Resuelto-ControlCheck1-G3-2425.p...

CVV__

Diseño y Pruebas i

3º Grado en Ingeniería Informática - Ingeniería del Software

Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

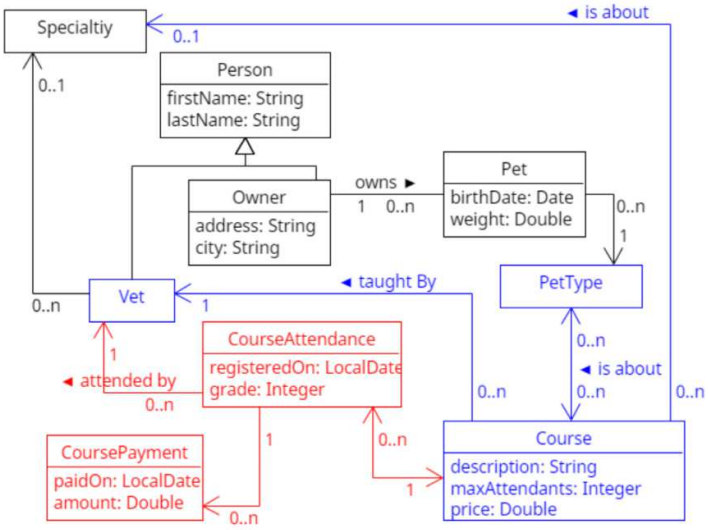
Accede al documento original

Control práctico de DP1 2024-2025 (Control-check 1)

Enunciado

En este ejercicio, añadiremos la funcionalidad de gestión de cursos de especialización sobre mascotas para los veterinarios de las clínicas. Concretamente, se proporciona una clase "Course" que representa los cursos disponibles, relacionados con los tipos de mascota sobre los que versan y el veterinario que actúa como profesor. Además, tendremos las clases "CourseAttendance" y "CoursePayment" que registran la información asociada con los veterinarios que asisten al curso como alumnos y el pago de las tasas correspondientes, dado que el profesor no trabaja gratis.

El diagrama UML que describe las clases y relaciones con las que vamos a trabajar 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 esta prueba debe aceptar la tarea disponible en:

https://classroom.github.com/a/dGR4ggrZ

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 la estructura y nombres proporcionados. Si los modifica probablemente no pueda hacer que pasen las pruebas.

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 funcionen 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 "mvnw 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: Excepto el ejercicio 5 (que dependen del 4) lo ejercicios del examen son independientes y puedes ser resueltos en cualquier orden.

Test 1 – Modificar el servicio de gestión de Asistencia a Cursos para que no permita guardar cursos cuyo instructor no conozca la especialidad a impartir

Modificar el método save del servicio de gestión de cursos (CourseService) de manera que se lance la excepción (UnfeasibleCourseException) en caso de que se intente guardar un curso sobre una especialidad que no tiene asociada el veterinario que lo va a impartir. Por ejemplo, imaginen un curso sobre la especialidad *radiology* que va a ser impartido por *Rafael Ortega* que tiene la especialidad de *surgery* y por tanto no podría impartir dicho curso. El método debe ser transaccional y hacer rollback de la transacción en caso de que se lance dicha excepción. Nótese, que existen relaciones entre la clase Course y la clase Specialty, entre Course y Vet, y entre Vet y Specialty.

Test 2 – Creación de servicios de gestión de las asistencias a cursos y los pagos; y del controlador para devolver los cursos disponibles

Parte 2A: Servicios de Gestión (1 punto):

Modificar las clases "CourseAttendanceService" y "CoursePaymentService", para que sean un servicio Spring de lógica de negocio y proporcionen una implementación a los métodos que permitan, usando una implementación del repositorio correspondiente (CourseAttendanceRepository / CoursePayment-Repository) que será inyectada mediante Spring (usando la anotación "*@Autowired(required=false)"*):

1. Obtener todas las asistencias a cursos/pagos de cursos (*CourseAttendance / CoursePayment*) almacenados en la base de datos como una lista usando el repositorio (método *getAll* en cada uno de los servicios correspondientes).
2. Grabar una asistencias a cursos/pago (*CourseAttendance / CoursePayment*) en la base de datos (método *save*).

Todos estos métodos **deben ser transaccionales**, pero las anotaciones asociadas deben realizarse a nivel de método, no a nivel de clase**.** No modifique por ahora la implementación del resto de métodos de los servicios, ni tampoco los repositorios (hasta que no implemente las entidades en el ejercicio 4 no podrá hacerlo correctamente).

Parte 2B: Controlador para API de Cursos (1 punto):

Crear un método en el controlador "*CourseController"* (alojada en el paquete org.springframework.samples.petclinic.course) que permita devolver todos los cursos existentes. El método debe responder a peticiones tipo GET en la URL:

http://localhost:8080/api/v1/courses

Así mismo debe crear un método para devolver un curso concreto, este método debe responder a peticiones en la url http://localhost:8080/api/v1/courses/X donde X es la Id del curso a obtener, y devolverá los datos del curso correspondiente. En caso de que no exista un curso con el id especificado el sistema debe devolver el código de estado 404 (NOT_FOUND).

Estos endpoint de la API asociados a la gestión de cursos deberían estar accesibles únicamente para usuarios de tipo Vet (que tengan la authority "VET").

Test 3 – Creación de un componente React de listado de cursos

Modificar el componente React proporcionado en el fichero "frontend/src/course/index.js" para que muestre un listado de los cursos disponibles en el sistema. Para ello debe hacer uso de la API lanzando una petición tipo GET contra la URL api/v1/courses.

Este componente debe mostrar los cursos en una tabla (se recomienda usar el componente Table de reactstrap) incluyendo columnas para su nombre, y el conjunto de tipos de mascotas para las que es aplicable el curso. Para esto último, el componente debe mostrar en la celda asociada una lista (preferiblemente no ordenada <ul>) con el nombre de los tipos de mascotas sobre los que versa el curso 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.

Test 4 – Creación de las entidades CourseAttendance y CoursePayment y sus repositorios asociados

Modificar las clases "CourseAttendance" y "CoursePayment" para que sean entidades. Estas clases están alojadas en el paquete "org.springframework.samples.petclinic.course", y deben tener los siguientes atributos y restricciones:

Para ambas clases:

- 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.

Para la clase CourseAttendance:

- El atributo de tipo fecha (LocalDate) llamado "registeredOn", que representa la fecha en que el alumno se registra en el curso, seguirá el formato "dd/MM/yyyy". Este atributo debe ser obligatorio. En la base de datos se almacenará con el nombre de columna "registrationDate".
- El atributo de tipo entero (Integer) llamado "grade", que representa la calificación global del curso. Este atributo es opcional, y debe estar en el rango de valores de 0 a 10, ambos inclusive.

Para la clase CoursePayment:

- El atributo de tipo fecha (LocalDate) llamado "paidOn", que representa la fecha en que el alumno realiza un pago (parcial o total) del precio del curso, seguirá el formato "dd/MM/yyyy". Este atributo debe ser obligatorio. En la base de datos se almacenará con el nombre de columna "paymentDate".
- El atributo de tipo doble (Double) llamado "amount", que representa la cantidad abonada en el pago. Este atributo será obligatorio.

Modificar las interfaces "CourseAttendanceRepository" y "CoursePaymentRepository" alojadas en el mismo paquete para que extiendan a CrudRepository. No olvide especificar sus parámetros de tipo.

Elimine las anotaciones @Transient de los métodos y atributos que las tengan en las entidades creadas anteriormente, así como la del atributo attendants de la clase Course. Se pide crear las siguientes relaciones entre las entidades:

Una relación bidireccional entre “CourseAttendance” y “Course” que represente la que aparece en el diagrama UML, teniendo en cuenta la cardinalidad que tiene.

Además, cree dos relaciones unidireccionales desde “CourseAttendance” hacia “Vet” y hacia “CoursePayment” que expresen las que aparecen en el diagrama UML (mostrado en la primera página de este enunciado), usando los atributos “attendant” y “payments” de la clase “CourseAttendance”, correspondientemente. Debe asegurarse de que las relaciones expresan adecuadamente la cardinalidad que muestra el diagrama UML, por ejemplo, algunos atributos pueden ser nulos puesto que la cardinalidad es 0..n pero otros no, porque su cardinalidad en el extremo navegable de la relación es 1.

Test 5 – Modificación del script de inicialización de la base de datos para incluir dos asistencias a cursos y sus relaciones

Modificar el script de inicialización de la base de datos, para que se creen los siguientes asistencias a curso (CourseAttendance) y pagos de cursos (CoursePayment):

CourseAttendance 1:

- id: 1
- registeredOn: 05-01-2023
- grade: 5

CourseAttendance 2:

- id: 2
- registeredOn: 18-12-2023
- grade: (sin valor aún)

CoursePayment 1:

- id: 1
- paidOn: 06-01-2023
- amount: 150.00

CoursePayment 2:

- id: 2
- paidOn: 10-01-2023
- amount: 150.00

CoursePayment 3:

- id: 3
- paidOn: 19-12-2023
- amount: 50.00

Además, debe modificar este script de inicialización de la base de datos para que:

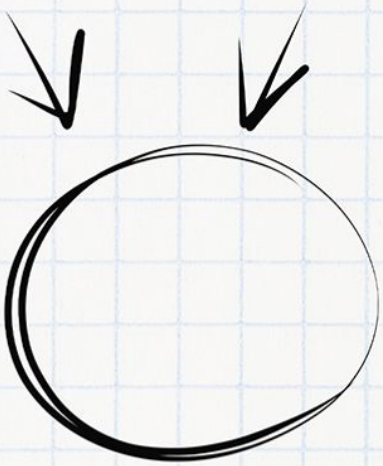
- El CourseAttendance cuyo id es 1 se asocie con el Vet cuyo id es 5
- El CourseAttendance cuyo id es 1 se asocie con el Course cuyo id es 1

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

- El CourseAttendance cuyo id es 2 se asocie con el Vet cuyo id es 3
- El CourseAttendance cuyo id es 2 se asocie con el Course cuyo id es 2
- El CourseAttendance cuyo id es 1 se asocie con los CoursePayment con ids 1 y 2
- El CourseAttendance cuyo id es 2 se asocie con el CoursePayment con id=3

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.

CONTROL CHECK 1 - GRUPO 3 - 24/25

TEST 1

```
@Transactional
public Course save(Course course) throws DataAccessException,
UnfeasibleCourseException {
   if ( course.getCourseSpecialty() != null &&
!course.getCourseSpecialty().equals(course.getTeacher().getSpe
cialty())) {
        throw new UnfeasibleCourseException();}
   return repo.save(course);

}
```

TEST 2 A

```
@Service
public class CoursePaymentService {
   CoursePaymentRepository repo;

   @Autowired
   public CoursePaymentService(CoursePaymentRepository cpr){
       this.repo=cpr;
   }

   @Transactional(readOnly = true)
   public List<CoursePayment> getAll() {
       return repo.findAll();
   }

   @Transactional
   public CoursePayment save(CoursePayment cp) {
       return repo.save(cp);
   }
}
```

```
@Service
public class CourseAttendanceService {
   CourseAttendanceRepository repo;

   @Autowired
   public CourseAttendanceService(CourseAttendanceRepository
car){
```

```
        this.repo=car;
    }
    @Transactional(readOnly = true)
    public List<CourseAttendance> getAll() {
        return repo.findAll();
    }
    @Transactional
    public CourseAttendance save(CourseAttendance ca) throws
DataAccessException, UnfeasibleCourseException {
        return repo.save(ca);
    }
}
```

TEST 2B

```
@RestController
public class CourseController {

    private final CourseService courseService;

    @Autowired
    public CourseController(CourseService courseService){
        this.courseService = courseService;
    }
    @GetMapping("/api/v1/courses")
    @ResponseStatus(HttpStatus.OK)
    public ResponseEntity<List<Course>> getCourse(){
        return new
ResponseEntity<>(courseService.findCourses(), HttpStatus.OK);
    }

    @GetMapping("/api/v1/courses/{courseId}")
    @ResponseStatus(HttpStatus.OK)
    public ResponseEntity<Course> getCourse(@PathVariable
Integer courseId){
        Course course = courseService.findCourseById(courseId);
        if (course == null) throw new
ResourceNotFoundException("Course not found");
        return new ResponseEntity<>(course, HttpStatus.OK);
    }

}
```

SecurityConfiguration.java

```
.requestMatchers(AntPathRequestMatcher.antMatcher("/api/v1/courses/**")).hasAuthority("VET")
```

TEST 4

```
@Getter
@Setter
@Entity
public class CoursePayment extends BaseEntity {
    @DateTimeFormat(pattern = "dd/MM/yyy")
    @NotNull
    @Column(name = "paymentDate")
    LocalDate paidOn;
    @NotNull
    Double amount;
}
```

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

    @DateTimeFormat(pattern = "dd/MM/yyy")
    @NotNull
    @Column(name = "registrationDate")
    LocalDate registeredOn;

    @Min(0)
    @Max(10)
    Integer grade;

    @OneToMany
    Set<CoursePayment> payments;

    @ManyToOne
    @NotNull
    Course course;

    @ManyToOne
    @NotNull
    Vet attendant;
}
```


```java
public interface CourseAttendanceRepository  extends
CrudRepository<CourseAttendance, Integer> {

   Optional<CourseAttendance> findById(Integer i);

   List<CourseAttendance> findAll();

   CourseAttendance save(CourseAttendance any);

}
```

```java
public interface CoursePaymentRepository extends
CrudRepository<CoursePayment, Integer> {

   Optional<CoursePayment> findById(Integer i);

   List<CoursePayment> findAll();

   CoursePayment save(CoursePayment any);

}
```

TEST 5

```sql
INSERT INTO course_attendance(id, registration_date, grade, course_id,
attendant_id) VALUES (1, '2023-01-05', 5, 1, 5), (2, '2023-12-18', null, 2,
3);
INSERT INTO course_payment(id, payment_date, amount) VALUES (1,
'2023-01-06', 150.00), (2, '2023-01-10', 150.00), (3, '2023-12-19', 50.00);
INSERT INTO course_attendance_payments(course_attendance_id, payments_id)
VALUES (1, 1), (1, 2), (2, 3);
```