Resuelto-ControlCheck2-G2-2425.p...

CVV__

Diseño y Pruebas i

3º Grado en Ingeniería Informática - Ingeniería del Software

Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

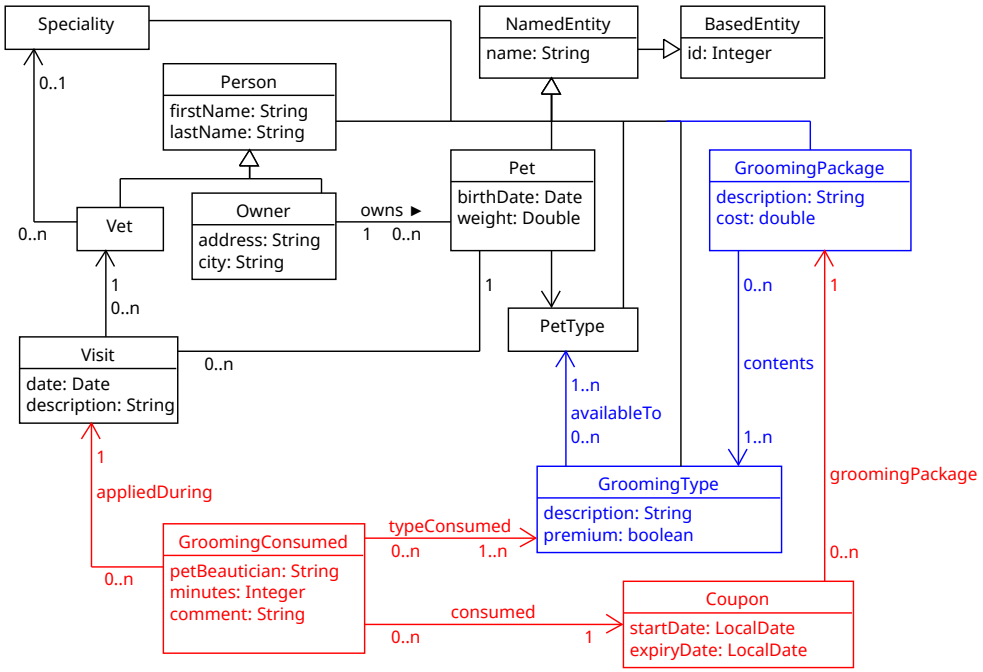
Accede al documento original

Control práctico de DP1 2024-2025 (Control check 2)

Enunciado

En este ejercicio, añadiremos la funcionalidad de ofrecer servicios de belleza a las mascotas (en inglés *pet grooming*) de forma combinada junto a las visitas a los veterinarios. Esto lo modelaremos con una clase llamada `GroomingType` que incluye una descripción. Ejemplos de `GroomingType` pueden ser "cepillado", "baño", "baño con masaje" o "corte de uñas". Los `GroomingType` se agrupan en `GroomingPackage`, que incluyen, además del nombre, una descripción y su coste. Los `GroomingType` están disponibles para un determinado tipo de mascotas (`PetType`). Por ejemplo, el "corte de uñas" está disponibles para gatos y perros, pero no para serpientes. Un `GroomingPackage` puede constar únicamente de un `GroomingType` como "baño con masaje" o de varios `GroomingType` como "cepillado + baño + corte de uñas". Finalmente, los usuarios compran un `Coupon` de un determinado `GroomingPackage` que van consumiendo (`GroomingConsumed`) en una o varias visitas. Los `Coupon` además tiene una fecha de validez determinada por los atributos `startDate` (inicio) y `expirationDate` (caducidad). Cuando un servicio de belleza es consumido (`GroomingConsumed`) en una visita, se debe indicar el nombre del esteticista de mascotas que realizó el servicio, el número de minutos invertidos y, además, es posible agregar un comentario sobre el servicio.

El diagrama UML que describe estas clases y relaciones es el siguiente:



Las clases para las que realizaremos el mapeo objeto-relacional como entidades JPA se han señalado en rojo. Las clases en azul son clases que se proporcionan ya mapeadas pero con las que se trabajará durante el control de laboratorio.

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 en backend, puede ejecutarlas (bien mediante su entorno de desarrollo favorito, bien mediante el comando "*mvnw test*" en la carpeta raíz del proyecto). Cada ejercicio correctamente resuelto valdrá dos puntos, el número de casos de prueba de cada ejercicio puede variar entre uno y otro y la nota se calculará en base al porcentaje de casos de prueba que pasan. Por ejemplo, si pasan la mitad (50%) de los casos de prueba de un ejercicio, usted obtendrá un punto (2*0,5=1), y si pasan un 10% obtendrá 0,2 puntos.

Para comenzar el control debe aceptar la tarea de este control práctico a través del siguiente enlace: https://classroom.github.com/a/XYaucvyc

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 el ordenador 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.

Nota importante 6: Los ejercicios del examen son todos independientes salvo el 2 que depende del 1.

Test 1 – Creación de las entidades GroomingConsumed y Coupon y sus relaciones (2 puntos)

Modifique las clases "GroomingConsumed" y "Coupon" alojadas en el paquete "org.springframework.samples.petclinic.grooming" para que sean una entidad y elimine las anotaciones @Transient de estas clases. Estas entidades deben tener los siguientes atributos y restricciones:

Para la clase GroomingConsumed:

- 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 "*petBeautician*" obligatorio que debe tener una longitud mínima de 5 caracteres y máximo de 60 y no puede estar formada por caracteres vacíos (espacios, tabuladores, etc.).
- Un atributo de tipo entero (Integer) llamado "*minutes*" obligatorio que debe ser un número estrictamente positivo.
- Un atributo opcional de tipo cadena (String) llamado "*comment*" para registrar los comentarios sobre el servicio realizado.

Para la clase Coupon:

- 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.
- Un atributo de tipo fecha (LocalDate) llamado "*startDate*", que representa la fecha en que comienza la oferta. Seguirá el formato "dd/MM/yyyy" (puede usar como ejemplo la clase Pet y su fecha de nacimiento para ver cómo se especificar dicho formato, pero nótese que el patrón del formato es distinto). Este atributo debe ser obligatorio y se almacenará en la base de datos con el nombre de columna "start".
- El atributo de tipo fecha (LocalDate) llamado "*expiryDate*", que representa la fecha en que termina la oferta, seguirá el formato "dd/MM/yyyy". Este atributo debe ser obligatorio. En la base de datos se almacenará con el nombre de columna "finish".

Además, se pide crear las siguientes relaciones entre las entidades.

1. Cree tres relaciones unidireccionales desde "GroomingConsumed" hacia "Visit" utilizando el atributo "*appliedDuring"*, de "GroomingConsumed" hacia "GroomingType" utilizando el atributo "*typeConsumed"* y de "GroomingConsumed" hacia "Coupon" utilizando el atributo "*consumed*", que expresen las relaciones que aparecen en el diagrama UML (mostrado en la primera página de este enunciado) respetando sus cardinalidades.
2. Cree una relación unidireccional desde "Coupon" hacia "GromingPackage" que represente la que aparece en el diagrama UML, teniendo en cuenta la cardinalidad y usando el atributo "*groomingPackage*" en la clase "Coupon"

Debe asegurarse de que las relaciones expresan adecuadamente la cardinalidad que muestra el diagrama UML, por ejemplo, el atributo groomingPackage no puede ser nulo puesto que la cardinalidad es 1.
Finalmente, modifique las interfaces "GroomingConsumedRepository" y "CouponRepository" alojadas en el mismo paquete para que extiendan a CrudRepository. No olvide especificar sus parámetros de tipo.

Test 2 – Modificación del script de inicialización de la base de datos (2 puntos)

Modifique el script de inicialización de la base de datos, para que se creen los siguientes Coupon:

Coupon 1:

- *id*: 1
- *startDate*: 05-01-2024
- *expiryDate*: 21-02-2024
- *grooming package*: 1

Coupon 2:

- *id*: 2
- *startDate*: 01-12-2024
- *expiryDate*: 31-1-2025
- *grooming package*: 3

Modifique el script de inicialización de la base de datos, para que se creen los siguientes `GroomingConsumed`:

- `GroomingConsumed` 1:
  - id: 1
  - petBeautician: "Mariano Rojas"
  - minutes: 20
  - comments: "A new visit is recommended in 15 days."

- `GroomingConsumed` 2:
  - id: 2
  - petBeautician: "Felipe Pineda"
  - minutes: 30
  - comments: "Anti-allergy shampoo was used."

- En el *`GroomingConsumed`* cuyo id es 1 se consume `GroomingType` 1 y `GroomingType` 3.
- En el *`GroomingConsumed`* cuyo id es 2 se consume `GroomingType` 7.
- El *`GroomingConsumed`* cuyo id es 1 se asocia con el `Coupon` 1 y con la `Visit` 1.
- El *`GroomingConsumed`* cuyo id es 2 se asocia con el `Coupon` 2 y con la `Visit` 10.

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 3 – Creación de controlador y componente frontend para GroomingTypes (2 puntos)

Test 3A – Creación del controlador para devolver los Grooming Types disponibles

Crear un método en el controlador "`GroomingTypeController`" (alojada en el paquete `org.springframework.samples.petclinic.grooming`) que permita devolver todos los `GroomingType` disponibles para el *`petType`* y los nombres de los paquetes donde se puede encontrar estos `GroomingType`. El método debe responder a peticiones tipo GET en la URL: http://localhost:8080/api/v1/groomingtypes/pettype/{petTypeID}

Es importante que dicho controlador devuelva los datos en el siguiente formato:

```
{
  petType: "cat",
  groomingTypes: [
      {name: "Hair cut", packages: ["Hair cut + Nails", "Bath + Legs massage", "Luxury package"]},
              …,
      {name: "Walk session", packages: ["Hair cut + Nails"]}
    ]
}
```

En caso de que el *`petTypeId`* proporcionado no exista, debe devolver el código de estado `404` (`NOT_FOUND`). Este endpoint de la API debería estar accesibles únicamente para usuarios de tipo Vet o de tipo Owner (que tengan la authority "`VET`" o bien la authority "`OWNER`").

Para implementar esta funcionalidad puede hacer uso del método `findPetTypeById` de `PetService` que devuelve un `PetType` dado su *id*, del método `findGroomingTypesByPetType` de `GroomingTypeService` que devuelve los `GroomingType` de un *`petType`* pasado por parámetro y del método `findPackageByGroomingType` de `GroomingPackageService` que devuelve los `GroomingPackage` de un *`groomingType`* pasado por parámetro.

Test 3B – Creación de un componente React de listado de servicios de belleza

Modificar el componente *React* proporcionado en el fichero "*frontend/src/grooming/index.js*" para que muestre un listado de los tipos de servicios de belleza disponibles en el sistema para un tipo de mascota. Para ello debe hacer uso de la API lanzando una petición tipo GET contra la URL api/v1/groomingtypes/pettype/{petTypeId} creada en el apartado anterior[1].

Este componente debe tomar como propiedad llamada `petTypeId` el identificador del tipo de mascota para el que se debe mostrar los tipos de servicios de belleza. Tras realizar la llamada a la API, el componente debe mostrar el nombre de la mascota como título de nivel 1 (`<h1>`) y los tipos de servicios de belleza (`GroomingType`) en una tabla (se recomienda usar el componente `Table` de `reactstrap`) incluyendo una columna para el nombre del tipo de servicio de belleza (`GroomingType`), y otra columna para el conjunto de paquetes de belleza (`GroomingPackage`) en las que aparece este tipo de servicio. Para esto último, el componente debe mostrar en la celda asociada una lista (preferiblemente no ordenada `<ul>`) con los nombres de los paquetes en la celda correspondiente.

Para poder lanzar este test y comprobar su resultado puede colocarse en la carpeta de *frontend* y ejecutar el comando `npm test` y pulsar 'a' en el menú de comandos de *jest*. Nótese que previamente debe haber lanzado al menos una vez el comando `npm install` para que todas las librerías de *node* estén instaladas.

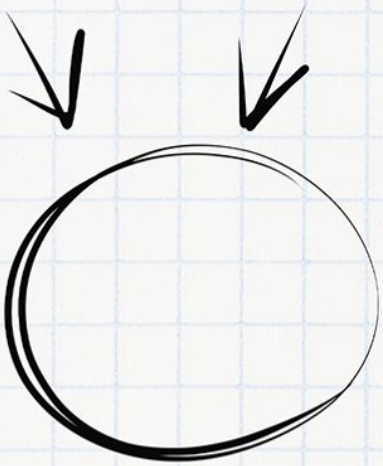
[1] Aunque se usa la misma URL, no hace falta que el apartado anterior esté completo para poder probar este.

Imagínate aprobando el examen

Necesitas tiempo y concentración

Planes	PLAN TURBO	PLAN PRO	PLAN PRO+
Descargas sin publi al mes	10	40	80
Elimina el video entre descargas	✓	✓	✓
Descarga carpetas	✗	✓	✓
Descarga archivos grandes	✗	✓	✓
Visualiza apuntes online sin publi	✗	✓	✓
Elimina toda la publi web	✗	✗	✓
Precios Anual	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo, ¿Qué nota vas a sacar?



WUOLAH

Test 4 – Anotar el repositorio de Grooming Packages con una consulta compleja (2 puntos)

Crear una consulta personalizada que pueda invocarse a través del método `findPackagesByCostGroomingTypes` del repositorio "`GroomingPackageRepository`" (alojado en el paquete `org.springframework.samples.petclinic.grooming`) que reciba dos parámetros: un valor double para representar el coste máximo del `GroomingPackage` (*maxCost*), y un valor Integer llamado *minServices* que represente un número mínimo de servicios de belleza asociados a un `GroomingPackage`.

El objetivo es devolver todos los `GroomingPackage` que tengan un precio menor o igual que *maxCost*, siempre que el número de tipos de servicios (`GroomingType`) que contengan los `GroomingPackage` sea igual o mayor a *minServices.*

A continuación, se muestra un ejemplo, sea la siguiente tabla de grooming packages y grooming types relacionados con esos paquetes,

GroomingPackage	contents GroomingType
{id:1, cost:50.0}	{id: 1, name: "Hair cut"}
{id:1, cost:50.0}	{id: 2, name: "Nail brushing"}
{id:1, cost:50.0}	{id: 3, name: "Legs massage"}
{id:1, cost:50.0}	{id: 4, name: "Walk session"}
{id:1, cost:50.0}	{id: 5, name: "Wash"}
{id:2, cost:30.0}	{id: 1, name: "Hair cut"}
{id:2, cost:30.0}	{id: 2, name: "Nail brushing"}
{id:2, cost:30.0}	{id: 3, name: "Legs massage"}
{id:3, cost:20.0}	{id: 7, name: "Clipping wings"}
{id:4, cost:45.0}	{id: 1, name: "Hair cut"}
{id:4, cost:45.0}	{id: 2, name: "Nail brushing"}
{id:4, cost:45.0}	{id: 6, name: "Nap"}

Si invocamos al método con los parámetros *maxCost*=45, *minServices*=3, el resultado debería ser el conjunto de `GroomingPackages` con *id*=2 e id=4. Dado que el `GroomingPackage` con *id*=2 tiene un *cost* menor que 60.0 (30.0), está asociado a 3 grooming types (1,2,3). El `GroomingPackage` con *id*=4 tiene un *cost* menor que 60.0 (45.0), está asociado a 3 grooming types (1,2,6) por lo que también aparecería en el resultado.

Finalmente, cree un método en la clase `GroomingPackageService` llamado `findPackagesByCostGroomingTypes` con dos parámetros: *maxCost y minService* (en este orden), que realice la llamada al método del repositorio y devuelva el resultado obtenido por éste.

Test 5 – Implementar una prueba para un algoritmo de validación del coste de paquetes (2 puntos)

En la clínica se ha decidido crear un sistema informático que permita validar el coste fijado a `GroomingPackages`. Para ello el algoritmo tomará como parámetros la instancia de `GroomingPackage` que queremos validar, el precio base de un `GroomingType` básico (atributo premium de GroomingType igual a falso) y el precio base de un `GroomingType` premium (atributo premium de `GroomingType` igual a verdadero).

El algoritmo debe comprobar que el coste fijado para el `GroomingPackage` (es decir, su atributo cost) es ***superior o igual*** al número de `GroomingType` básicos[2] del `GroomingPackage` multiplicado por el precio base de un básico más el número de `GroomingType` premium del `GroomingPackage` multiplicado por el precio base del premium. Además, si el paquete sólo incluye `GroomingType` premium, el precio base de un `GroomingType` premium recibido por parámetro debe multiplicarse por 1,10 (es decir, tiene un suplemento del 10%). En caso contrario, se lanzará una excepción del tipo `WrongPriceException`.

La interfaz del algoritmo está especificada en la interfaz "`PackageCostAlgorithm`" que se encuentra en el paquete "`org.springframework.samples.petclinic.groomingpackages.packagecost`". Se proporcionan cinco implementaciones del algoritmo (una correcta y cuatro incorrectas).

Modifique la clase de pruebas llamada "`PackageCostAlgorithmTest`" que se encuentra en la carpeta "*src/main/test/groomingpackages*" y especifique tantos métodos con casos de prueba como considere necesarios para validar el correcto funcionamiento del algoritmo. Nótese que la clase tiene un atributo de tipo `PackageCostAlgorithm` llamado `algorithm`, use dicho atributo como sujeto bajo prueba en todos sus métodos de prueba.

Su implementación del test **no debe usar mocks, ni anotaciones de pruebas de spring** (@DataJpaTest, @SpringBootTest, etc.), **ni tests parametrizados, y todos los métodos anotados con @Test deben ser sin parámetros**.

[2] Un `GroomingType` básico es aquel cuyo atributo `premium` es igual a `false`. Un `GroomingType` premium es aquel cuyo atributo `premium` es igual a `true`.

CONTROL CHECK 2 - GRUPO 2 - 24/25

TEST 1

```
@Setter
@Getter
@Entity
public class GroomingConsumed extends BaseEntity {

   @NotNull @Size(min=5, max=60) @NotBlank
   String petBeautician;

   @NotNull @Min(0)
   Integer minutes;

   String comment;

   @ManyToOne @NotNull
   Visit appliedDuring;

   @ManyToOne @NotNull
   Coupon consumed;

   @ManyToMany @NotNull
   List<GroomingType> typeConsumed;

}
```

```
@Getter
@Setter
@Entity
public class Coupon extends BaseEntity {

   @Column(name="start") @DateTimeFormat(pattern="dd/MM/yyyy")
@NotNull
   LocalDate startDate;


   @Column(name="finish")
@DateTimeFormat(pattern="dd/MM/yyyy") @NotNull
   LocalDate expiryDate;

   @ManyToOne @NotNull
   GroomingPackage groomingPackage;
```

```
}
```

```
public interface CouponRepository extends
CrudRepository<Coupon, Integer> {
   Coupon save(Coupon o);
   List<Coupon> findAll();
   Optional<Coupon> findById(Integer id);
}
```

```
 public interface GroomingConsumedRepository extends
CrudRepository<GroomingConsumed, Integer> {

   GroomingConsumed save(GroomingConsumed gc);
   List<GroomingConsumed> findAll();
   Optional<GroomingConsumed> findById(Integer id);

}
```

TEST 2

```
INSERT INTO coupon(id, start, finish, grooming_package_id)
VALUES (1, '2024-01-05', '2024-02-21', 1), (2, '2024-12-01',
'2025-01-31', 3);
INSERT INTO grooming_consumed(id, pet_beautician, minutes,
comment, applied_during_id, consumed_id) VALUES
(1, 'Mariano Rojas', 20, 'A new visit is recommended in 15
days.',1, 1), (2, 'Felipe Pineda', 30, 'Anti-allergy shampoo
was used.', 10, 2);
INSERT INTO
grooming_consumed_type_consumed(grooming_consumed_id,
type_consumed_id) VALUES (1, 1), (1, 3), (2, 7);
```

TEST 3

```
@RestController
public class GroomingTypeController {

   @Autowired
   private GroomingTypeService groomingTypeService;

   @Autowired
   private GroomingPackageService groomingPackageService;
```


```
    @Autowired
    private PetService petService;

    public GroomingTypeController(GroomingTypeService
groomingTypeService,
                                  GroomingPackageService
groomingPackageService,
                                  PetService petService){
        this.groomingTypeService=groomingTypeService;
        this.groomingPackageService=groomingPackageService;
        this.petService=petService;
    }

    @GetMapping("/api/v1/groomingtypes/pettype/{petTypeId}")
    @ResponseStatus(HttpStatus.OK)
    public ResponseEntity<PetDTO>
getAllGroomingInfo(@PathVariable Integer petTypeId){
        PetType petType =
petService.findPetTypeById(petTypeId);
        if (petType == null) throw new
ResourceNotFoundException("Pet type not found");

        List<GroomingType> gTypes =
groomingTypeService.findGroomingTypesByPetType(petType);

        List<GroomingTypeDTO> gTypesDTOs = new ArrayList<>();
        for (GroomingType gt: gTypes) {
            List<GroomingPackage> gPack =
groomingPackageService.findPackageByGroomingType(gt);
            gTypesDTOs.add(new GroomingTypeDTO(gt.getName(),
gPack));
        }

        PetDTO res = new PetDTO(petType.getName(), gTypesDTOs);
        ResponseEntity response =  new ResponseEntity<>(res,
HttpStatus.OK);
        return response;

    }
}
```

```
@Getter
@Setter
public class GroomingTypeDTO {
```

```
    String name;
    List<String> packages;

    public GroomingTypeDTO(String name, List<GroomingPackage>
gPack){
        this.name = name;
        this.packages = gPack.stream().map(gp ->
gp.getDescription()).toList();

    }
}
```

```
@Getter
@Setter
public class PetDTO {
    String petType;
    List<GroomingTypeDTO> groomingTypes;

    public PetDTO(String petType, List<GroomingTypeDTO> gTypes)
{
        this.petType = petType;
        this.groomingTypes = gTypes;
    }
}
```

SecurityConfiguration.java

```
.requestMatchers(HttpMethod.GET,
"/api/v1/groomingtypes/pettype/**").hasAnyAuthority("OWNER",
"VET")
```

TEST 4

```
@Query("SELECT gp FROM GroomingPackage gp WHERE gp.cost
<=:maxCost AND size(gp.contents) >=:minServices")
    public List<GroomingPackage>
findPackagesByCostGroomingTypes(double maxCost, Integer
minServices);
}
```

***GroomingPackageService**

```
@Transactional
public List<GroomingPackage>
findPackagesByCostGroomingTypes(Double maxCost, Integer
minService){
```

```
    return gr.findPackagesByCostGroomingTypes(maxCost,
minService);
}
```

TEST 5

```
@Test
public void someTest(){

    GroomingType gt1 = createGroomingType("1", true);
    GroomingType gt2 = createGroomingType("2", true);
    GroomingType gt3 = createGroomingType("3", false);
    GroomingType gt4 = createGroomingType("4", false);

    GroomingPackage gp1 = createGroomingPackage(43.99,
List.of(gt1, gt2));
    GroomingPackage gp5 = createGroomingPackage(44.00,
List.of(gt1, gt2));

    GroomingPackage gp2 = createGroomingPackage(19.99,
List.of(gt3, gt4));
    GroomingPackage gp6 = createGroomingPackage(20.00,
List.of(gt3, gt4));

    GroomingPackage gp3 = createGroomingPackage(29.99,
List.of(gt1, gt3));
    GroomingPackage gp7 = createGroomingPackage(30.0,
List.of(gt1, gt3));

    GroomingPackage gp4 = createGroomingPackage(100.0,
List.of());


    Assertions.assertThrows(WrongPriceException.class,
            () -> algorithm.validatePackageCost(gp1, 10, 20));
    Assertions.assertDoesNotThrow(
            ()-> algorithm.validatePackageCost(gp5, 10, 20));

    Assertions.assertThrows(WrongPriceException.class,
            () -> algorithm.validatePackageCost(gp2, 10, 20));
    Assertions.assertDoesNotThrow(
            ()-> algorithm.validatePackageCost(gp6, 10, 20));

    Assertions.assertThrows(WrongPriceException.class,
            () -> algorithm.validatePackageCost(gp3, 10, 20));
```


```
    Assertions.assertDoesNotThrow(
            () -> algorithm.validatePackageCost(gp7, 10, 20));

    Assertions.assertDoesNotThrow(
            () -> algorithm.validatePackageCost(gp4, 10, 20));

}
```