Informe individual de actividades del proyecto PeliculasOnline - Diciembre

# Datos generales

URL del Repositorio de GitHub: <https://github.com/dantolvil/dp1-2020-grupo-dicembre-2>

Nombre de usuario en GitHub: dantolvil

Rama del usuario en Github: dantolvil

# Participación en el proyecto

## Historias de usuario en las que he participado

## Al tratarse de un grupo compuesto por un único miembro, todas las historias de usuario han sido implementadas por Daniel Toledo Villalba.

## A continuación, se indican algunos ejemplos de estas historias de usuario.

# HU-10 - Alta de películas y HU-12-Búsqueda de películas.

# En total se han implementado 21 historias de usuario.

## Reglas de negocio

## Al tratarse de un grupo compuesto por un único miembro, todas las reglas de negocio han sido desarrolladas por Daniel Toledo Villalba.

## A continuación, se indican algunos ejemplos de estas reglas de negocio.

R3 - No mostrar películas sin clasificación y R6 - Películas duplicadas.

## En total se han aplicado 6 reglas de negocio.

## Funcionalidad implementada

## Al tratarse de un grupo compuesto por un único miembro, todas las clases, interfaces y vistas han sido implementadas por Daniel Toledo Villalba.

## A continuación, se indican algunos ejemplos de la funcionalidad implementada.

## Clases de modelo Java: Cinema, Film, Seat.

## Interfaces de los repositorios: CinemaRoomRepository, TrailerRepository, SubscriberRepository.

## Clases de servicios: DirectorService, RoleService, SessionService.

## Clases de controladores: FimClassificationController, ActorController, RoleController.

## Vistas jsp: cinemas, films, roles.

## En total se han implementado 13 clases de dominio java, servicios, repositorios y controladores por cada entidad sin contar todo lo relacionado a BaseEntity y Authorities. Adicionalmente se han implementado 13 carpetas para las vistas y un total de 52 archivos.jsp, 4 por cada entidad.

## Pruebas implementadas

Pruebas unitarias

## Al tratarse de un grupo compuesto por un único miembro, todas las pruebas unitarias han sido implementadas por Daniel Toledo Villalba.

## A continuación, se indican algunos ejemplos de estas pruebas unitarias.

## Test unitarios asociados a los servicios: CinemaService, SeatService, ActorService.

## Test unitarios asociados a los controladores: CinemaRoomController, SessionController, SubscriberController.

## Test unitarios asociados a los repositorios: TrailerRepository, FilmRepository, FilmRepository.

## Validadores y formatters:

## X métodos anotados con etiqueta @Test

Pruebas de Controlador

## Al tratarse de un grupo compuesto por un único miembro, todas las pruebas positivas y negativas asociadas a los controladores han sido implementadas por Daniel Toledo Villalba.

## A continuación, se indican algunos ejemplos de estas pruebas.

Escenarios positivos: HU1+E1 - Búsqueda de cines, HU2+E1 - Alta de un cine, HU4+E1 - Alta de subscriptor.

Escenarios negativos: H8-E1 - Reserva de asiento rechazada, HU9-E1 - Historial de reservas, HU10-E1 - Alta de una película sin datos.

# Ejemplos de funcionalidades implementadas

## Entidades

## Entidad Film

### Películas(Film).

* RUTA: src\main\java\org\springframework\samples\peliculasOnline\model\Film.java
* Está clase java engloba todo lo necesario para la entidad Película(Film) del proyecto, en ella se detallan los atributos de la clase Film: title, description, date y category.

Además, se incluyen las relaciones de esta entidad con las entidades Subscriber, Director y Actor.

Su función es persistir el modelo de dominio UML a lenguaje java para la entidad película.

@Entity

@Table(name = "films")

**public** **class** Film **extends** BaseEntity{

@Column(nullable = **false**)

**private** String title;

@Column

**private** String description;

@Column

**private** Date releaseDate;

@Column

**private** String category;

//Constructor:

**public** Film(String title, String description, Date releaseDate, String category) {

**super**();

**this**.title = title;

**this**.description = description;

**this**.releaseDate = releaseDate;

**this**.category = category;

}

**public** Film() {

**super**();

}

//Getters and Setters:

**public** String getTitle() {

**return** title;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

**public** String getDescription() {

**return** description;

}

**public** **void** setDescription(String description) {

**this**.description = description;

}

**public** Date getReleaseDate() {

**return** releaseDate;

}

**public** **void** setReleaseDate(Date releaseDate) {

**this**.releaseDate = releaseDate;

}

**public** String getCategory() {

**return** category;

}

**public** **void** setCategory(String category) {

**this**.category = category;

}

//Relationships:

**private** List<Cinema> cinemas;

**private** List<Actor> actors;

**private** Director director;

**private** List<Trailer> trailers;

**public** List<Cinema> getCinemas() {

**return** cinemas;

}

**public** **void** setCinemas(List<Cinema> cinemas) {

**this**.cinemas = cinemas;

}

**public** List<Actor> getActors() {

**return** actors;

}

**public** **void** setActors(List<Actor> actors) {

**this**.actors = actors;

}

**public** Director getDirector() {

**return** director;

}

**public** **void** setDirector(Director director) {

**this**.director = director;

}

**public** List<Trailer> getTrailers() {

**return** trailers;

}

**public** **void** setTrailers(List<Trailer> trailers) {

**this**.trailers = trailers;

}

## Servicio

### Películas(FilmService)

* Ruta: src\main\java\org\springframework\samples\peliculasOnline\service\FilmService.java

· Esta clase corresponde al servicio de la entidad Film, en ella se incluyen los métodos Crud y demás métodos que serán nutridos por el repositorio asociado desde el que se accederá a la base de datos. Estos métodos se usarán en el controlador de la entidad Film.

* Se han incluido comentarios en el código para detallar los diferentes métodos.
* Se ha usado @Autowired para inyectar dependencias con otras entidades dentro de Spring.

@Service

**public** **class** FilmService {

**private** FilmRepository filmRepository;

@Autowired

**public** FilmService(FilmRepository filmRepository) {

**this**.filmRepository = filmRepository;

}

//Encontrar una película según su tipo

@Transactional(readOnly = **true**)

**public** Collection<Film> findFilmTypes() **throws** DataAccessException {

**return** filmRepository.findFilmTypes();

}

@Transactional

**public** **void** save(Film film) **throws** DataAccessException {

filmRepository.save(film);

}

@Transactional(readOnly = **true**)

**public** Film findFilmById(**int** id) **throws** DataAccessException {

**return** filmRepository.findById(id);

}

//Guardar una película @Transactional

**public** **void** saveFilm(Film film) **throws** DataAccessException {

filmRepository.save(film);

}

## Controlador

### FILMCONTROLLER

RUTA:

src\main\java\org\springframework\samples\peliculasOnline\web\FilmController.java

* En el controlador de la entidad Film se han incluido una serie de métodos para administrar las peticiones HTTP y realizar las redirecciones a las vistas correspondientes.

Mediante @Autowired se realizan las inyecciones de dependencia entre entidades.

* Se han incluido comentarios en el código para detallar los diferentes métodos.

@Controller

**public** **class** FilmController<filmService> {

**private** **static** **final** String ***VIEWS\_FILM\_CREATE\_OR\_UPDATE\_FORM*** = "films/createOrUpdatefilmForm";

**private** **final** FilmService filmService;

@Autowired

**public** FilmController(FilmService filmService) {

**this**.filmService = filmService;

}

@InitBinder

**public** **void** setAllowedFields(WebDataBinder dataBinder) {

dataBinder.setDisallowedFields("id");

}

//HTTP(GET). Mediante GET se recuperan los datos de la entidad Film.

@GetMapping(value = "/films/new")

**public** String initCreationForm(Map<String, Object> model) {

FilmClassification film = **new** FilmClassification(**null**, **null**);

model.put("film", film);

**return** ***VIEWS\_FILM\_CREATE\_OR\_UPDATE\_FORM***;

}

//HTTP(POST). Si no se producen errores en el modelo, se guardan los datos en el servicio usando el método save().

// Por el contrario, si se produce un error se nos redirecciona a la vista con el formulario para la creación.

@PostMapping(value = "/films/new")

**public** String processCreationForm(@Valid Film film, BindingResult result) {

**if** (result.hasErrors()) {

**return** ***VIEWS\_FILM\_CREATE\_OR\_UPDATE\_FORM***;

}

**else** {

**this**.filmService.save(film);

**return** "redirect:/films/" + film.getId();

}

}

@GetMapping(value = "/films/find")

**public** String initFindForm(Map<String, Object> model) {

model.put("film", **new** FilmClassification(**null**, **null**));

**return** "films/findfilms";

}

//Mediante el uso del servicio FilmService se obtiene la película que va a

//ser modificada a través de su Id y se redirecciona a la vista de edición.

@GetMapping(value = "/films/{filmId}/edit")

**public** String initUpdateOwnerForm(@PathVariable("filmId") **int** filmId, Model model) {

Film film = **this**.filmService.findFilmById(filmId);

model.addAttribute(film);

**return** ***VIEWS\_FILM\_CREATE\_OR\_UPDATE\_FORM***;

}

//Al realizar la edición la película, se comprueba si presenta errores,

//si todo ha ido bien se redirecciona a la vista de los detalles de la película. Por el contrario, si se encuentran errores se devolverá a la vista anterior.

@PostMapping(value = "/films/{filmId}/edit")

**public** String processUpdateOwnerForm(@Valid Film film, BindingResult result,

@PathVariable("filmId") **int** filmId) {

**if** (result.hasErrors()) {

**return** ***VIEWS\_FILM\_CREATE\_OR\_UPDATE\_FORM***;

}

**else** {

film.setId(filmId);

**this**.filmService.save(film);

**return** "redirect:/films/{filmId}";

}

}

//Se muestra la película que se obtiene por url a través del filmId.

@GetMapping("/films/filmId}")

**public** ModelAndView showfilm(@PathVariable("filmId") **int** filmId) {

ModelAndView mav = **new** ModelAndView("films/filmDetails");

mav.addObject(**this**.filmService.findFilmById(filmId));

**return** mav;

}

## Repositorio

### FILMREPOSITORY

RUTA:

src\main\java\org\springframework\samples\peliculasOnline\repository\FilmRepository.java

* A través del repositorio de película(filmRepository) se obtienen los registros de la base de datos mediante consultas JPQL.
* Se han incluido comentarios en el código para detallar la función de cada consulta.

**public** **interface** FilmRepository **extends** Repository<Film, Integer>{

//Seleccionar todas las películas del sistema

@Query("select f from Film f")

Collection<Film> findAll();

Film findById(**int** id) **throws** DataAccessException;

//Seleccionar una película con tipo igual al que se pasa por parámetro

@Query("select f from Film f where f.type like %?1")

**public** Collection<Film> findFilmTypes(String Type);

**void** save(Film film);

}

## Validador y anotación asociada.

## Validador para la creación de contraseña

## Se ha implementado una validación para la creación de las contraseñas en el sistema como se indica a continuación

## @Pattern(regexp = "^(?=.[a-z])(?=.[A-Z])(?=.\*\d)[a-zA-Z\d]{8,}$", message=" La contraseña debe incluir como mínimo 8 caracteres, una letra mayúscula, una letra minúscula y al menos un número")

# Ejemplos de pruebas implementadas

## Pruebas unitarias

**RESERVAASIENTOSERVICETEST**

**Ruta:** src\test\java\org\springframework\samples\peliculasOnline\service\ReservaAsientoServiceTests.java

* Se quiere comprobar que se puede encontrar una reserva de asiento asociada a una sala de una y una sesión por su nombre.

@Test

**void** shouldFindReservaAsientoByName() { Seat asiento = **new** Seat(); asiento.setSesion(1); asiento.setCinemaRoom(12); asiento.setCinema("Cinesa Triana");

LocalDate fechaReserva=LocalDate.*of*(2021, 10, 26); asiento.setFechaReserva(fechaReserva);

asiento.setAsiento(5);

ReservaAsiento reservaAsiento =

**new** ReservaAsiento (); reservaAsiento.setFechaReserva(LocalDate.*now*();

reservaAsiento.setNumeroTarjeta("371569635456431"); reservaAsiento.setCvc("245"); reservaAsiento.setPrecioFinal(7.5);

Subscriber subscriptor = **new** User(); subscriptor.setUsername("dantolvil"); reservaAsiento.setUser(subscriptor);

* **Arrange/Fixture**: Se crea una nueva reserva de asiento, para ello se debe crear un asiento y un subscriptor.

=================================ARRANGE/FIXTURE==================================

**this**.reservaAsientoService.saveReservaAsiento(reservaAsiento);

Collection<ReservaAsiento> reservaAsientos =

**this**.reservaAsientoService.buscarReservaAsiento("dantolvil");

* **Act**: Una vez se han realizado todas las operaciones, se selecciona la opción de guardar la reserva de asiento.

Se crea un conjunto en el que se añaden las reservas de asiento realizadas con el subscriptor de nombre “dantolvil”.

======================================ACT=========================================

*assertThat*(reservaAsientos.size()).isEqualTo(1);

}

* **Assert**: Se comprueba que el conjunto de las reservas tiene tamaño 1, en este caso sería la reserva del subscriptor “dantolvil”.

======================================ASSERT======================================

Pruebas unitarias parametrizadas

Pruebas de controlador **UserControllerTests.**

### Ruta:

src\test\java\org\springframework\samples\peiculasOnline\web\UserControllerTests.java

### ARRANGE/FIXTURE

@WebMvcTest(controllers=UserController.**class**,

excludeFilters = @ComponentScan.Filter(type = FilterType.***ASSIGNABLE\_TYPE***, classes

= WebSecurityConfigurer.**class**), excludeAutoConfiguration= SecurityConfiguration.**class**) **public class** UserControllerTests {

**private static final** String ***TEST\_USER\_ID*** = "dantolvil";

@Autowired

**private** UserController userController;

@MockBean

**private** UserService userService;

@MockBean

**private** SeatService seatService;

@Autowired

**private** MockMvc mockMvc;

**private** Subscriber subscriber;

@BeforeEach

**void** setup() {

subscriber = **new** Subscriber(); subscriber.setUsername("dantolvil"); subscriber.setPassword("US23"); subscriber.setPhone("955134297"); subscriber.setDni("250603212X");

*given*(**this**.userService.findByUsername(***TEST\_USER\_ID***)).willReturn(user);

}

**TEST NEGATIVO:**

@Test

**void** testProcessCreationFormHasErrors() **throws** Exception { mockMvc.perform(*post*("/subscribers/new").param("username", "dantolvil")

.with(*csrf*())

.param("password", "US23")

.param("phone", "")

.param("dni", "250603212X"))

· Act: Para esta prueba negativa se ha usado el usuario ya creado en el apartado anterior sin incluir el número de teléfono(phone). Al no rellenar este campo el sistema no nos va a permitir crear el subscriptor, puesto que este valor no puede estar vacío.

==============================ACT=================================================

.andExpect(*status*().isOk())

.andExpect(*model*().attributeHasErrors("subscriber"))

.andExpect(*model*().attributeHasFieldErrors("subscriber","phone"))

.andExpect(*view*().name("users/createOrUpdateSubscriberForm"));

}

· Assert: Aquí le indicamos que el modelo user tiene un campo con error, en este caso teléfono para que no nos cree el nuevo usuario y nos reenvíe al mismo formulario para indicar mediante el mensaje de una validación sintáctica que rellene el campo.

============================ASSERT================================================

**TEST POSITIVO.**

@Test

**void** testProcessUpdateFormSuccess() **throws** Exception { mockMvc.perform(*post*("/subscribers/{username}/edit", ***TEST\_USER\_ID***)

.with(*csrf*())

.param("username", "dantolvil")

.param("password", "US23")

.param("phone", "674321345")

.param("dni", "250603212X"))

·Act: Utilizamos el usuario ya creado anteriormente con todos sus parámetros válidos para comprobar que se crea correctamente.

===============================ACT================================================

.andExpect(*status*().is3xxRedirection())

.andExpect(*view*().name("redirect:/users/{username}"));

}

·Assert: Comprobamos que lo crea y nos redirige a la página del usuario.

=================================ASSERT===========================================

# Principales problemas encontrados

### Borrado de cines con reserva de asientos para la sesión.

# Cuando se necesita eliminar un cine que presenta asientos reservados para una sala de una sesión asociada se generan conflictos si un subscriptor ya había reservado ese asiento en la sala(CinemaRoom), se ha optado por ocultarlo de la lista de cines este cien y solo mostrar aquellos cines en los cuales se puedan realizar reservas de asientos asociados a las salas disponibles y la sesión asociada. De esta forma aquellos subscriptores que hubiesen reservado un asiento en una sala de una sesión asociada a un cine que se va a eliminar de la plataforma conservarán su reserva.

# Otros comentarios

# - Sincronización IDE Eclipse y Github:

# Al usar el IDE eclipse he encontrado varios problemas para configurar la sincronización del repositorio local de Eclipse con el repositorio de Github. Tras varios intentos con mal resultado y dado que debía seguir avanzando en la implementación del proyecto para la entrega, decidí realizar todos los commits y subir todos los documentos directamente en la herramienta Github sin usar el conector de Eclipse.

# 

# - Problema con repositorio Github:

Al Inicio del desarrollo del proyecto se creó el repositorio “https://github.com/dantolvil91/dp1-2020-grupo-dicembre-2” en Github. Tras unas semanas usando ese repositorio, el usuario de Github que estaba usando “dantolvil91” se bloqueó. Por este motivo se ha usado a partir de ese momento el usuario de Github “dantolvil” y se ha creado un nuevo repositorio “https://github.com/dantolvil/dp1-2020-grupo-dicembre-2” a partir del primero mencionado para seguir con la implementación del proyecto.