 (CREATED) junto con el propio seguro grabado en formato JSON en el cuerpo de la petición.. Por tanto, deberá implementar el método "`save`" del servicio de gestión de seguros. Recuerde que dicho método debe ser transaccional.

Test 8 – Obtención de seguro por ID y modificación a través de la API de la representación de las mascotas asociadas a dicho seguro

Se propone modificar el controlador de seguros para incluir una operación que permita obtener los datos de un seguro en base a su id. El método debe responder a peticiones tipo GET en la URL: "/api/v1/insurances/X", donde X es el id del seguro.

Si se solicita un seguro cuya id no existe en la base de datos se debe devolver una respuesta con código 404.

Se propone además modificar la representación de los datos proporcionados por la API para que las mascotas asociadas al seguro se muestren como una cadena de texto simple con su nombre. Ejemplo de JSON de representación de un producto:

```
{
        "id": 1,
        "name": "Premium Insurance",
        "price": 1000.0,
        "pets": [ "Rosy", "Jewel" ]
}
```

Puede usar el mecanismo que considere oportuno para la codificación de los datos (crear un JSON serializer / deserializador y anotar el atributo en la clase "Insurance" o usar un DT[1]O en esta operación específica del controlador). Recuerde que para obtener el seguro correspondiente debe usar el servicio de gestión de seguros, y que si escoge la opción del JSON serializer, también debe implementar el deserializer obligatoriamente.

Test 9 – Modificación de seguros existentes y configuración de seguridad

Se propone modificar el controlador de seguros para incluir una operación que permita modificar un seguro en base a su ID. El método debe responder a peticiones tipo PUT en la URL: "/api/v1/insurances/X", donde X es el id del seguro. Para ello deberá crear también un deserializador o DTO que permita construir objetos con las mascotas asociadas expresadas directamente como un array con sus nombres (véase la representación del Test 8).

Si se solicita modificar un seguro cuya id no existe en la base de datos se debe devolver una respuesta con código 404.

El controlador debe encargarse de validar los datos del seguro modificado, mostrar los errores encontrados si existieran como parte de la respuesta (con código de estado 400 en ese caso), y si no existen errores, almacenar el seguro a través del servicio de gestión de seguros.

Recuerde configurar esta operación como accesible únicamente para los administradores del sistema (authority="admin").

Test 10 - Implementación de una regla de negocio que haga que no se puedan realizar subidas de precio en seguros existentes superiores al 70% de su precio anterior

Se solicita implementar una regla de negocio en el sistema que impida realizar modificaciones en un seguro que suponga un aumento de precio superior al 70% de su precio. Para ello debe hacer uso de la excepción "`UnfeaseibleInsuranceModificationException`", que deberá lanzarse por parte del servicio de gestión de seguros en caso de que se detecte esta situación. El controlador deberá capturar dicha excepción y devolver un código de respuesta 400 en ese caso. La transacción asociada a la modificación del seguro debe echarse atrás en caso de que se lance la excepción.

[1] En este caso recomendamos el uso de DTOs tanto para el ejercicio 8 como el 9 puesto que la solución es más sencilla con este mecanismo.