***Informe Individual Ingeniería de Software***

***Airelys Collazo Perez C-312 Movie++***

**1-Modelacion de la base de datos.**

Alejandro Escobar y yo fuimos los encargados de modelar la base de datos, este es un trabajo bastante difícil pues hay que tener una idea de lo que se va a necesitar en el futuro para el desarrollo de la aplicación, es algo complicado virar hacia atrás en un proyecto por una base de datos mal modelada.

Después de muchas ideas y mucho pensar llegamos a una que fue la base de nuestro proyecto, esta, por supuesto, en el camino sufrió pequeños cambios, pero estos fueron mínimos, añadir variables, eliminarlas, etc.

Con la idea de Alejandro de que un horario se conforma por una entidad Película, una entidad Sala y una entidad Compra yo llegue al modelo final de la base de datos, que es el que podemos ver en el informe técnico de nuestra aplicación.

Aquí dejo el código de Horary que es una de los más importantes.

public class Horary : BaseEntity

{

public int MovieId { get; set; }

public int Movie\_LocalId { get; set; }

[Required, Display(Name = "Fecha", Description = "Fecha", Prompt = "Fecha")]

[DataType(DataType.Date)]

public DateTime Date { get; set; }

[Required, Display(Name = "Hora", Description = "Hora", Prompt = "Hora")]

[DataType(DataType.Time)]

public DateTime Time { get; set; }

public int ReservedTickets { get; set; }

public int Price { get; set; }

public int PriceInPoints { get; set; }

public int PointsForBuying { get; set; }

public virtual Movie Movie { get; set; }

public virtual Movie\_Local Movie\_Local { get; set; }

public virtual ICollection<Buy\_Ticket> Buys\_Tickets { get; set; }

}

**2-Sistema de compra.**

Para explicar el sistema de compra de la aplicación voy ir pasando por cada método de este controlador.

Lo primero que se va a verificar si hay otras compras realizándose (ya sea por taquilla o un usuario normal) o que caducaron para tener actualizado constantemente los asientos reservados.

Nuestra compra será un BuyTicket por eso desde el principio del controlador vamos a irlo creando. Cuando se seleccionó el horario que se va a comprar ya le vamos a añadir a nuestro BuyTicket el id de este horario, el id del usuario que está realizando la compra, la fecha en la que se está realizando y que la comprar aún no ha sido completada.

public IActionResult Index(int id)

{

\_BuyEntranceService.TenMinutesCheck(seconds);

var \_pendingTicket = \_BuyEntranceService.GetUserPendingTicket(\_UserManager.GetUserId(User), seconds);

if (\_pendingTicket != null)

{

\_ticket = \_pendingTicket;

if (User.IsInRole(Roles.Roles.Ticket\_Agent.ToString()))

return RedirectToAction(nameof(PayInTicketOffice));

return RedirectToAction(nameof(Buy));

}

\_ticket = new Buy\_Ticket()

{

HoraryId = id,

ApplicationUserId = \_UserManager.GetUserId(User),

Date = DateTime.Now,

PayCompleted = false

};

ViewData["Total"] = 0;

foreach (var item in \_UserTypeService.GetAllUserTypes())

{

ViewData[item.Type] = 0;

}

return View(\_UserTypeService.GetAllUserTypes());

}

Seguido a esto el usuario o el taquillero selecciona la cantidad de entradas por los diferentes tipos de usuarios con los distintos descuentos. Con esto nosotros guardaremos la cantidad de entradas que se piden y calcularemos la cantidad total a pagar, también verificamos que exista capacidad para la cantidad de entradas que se piden.

public IActionResult Index(IFormCollection form)

{

var horary = \_HoraryService.GetAllHoraries()

.Include(h => h.Movie)

.Include(h => h.Movie\_Local)

.FirstOrDefault(m => m.Id == \_ticket.HoraryId);

int Total = 0;

int totalOfEnt = 0;

foreach (var item in \_UserTypeService.GetAllUserTypes())

{

int noEnt = int.Parse(form[item.Type].ToString());

if (noEnt != 0)

{

\_ticket.VoucherUserTypes += item.Type + ": " + noEnt.ToString() + " entrances" + '\n';

}

totalOfEnt += noEnt;

Total += noEnt \* (horary.Price - item.Discount);

ViewData[item.Type] = noEnt;

}

\_ticket.Payment = Total;

\_ticket.NumberOfEntrance = totalOfEnt;

if (totalOfEnt > horary.Movie\_Local.Capacity - horary.ReservedTickets)

{

ViewData["Error"] = "NO CAPACITY";

return View(\_UserTypeService.GetAllUserTypes());

}

if (Total == 0)

return View(\_UserTypeService.GetAllUserTypes());

return RedirectToAction(nameof(SelectSeats));

}

A continuación, se mandará a la vista los asientos a seleccionar actualizados, los reservados dirán Reserved y los demás se podrán seleccionar,y además, se mostrará el total a pagar ya calculado.

public IActionResult SelectSeats()

{

ViewData["Total"] = \_ticket.Payment;

\_BuyEntranceService.TenMinutesCheck(seconds);

var horary = \_HoraryService.GetAllHoraries()

.Include(h => h.Movie)

.Include(h => h.Movie\_Local)

.FirstOrDefault(m => m.Id == \_ticket.HoraryId);

AllSeats = new bool[horary.Movie\_Local.Rows, horary.Movie\_Local.Columns];

AllSeats = \_BuyEntranceService.ShowMovieLocalSeats(AllSeats, horary.Id);

ViewData["NoEnt"] = \_ticket.NumberOfEntrance;

return View(AllSeats);

}

Luego, los asientos seleccionados se reservarán y los añadiremos a nuestro BuyTicket.

[HttpPost]

public IActionResult SelectSeats(IFormCollection form)

{

var horary = \_HoraryService.GetAllHoraries()

.Include(h => h.Movie)

.Include(h => h.Movie\_Local)

.FirstOrDefault(m => m.Id == \_ticket.HoraryId);

if (form.Count() - 3 != \_ticket.NumberOfEntrance)

{

ViewData["NoEnt"] = \_ticket.NumberOfEntrance;

ViewData["Error"] = "Number of seats dont match with number of entrances.";

return View(AllSeats);

}

\_ticket.Date = DateTime.Now;

\_BuyTicketService.InsertBuyTicket(\_ticket);

\_ticket = \_BuyEntranceService.InsertReservedSeats(\_ticket, form);

horary.ReservedTickets += \_ticket.NumberOfEntrance.Value;

\_HoraryService.UpdateHorary(horary);

if (form.ContainsKey("Buy"))

return RedirectToAction(nameof(Buy));

return RedirectToAction(nameof(PayInTicketOffice));

}

Ahora se pasa al método pagar, pero el método de pago depende de si es usuario actual es uno normal o es un taquillero. Veamos primero el caso de un usuario normal.

El usuario tendrá dos cuadros de texto donde pondrá su tarjeta de crédito y su código, pero también si es un socio y los puntos que tiene le alcanza para pagar las entradas le saldrá la opción PayWithPoints donde pagaría con sus puntos (esto lo guardamos en TempData para pasárselo a la vista y muestre la opción si se cumple lo antes dicho).

public IActionResult Buy()

{

\_ticket = \_BuyTicketService.GetAllBuy\_Tickets()

.Include(t => t.Horary)

.Include(t => t.ApplicationUser)

.FirstOrDefault(t => t.Id == \_ticket.Id);

var horary = \_ticket.Horary;

var user = \_ticket.ApplicationUser;

int time = \_BuyEntranceService.GetTime(\_ticket.Date, seconds);

if (time < 0) return RedirectToAction(nameof(Cancel));

ViewData["Time"] = time;

if (user.Code != null && user.Puntuation >= horary.PriceInPoints \* \_ticket.NumberOfEntrance)

{

TempData["IsMember"] = true;

}

return View();

}

Ahora nosotros pasamos a verificar si los datos de la tarjeta son correctos (intentando simular algo real), verificamos si la tarjeta tiene 16 dígitos y si el código es de 4 dígitos. Esta tarjeta que el usuario introduce nosotros la guardamos en la base de datos para después si el usuario tiene que cancelar su compra se le reintegra el dinero a dicha tarjeta(el dinero que tiene la tarjeta es un número aleatorio para que se dé el caso que no tenga dinero, incluso guardamos el código para si después introduce otro código que le salga código incorrecto, sabemos que esto en la vida real no pasa pues los códigos de las tarjetas se pueden cambiar pero solo queríamos simular los posibles errores que puede dar). Por último se crearía el comprobante de la compra y se lo añadimos a nuestro BuyTicket para que todo quede registrado.

[Authorize(Roles = "Basic\_User,Admin,Manager")]

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Buy(long Number, int Code)

{

int time = \_BuyEntranceService.GetTime(\_ticket.Date, seconds);

if (time < 0)

{

TempData["Error"] = "Your buy expired. Please try again";

return RedirectToAction(nameof(Cancel));

}

ViewData["Time"] = time;

if (Number.ToString().Length != 16)

{

ViewData["Error"] = "Invalid Operation: Credit Card must have 16 numbers";

return View();

}

\_ticket = \_BuyTicketService.GetAllBuy\_Tickets()

.Include(b => b.Reserved\_Seats)

.Include(b => b.Horary)

.Include(b => b.ApplicationUser)

.FirstOrDefault(t => t.Id == \_ticket.Id);

var horary = \_ticket.Horary;

var user = \_ticket.ApplicationUser;

CreditCard \_creditCard = new CreditCard()

{

ApplicationUserId = user.Id,

Number = Number,

Code = Code,

Money = new Random().Next(0, 10000)

};

var exists = \_CreditCardService.GetByNumber(Number);

if (exists == null)

{

\_CreditCardService.InsertCreditCard(\_creditCard);

}

else

{

\_creditCard = exists;

if (exists.Code != Code)

{

ViewData["Error"] = "Incorrect CreditCard code";

return View();

}

}

if (\_creditCard.Money < \_ticket.Payment)

{

ViewData["Error"] = "NO CASH";

return View();

}

else

{

\_creditCard.Money -= \_ticket.Payment.Value;

\_CreditCardService.UpdateCreditCardMoney(\_creditCard);

if (user.Code != null)

{

user.Puntuation += horary.PointsForBuying \* \_ticket.NumberOfEntrance;

\_UserService.UpdateUser(user);

}

\_ticket.Date = DateTime.Now;

\_ticket.Voucher = "Date: " + \_ticket.Date.ToString() + '\n' +

"User: " + user.FirstName + " " + user.LastName + '\n' +

"Number of entrances: " + \_ticket.NumberOfEntrance.ToString() + '\n' +

"User Types: " + '\n' + \_ticket.VoucherUserTypes +

"Price : " + \_ticket.Payment.ToString() + '\n';

\_ticket.PayWithPoints = false;

\_ticket.CreditCardId = \_creditCard.Id;

\_ticket.PayCompleted = true;

string \_code = \_ticket.Id.ToString();

\_ticket.Voucher += "Pay Code: " + user.Id.Substring(0, 12 - \_code.Length) + \_code + '\n' +

"Seats :" + '\n' + \_ticket.VoucherSeats;

\_BuyTicketService.UpdateBuyTicket(\_ticket);

TempData["Success"] = "Successful Pay";

}

return RedirectToAction(nameof(ViewVoucher));

}

Lo antes dichos es en el caso que pague con tarjeta de crédito, ahora veamos si paga con puntos. En el comprobante aparecería que pagó con puntos y se hacen las actualizaciones necesarias como por ejemplo se actualiza la puntuación del socio.

public IActionResult BuyByPoints()

{

if (DateTime.Now.Subtract(\_ticket.Date).TotalSeconds > seconds)

{

TempData["Error"] = "Your buy expired. Please try again";

return RedirectToAction(nameof(Cancel));

}

\_ticket = \_BuyTicketService.GetAllBuy\_Tickets()

.Include(b => b.Reserved\_Seats)

.Include(b => b.Horary)

.Include(b => b.ApplicationUser)

.FirstOrDefault(t => t.Id == \_ticket.Id);

var horary = \_ticket.Horary;

var user = \_ticket.ApplicationUser;

\_ticket.Date = DateTime.Now;

\_ticket.Payment = \_ticket.NumberOfEntrance \* horary.PriceInPoints;

\_ticket.Voucher = "Date: " + \_ticket.Date.ToString() + '\n' +

"User: " + user.FirstName + " " + user.LastName + '\n' +

"Number of entrances: " + \_ticket.NumberOfEntrance.ToString() + '\n' +

"User Types: " + '\n' + \_ticket.VoucherUserTypes +

"Points : " + \_ticket.Payment.ToString() + '\n';

\_ticket.PayWithPoints = true;

\_ticket.PayCompleted = true;

string \_code = \_ticket.Id.ToString();

\_ticket.Voucher += "Pay Code: " + user.Code.ToString().Substring(0, 12 - \_code.Length) + \_code + '\n' +

"Seats :" + '\n' + \_ticket.VoucherSeats;

\_BuyTicketService.UpdateBuyTicket(\_ticket);

user.Puntuation -= \_ticket.Payment;

\_UserService.UpdateUser(user);

TempData["Success"] = "Success: " + user.Puntuation + "Points left";

return RedirectToAction(nameof(ViewVoucher));

}

Cabe resaltar, que en los dos métodos de pago vistos siempre se comprueba que no hayan pasado 10 minutos pues solo se tiene 10 minutos para efectuarla, si no, esta se cancela.

Ahora veamos el caso que el usuario es un taquillero, cuando este pasa a efectuar el pago tiene dos campos posibles a llenar, en uno se inserta el efectivo que le entrega el cliente y en el otro el código del cliente si es socio del cine, si solo se pone el efectivo se paga con este y se actualiza el comprobante con lo que pidió el cliente y el cambio que lleva, si se llena solo el código del socio, si tiene suficientes puntos la compra se efectúa con puntos, ahora si el taquillero introduce efectivo y código se pagaría con efectivo pero se le sumarían puntos al socio por comprar. Es importante, que al igual que el usuario normal tiene 10 minutos para realizar la compra.

[Authorize(Roles = "Ticket\_Agent")]

[HttpPost]

public IActionResult PayInTicketOffice(int cash, long code)

{

int time = \_BuyEntranceService.GetTime(\_ticket.Date, seconds);

if (time < 0)

{

return RedirectToAction(nameof(Cancel));

}

ViewData["Time"] = time;

\_ticket = \_BuyTicketService.GetAllBuy\_Tickets()

.Include(b => b.Reserved\_Seats)

.Include(b => b.Horary)

.Include(b => b.ApplicationUser)

.FirstOrDefault(t => t.Id == \_ticket.Id);

var \_ticketAgent = \_ticket.ApplicationUser;

\_ticket.Date = DateTime.Now;

\_ticket.Voucher = "Date: " + \_ticket.Date.ToString() + '\n' +

"Ticket Agent: " + \_ticketAgent.FirstName + " " + \_ticketAgent.LastName + '\n' +

"Number of entrances: " + \_ticket.NumberOfEntrance.ToString() + '\n' +

"User Types: " + '\n' + \_ticket.VoucherUserTypes;

\_BuyTicketService.UpdateBuyTicket(\_ticket);

if (cash != 0)

{

int change = cash - \_ticket.Payment.Value;

if (change < 0)

{

ViewData["msg"] = "Cash not enough.";

return View();

}

\_ticket.Voucher += "Price: " + \_ticket.Payment.ToString() + '\n'

+ "Change: " + change.ToString() + '\n' + "Seats: " + '\n' + \_ticket.VoucherSeats;

\_ticket.PayWithPoints = false;

\_ticket.PayCompleted = true;

\_BuyTicketService.UpdateBuyTicket(\_ticket);

if (code != 0)

{

var user = \_UserService.GetAllUsers()

.FirstOrDefault(m => m.Code != null && m.Code.Value == code);

if (user == null)

{

ViewData["msg"] = "Code doesnt belong to any user.";

return View();

}

user.Puntuation += \_ticket.NumberOfEntrance \* \_ticket.Horary.PointsForBuying;

\_UserService.UpdateUser(user);

}

return RedirectToAction(nameof(ViewVoucher));

}

else if (code != 0)

{

var user = \_UserService.GetAllUsers()

.FirstOrDefault(m => m.Code != null && m.Code.Value == code);

if (user == null)

{

ViewData["msg"] = "Code doesnt belong to any user.";

return View();

}

else if (user.Puntuation < \_ticket.NumberOfEntrance \* \_ticket.Horary.PriceInPoints)

{

ViewData["msg"] = "User have no enough points to pay";

return View();

}

\_ticket.Payment = \_ticket.NumberOfEntrance \* \_ticket.Horary.PriceInPoints;

user.Puntuation -= \_ticket.Payment;

\_UserService.UpdateUser(user);

\_ticket.Voucher += "Points: " + \_ticket.Payment.ToString() + '\n' +

"Your Points: " + user.Puntuation.ToString() + '\n' + "Seats: " + '\n' + \_ticket.VoucherSeats;

\_ticket.PayWithPoints = true;

\_ticket.PayCompleted = true;

\_BuyTicketService.UpdateBuyTicket(\_ticket);

return RedirectToAction(nameof(ViewVoucher));

}

ViewData["msg"] = "Invalid Values";

return View();

}

Ya después se muestra el comprobante y se da la opción de descargarlo.

public IActionResult ViewVoucher(int id)

{

if (id != 0)

{

\_ticket = \_BuyTicketService.Get(id);

}

return View(new List<string> { \_ticket.Voucher });

}

public IActionResult PDF()

{

return new ViewAsPdf("ViewVoucher", new List<string> { \_ticket.Voucher })

{

FileName = "Voucher.pdf",

PageOrientation = Rotativa.AspNetCore.Options.Orientation.Portrait,

PageSize = Rotativa.AspNetCore.Options.Size.A4

};

}

Si en alguno de los pasos de la compra se pulsa cancelar compra, nos dirige al siguiente método para actualizar los datos, y de ahí se pasa para Horaries con todas sus opciones listadas.

public IActionResult Cancel()

{

\_ticket = \_BuyTicketService.GetAllBuy\_Tickets()

.Include(b => b.Reserved\_Seats)

.Include(b => b.Horary)

.FirstOrDefault(t => t.Id == \_ticket.Id);

\_ticket.Horary.ReservedTickets -= \_ticket.NumberOfEntrance.Value;

\_HoraryService.UpdateHorary(\_ticket.Horary);

\_ReservedSeatsService.RemoveReservedSeats(\_ticket.Reserved\_Seats);

\_BuyTicketService.RemoveBuyTicket(\_ticket);

TempData["msg"] = "Buy has been canceled";

return RedirectToAction("Index", "Horaries");

}

Nota: En el controlador solo se dejaron las condiciones de las posibles cosas que podían suceder. La lógica de los métodos en particular esta en la capa de los servicios, y si hay actualizaciones o consultas con los datos se llama de los servicios a los repositorios que son los que interactúan con la base de datos.

Realicé también el UnitTesting del proyecto y en el informe técnico hice el manual de usuario.