Informe individual de actividades del proyecto Parchis&Oca – Septiembre

# Datos generales

URL del Repositorio de GitHub: <https://github.com/dantolvil/dp1-2021-2022-g1-septiembre>

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 y HU-12.

# En total se han implementado **X** 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 - y R5 -**

## En total se han aplicado 5 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: Game, GameAction, GameBoard.

## Interfaces de los repositorios: GameRepository, GamePieceRepository, BoardFieldRepository.

## Clases de servicios: GamerService, PlayerService, TurnService.

## Clases de controladores: GameController, OcaController, ParchisController.

## Vistas jsp con las siguientes carpetas: game, admins, players.

## En total se han implementado 10 clases de dominio java, 10 servicios, 10 repositorios y 7 controladores para las diferentes entidades sin contar todo lo relacionado a BaseEntity y Authorities. Adicionalmente se han implementado 6 archivos.jsp, 3 para la carpeta admins, 1 para game y 2 para players. Por último, se han implementado dos vistas adicionales que son Welcome y Exception para mostrar las excepciones y el home de la aplicación web.

## 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, HU2+E1, HU4+E1.

Escenarios negativos: H8-E1, HU9-E1, HU10-E1.

# Ejemplos de funcionalidades implementadas

## Entidades

## Entidad Player

### Player

* RUTA:

src\main\java\org\springframework\samples\parchis\_oca\player\Player.java

* Está clase java engloba todo lo necesario para la entidad Player del proyecto, en ella se detallan los atributos de la clase java Player: email, además hereda los atributos de la clase Peson, firstName y lastName.

Además, se incluyen las relaciones de esta entidad con la entidad Turn y GamePiece.

Su función es persistir el modelo de dominio UML a lenguaje java para la entidad Player y definir los atributos y relaciones de esta entidad.

@Getter

@Setter

@Entity

@Table(name = "players")

**public** **class** Player **extends** Person {

//Attributes

@Column(name = "email")

@NotBlank

**private** String email;

//Relationships

@OneToMany(cascade = CascadeType.***ALL***)

**private** List<Turn> turn;

@OneToOne(cascade = CascadeType.***ALL***)

@JoinColumn(name = "gamePiece")

**private** GamePiece gamePiece;

**public** **boolean** checkAlreadyCreatedGames() {

**return** **false**;

}

}

## Servicio

### PlayerService

* Ruta: src\main\java\org\springframework\samples\parchis\_oca\player\PlayerService.java

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

* 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** PlayerService {

**private** PlayerRepository playerRepository;

@Autowired

**public** PlayerService(PlayerRepository playerRepository) {

**this**.playerRepository = playerRepository;

}

//Recuperar el jugador actual

**public** Optional <Player> getCurrentPlayer() {

Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();

String currentPrincipalName = authentication.getName();

**return** findPlayer(currentPrincipalName);

}

**public** Optional <Player> findPlayer(String player) {

**return** playerRepository.findById(player);

}

//Guardar y crear un jugador @Transactional

@Transactional

**public** **void** savePlayer(Player player) **throws** DataAccessException {

//creating player

playerRepository.save(player);

}

}

## Controlador

### PlayerController

RUTA:

src\main\java\org\springframework\samples\parchis\_oca\PlayerController.java

* En el controlador de la entidad Player 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** PlayerController {

**private** **static** **final** String ***VIEWS\_PLAYER\_CREATE\_OR\_UPDATE\_FORM*** = "players/createOrUpdateOwnerForm";

**private** **static** **final** String ***VIEWS\_PLAYERS\_CREATE\_OR\_UPDATE\_FORM*** = **null**;

@Autowired

**private** PlayerService playerService;

@InitBinder

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

dataBinder.setDisallowedFields("id");

}

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

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

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

Player player = **new** Player();

model.put("player", player);

**return** ***VIEWS\_PLAYER\_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 = "/players/new")

**public** String processCreationForm(@Valid Player player, BindingResult result) {

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

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

}

**else** {

**this**.playerService.savePlayer(player);

**return** "redirect:/players/" + player.getId();

}

}

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

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

model.put("player", **new** Player());

**return** "players/findPlayers";

}

//Mediante el uso del servicio PlayerService se obtiene el jugador que va a

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

@GetMapping(value = "/players/{playerId}/edit")

**public** String initUpdatePlayerForm(@PathVariable("playerId") String playerId, Model model) {

Optional<Player> player = **this**.playerService.findPlayer(playerId);

model.addAttribute(player);

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

}

//Al realizar la edición del jugador se comprueba si presenta errores,

//si todo ha ido bien se re direcciona a la vista de los detalles del jugador.

//Por el contrario, si se encuentran errores se devolverá a la vista anterior.

@PostMapping(value = "/players/{playerId}/edit")

**public** String processUpdatePlayerForm(@Valid Player player, BindingResult result, @PathVariable("playerId") **int** playerId) {

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

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

}

**else** {

player.setId(playerId);

**this**.playerService.savePlayer(player);

**return** "redirect:/players/{playerId}";

}

}

//Se muestra el jugador que se obtiene por url a través del playerId.

@GetMapping("/players/{playerId}")

**public** ModelAndView showOwner(@PathVariable("playerId") String playerId) {

ModelAndView mav = **new** ModelAndView("players/playerDetails");

mav.addObject(**this**.playerService.findPlayer(playerId));

**return** mav;

}

}

## Repositorio

### PlayerRepository

RUTA:

src\main\java\org\springframework\samples\parchis\_oca\player\PlayerRepository.java

* A través del repositorio de jugador(PlayerRepository) se obtienen los registros de la base de datos mediante consultas JPQL.
* En este caso no se ha implementado ninguna consulta JPQL para la entidad Player.

**public** **interface** PlayerRepository **extends** Repository<Player, Integer> {

**void** save(Player owner) **throws** DataAccessException;

Optional<Player> findById(String player);

}

## 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

**ReservaAsientoServiceTests**

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** Subscriber(); subscriptor.setUsername("dantolvil"); reservaAsiento.setSubscriber(subscriptor);

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

**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”.

*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”.

Pruebas unitarias parametrizadas

Pruebas de controlador

**SubscriberControllerTests**

### RUTA:

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

* **Arrange/Fixture:**

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

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

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

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

@Autowired

**private** SubscriberController subscriberController;

@MockBean

**private** SubscriberService subscriberService;

@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**.subscriberService.findByUsername(***TEST\_ SUBSCRIBER \_ID***)).willReturn(subscriber);

}

**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:** Se ha usado el subscriptor 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.

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

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

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

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

}

· **Assert:** Se indica que el subscriptor tiene un campo con error, en este caso el campo phone. Por ello no se crea el subscriber y se redirecciona al formulario de creación mostrando un mensaje de error indicando que falta informar este campo.

**Test Positivo:**

@Test

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

.with(*csrf*())

.param("username", "dantolvil")

.param("password", "US23")

.param("phone", "674321345")

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

·**Act:** Se usa el subscriptor que se ha creado anteriormente con los parámetros válidos para comprobar que se fue creado correctamente.

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

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

}

·**Assert:** Se comprueba que se crea el subscriptor y se redirige a la página del subscriptor.

# 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.

# 

### Otros problemas