Implementando los casos faltantes y arreglando cositas

A esta altura de juego nos faltan implementar 4 casos de uso que son:

- Ver detalles del evento
- Ver eventos en nuestro feed
- Remover un evento de mi calendario
- Dejar de seguir a un coach

A partir de ahora, como estos casos son tan simples de implementar comenzaremos directamente con la implementación de los mismos para concentrarnos en cosas mas importantes como la refactorización.

Implementación del Detalle de Evento

Bien, vamos a ver la lógica de implementación.

- En el home page, en donde mostramos la lista de eventos, quiero convertir el nombre del coach en un hipervínculo, al hacer clic, deseo ir a una nueva página donde pueda ver los detalles del evento.
- Ahí quiero que se renderice el nombre del coach y justo al lado un botón de seguimiento, si el usuario logueado está siguiendo a este coach, el botón debe ponerse azul y la etiqueta debe decir “Following”, en caso contrario el botón debe ser azul y la etiqueta debe decir “Follow”.
- Debajo vamos a tener el lugar donde se va a realizar el evento y el horario.

- Si el usuario logueado va a asistir al evento seleccionado, voy a agregar una tercera línea que va a tener un mensaje que diga “You are going to go to this event” en caso contrario no renderizamos nada.
- De igual forma debemos de validar que el botón de seguir o dejar de seguir, solo aparezca a usuarios logueados.
- El botón de seguimiento debe tener la implementación del cambio (toggle), es decir, si estoy siguiendo al coach y hago clic debo dejarlo de seguir y cambiar.

Para resolver nuestros requisitos de implementación, lo primero que vamos es a modificar nuestro “_EventsPartial.cshtml” en la sección donde renderizamos el nombre del coach, lo vamos a modificar por lo siguiente:

Para implementar lo que pedimos tienes que hacer lo siguiente

```
<span class="coach">
 <a asp-controller="Events" asp-action="Details" asp-route-id="@item.Id">
     @item.Coach.Name
 </a>
    @if (item.IsCanceled)
    {
        <span class="label label-warning">Canceled</span>
    }
</span>
```

Posterior a eso entonces tenemos nuestro controlador Events y crear la acción detalle, donde vamos a recibir un Id, y tendremos que buscar ese evento en la base de datos, dado el id, le incluimos

el coach, pues en el detalle vamos a renderizar su nombre, por ende, para que no venga vacío debemos incluirlo, lo propio hacemos con el tipo de evento si no podemos encontrar el evento, retornamos un notfound.

```
public IActionResult Details(int id)
{
    var ev = _context.Events
        .Include(p => p.Coach)
        .Include(p => p.Type)
        .SingleOrDefault(p => p.Id == id);

    if (ev == null)
        return NotFound();

}
```

Creamos un ViewModel porque el objeto evento no es suficiente para renderizar la información que queremos mostrar en este View, en este caso hay atributos que necesitamos que no son parte del modelo de dominio o clase y esto es algo que se repite muchas veces, por esto es que son tan importantes los ViewModels, por ejemplo, si el usuario esta siguiendo al coach o no, o si esta asistiendo al evento o no, no hacen parte del dominio Events, sin embargo yo deseo almacenar ese valor para poder tomar una decisión en el Front, revisemos el ViewModel como quedaría llamándolo “EventDetailsViewModel”, donde tendremos una instancia a la clase Event y dos valores booleanos uno para saber si el usuario está asistiendo al evento y otro para saber si está siguiendo al coach.

```
public class EventDetailsViewModel
{
    public Event Event { get; set; }

    public bool IsAttending { get; set; }

    public bool IsFollowing { get; set; }
}
```

Volvemos al controlador, donde instanciamos el ViewModel y si el usuario está logueado verificamos si tiene registros en la tabla Attendances y seteamos la propiedad IsAttending, esto lo hacemos con el Any, que nos devuelve un resultado verdadero o falso si se cumple la condición de encontrar algún registro con los filtros y datos suministrados. Hacemos lo propio con la tabla Followings para setear la propiedad IsFollowing. Finalmente retornamos una vista de nombre Details donde le pasamos el objeto que creamos.

```
            if (ev == null)
                return NotFound();

            var vm = new EventDetailsViewModel { Event = ev };

            if(User.Identity.IsAuthenticated)
            {
                var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

                vm.IsAttending = _context.Attendances
                    .Any(a => a.EventId == ev.Id && a.AttendeeId == userId);

                vm.IsFollowing = _context.Followings
                  .Any(a => a.FolloweeId == ev.CoachId && a.FollowerId == userId);
            }

            return View("Details", vm);
```

Luego generamos la vista aquí lo que hacemos es renderizar el nombre del coach, si el usuario esta autenticado renderizo los botones de seguimiento o no, añadimos un párrafo con los datos del lugar y la fecha y si vamos a asistir al evento mostramos el párrafo que habíamos dicho que pondríamos, después probamos.

```
@model EcCoach.Web.ViewModels.EventDetailsViewModel
@{
    ViewBag.Tittle = "Details";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

```
<h2>
    @Model.Event.Coach.Name
    @if (User.Identity.IsAuthenticated)
    {
        if (Model.IsFollowing)
        {
            <button class="btn btn-info">Following</button>
        }
        else
        {
            <button class="btn btn-default">Follow</button>
        }
    }
</h2>
<p>
    Performing at @Model.Event.Venue on @Model.Event.DateTime.ToString("d MMM") at
@Model.Event.DateTime.ToString("HH:mm")
</p>

@if(User.Identity.IsAuthenticated && Model.IsAttending)
{
    <p>
        You are going to go to this event. :D
    </p>
}
```

Evitando los Magic Strings 2.0

Como habíamos visto en capítulos anteriores, los strings mágicos son peligrosos, a pesar de que nosotros hemos adoptado la medida de usar nombres estándares para nuestras acciones, puede llegar el momento en que necesitemos hacer un redirect a una con un nombre distinto al que habíamos ideado originalmente, por esto en NetCore se inventaron un modo mucho más simple de evitar los magic strings y es haciendo uso del NameOf, este lo que hace es que se convierte en un string del tipo de la acción dentro del controlador que estemos llamando.

Así que vamos a reemplazar el anterior “Details” por su nueva forma. Les queda de tarea hacer este procedimiento con los demás controladores y cada return View.

```
return View("Details", vm);
return View(nameof(Details), vm);
```

LookUp

Bien, ahora vamos a corregir un error que tenemos, si nos vamos al Feed donde salen los eventos y yo hago clic en “Going”, el cambia a azul como que voy a ir, como debe de ser, este es el funcionamiento correcto, pero si refresco la página se va, aunque si verifico en la base de datos puedo ver que esta bien, pues, tengo que corregir el estado de inicio del evento.

Lo que haremos es ir al controlador home, en la acción Index y lo que vamos a hacer es que antes de inicializar el ViewModel, voy a loguear todas las posibles asistencias futuras del usuario.

Nosotros vamos a necesitar convertir la lista que vamos a manipular a una estructura de datos que me permita buscar rápido dentro de la lista en cuestión una asistencia; porque cuando rendericemos cada evento, necesitamos marcar si yo (el usuario logueado) voy a asistir o no al evento que se está renderizando, para eso hacemos uso de un LookUp.

Un LookUp es como un diccionario, internamente usa una tabla hash para rápidamente buscar y localizar objetos por eso es tan potente y veloz el LookUp recibe una expresión lambda que nos va a servir para indicar la llave y después de eso instanciamos el resultado en el ViewModel.

```
        g.Venue.Contains(query));
            }

            var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
            var attendances = _context.Attendances
                .Where(a => a.AttendeeId == userId && a.Event.DateTime > DateTime.Now)
                .ToList()
                .ToLookup(a => a.EventId);

            var vm = new EventsViewModel
            {
                UpcomingEvents = upcomingEvents,
                ShowActions = User.Identity.IsAuthenticated,
                Heading = "Upcoming Events ", //My upcoming events on the other screen
                SearchTerm = query,
                Attendances = attendances
            };
```

Obviamente la propiedad Attendances no la tenemos en el ViewModel por lo que tenemos que crearla y lo podemos hacer con el Intellisense quedando de la siguiente forma.

```
public class EventsViewModel
{
    public IEnumerable<Event> UpcomingEvents { get; set; }
    public bool ShowActions { get; set; }
    public string Heading { get; set; }
    public string SearchTerm { get; set; }
    public ILookup<int, Attendance> Attendances { get; internal set; }
}
```

El ILookUp es una interfaz genérica que recibe dos parámetros el primero es el tipo de la llave, y el segundo es el tipo de elemento que deseo almacenar en este LookUp.

Ahora vamos a renderizar en la vista que es nuestro Partial “_EventsPartial.cshtml”, sin embargo en algún momento de la vida que no recuerdo el código de la vista se la habían hecho ligeras modificaciones quedando de esta forma.

```
<div class="details">
```

```
<div class="details">
 <span class="coach">
   <a asp-controller="Events" asp-action="Details" asp-route-id="@item.Id">
      @item.Coach.Name
   </a>
   @if (item.IsCanceled)
   {
    <span class="label label-warning">Canceled</span>
   }
   @if (Model.ShowActions)
   {
    <button data-user-id="@item.CoachId"
            asp-route-id="@item.CoachId"
            class="btn btn-default btn-sm pull-right js-toggle-follow">
     Follow
    </button>
   }
 </span>
 <span class="type">
            @item.Type.Name
 </span>
    @if (Model.ShowActions && !item.IsCanceled)
    {
      <button data-event-id="@item.Id"
              asp-route-id="@item.Id"
              class="btn btn-default btn-sm pull-right js-toggle-attendance">
      Going?
      </button>
      <button data-user-id="@item.CoachId"
              asp-route-id="@item.CoachId"
              class="btn btn-default btn-sm pull-right js-toggle-follow">
       Follow
      </button>
    }
   </div>
```

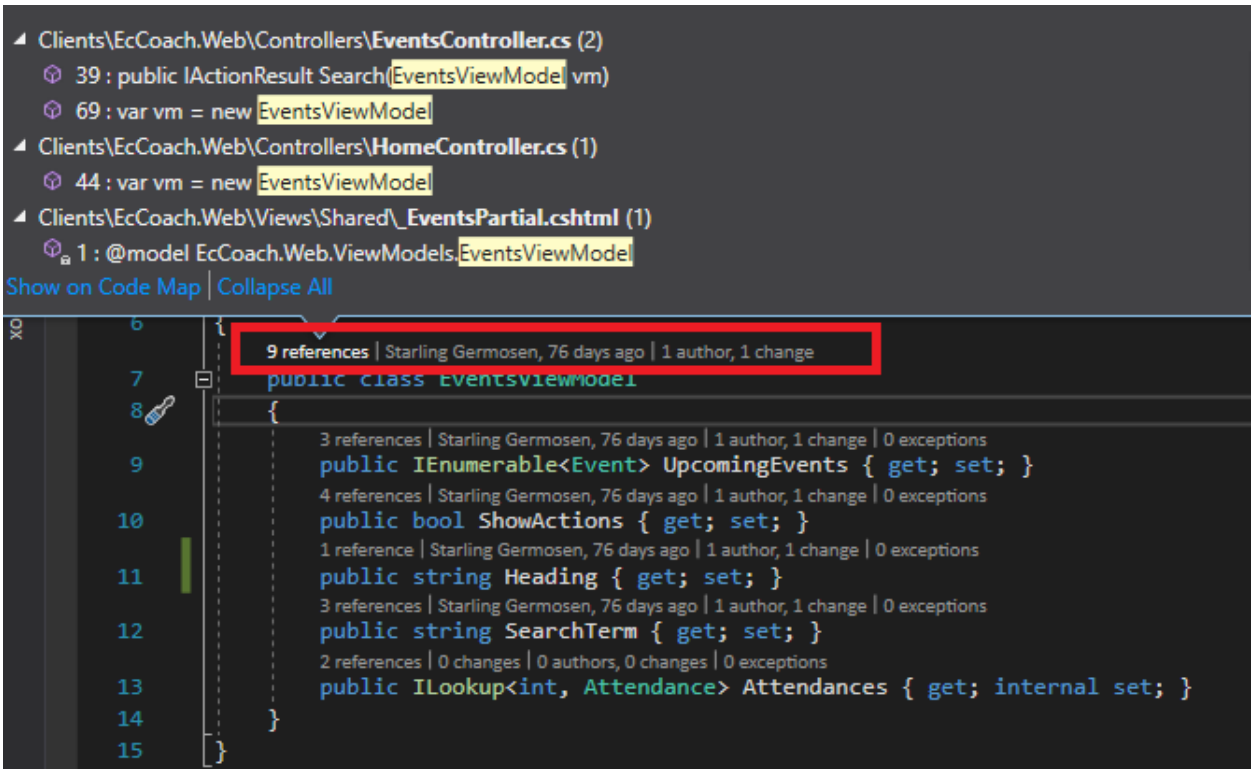
Donde lo que habíamos hecho era mover el botón de seguir al coach justo al lado del nombre de su nombre y con la propiedad ShowActions que habíamos programado en el controlador, validamos si debemos mostrar o no la acción de seguir, lo propio para el indicador de si vamos a ir al evento o no, donde validamos también que el evento no esté cancelado para poder mostrar la opción.

Bueno ahora, nos valemos de la técnica de Razor y código C# para cargar de manera dinámica una clase btn info u otra; dentro de cualquier elemento del Dom, yo puedo hacer uso del @ para

indicar que voy a iniciar una instrucción de C#, y hago uso del paréntesis para indicar que es “In Line Code”. Si tenemos un resultado de asistencia, renderizamos una clase btn-info en caso contrario btn-default. Quedaría de esta forma.

```
<button data-event-id="@item.Id"
        asp-route-id="@item.Id"
        class="btn btn-default
               @(Model.Attendances.Contains(item.Id) ? "btn-info" : "btn-default")
               btn-sm pull-right js-toggle-attendance">
        Going?
</button>
```

Si probamos, veremos que todo está nítido, carga de manera inicial el color que debe cargar y cambia cuando hago la operación. pero ahora surge un inconveniente, nosotros tenemos una nueva propiedad, que es el look up, la cual si no la inicializamos nos puede arrojar excepciones por referencia nula, por ende, tenemos que buscar donde quiera que hayamos implementado el ViewModel de Events y en todas sus referencias me aseguro de que al inicializar el EventsViewModel, le pase valores al LookUp si lo amerita, para esto sencillamente nos situamos arriba de la clase y podemos ver al lado de la ultima persona que modifico, la cantidad de referencias, desde aquí podemos verlas e ir una por una a tomar los correctivos de lugar las de vistas no nos interesan, solo las de clases.



Podemos ver que tenemos una referencia en EventsController, en la parte del Search, solo hacemos una redirección por lo que no tenemos que preocuparnos, esa la podemos dejar así.

Pero en la otra aquí en la acción Attending, tenemos una referencia a EventsViewModel, donde retornamos para mostrar los eventos a los que estoy atendiendo, así que vamos al controlador home, copiamos el código para marcar si esta asistiendo, porque no queremos un Null Reference Exeption, pero aquí vienen a crucificarme los jinetes del Apocalipsis y con razón, porque estoy violando el principio DRY, bueno, eso es algo que vamos a resolver mas tarde, esto lo hacemos de hecho para darle mas sentido al patrón repositorio, así que tranquilos, no me maten, por ahora vamos a copypastear y después metemos mano,

```
var events = _context.Attendances
.Where(a => a.AttendeeId == userId)
.Select(a => a.Event)
.Include(p => p.Type)
.Include(p => p.Coach)
.ToList();

var attendances = _context.Attendances
  .Where(a => a.AttendeeId == userId && a.Event.DateTime > DateTime.Now)
  .ToList()
  .ToLookup(a => a.EventId);

var vm = new EventsViewModel
{
    UpcomingEvents = events,
    ShowActions = User.Identity.IsAuthenticated,
    Attendances= attendances
};
```

Recordando; siempre que creamos una nueva propiedad, debemos revisar todas las referencias que se tienen hacia ese objeto y validar que necesitemos o no inicializar esos nuevos elementos. Seguimos con nuestro proyecto.

Eliminando la Participación en un Evento

Nosotros tenemos implementado que si no voy a asistir a un evento puedo hacer clic y cambia el botón, además de que este siempre carga en el estado correcto, pero no tenemos la parte inversa, para eliminar una asistencia.

En el script que tenemos en “Events.cshtml” nos damos cuenta que tenemos siempre la asunción de que tenemos que crear, pero no siempre es correcto, por lo que tenemos que validar que si tiene una clase default haga el créate, pero, en caso contrario llame una nueva API que en este caso sería de eliminar.

```
$(".js-toggle-attendance").click(function (e) {

    var button = $(e.target);
    if (button.hasClass("btn-default")) {

      $.post("/api/attendances/", { eventId: button.attr("data-event-id") })
             .done(function () {
                 button.removeClass("btn-default").addClass("btn-info").text('Attending')
              })
             .fail(function () {
                 alert("Someting fail");
              }); //Faltaba este punto y coma :'(
     }//No me dejes cuando hagas copy :D
});
```

En el caso contrario entonces significa que tiene la clase info, por lo que no tenemos que preguntar y podemos ir directamente a un else donde copypasteamos el metodo del apartado anterior y hacemos un par de modificaciones, la primera es que en lugar de un post vamos a hacer una llamada ajax, donde el método va a ser DELETE, luego, en caso de que sea satisfactorio en lugar de remover la clase default la que removemos es la info, le añadimos la default y cambiamos el texto por la pregunta "Going?"

```
   else {
         $.ajax({
             url: "/api/attendances/" + button.attr("data-event-id"),
             method: "DELETE"
         })
           .done(function () {
              button.removeClass("btn-info").addClass("btn-default").text('Going?')
           })
           .fail(function () {
              alert("Someting fail");
           });
        }
```

Este código esta grande y feo, pero tranquilos que después lo vamos a arreglar, por ahora concentrémonos en que funcione y vamos a crear el método para borrar la asistencia.

Creamos una acción en AttendancesController llamada DeleteAttendance que recibe un id como parámetro, la marcamos con un Request de Delete, para que funcione como tal, obtenemos el usuario logueado, localizamos la asistencia de este, si la asistencia es nula retornamos Notfound, en caso contrario la borramos, guaramos y retornamos ok con el id que eliminamos. Quedando algo como esto.

```
[HttpDelete]
public IActionResult DeleteAttendance(int id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

    var attendance = _context.Attendances
        .SingleOrDefault(a => a.AttendeeId == userId && a.EventId == id);

    if (attendance == null)
        return NotFound();

    _context.Attendances.Remove(attendance);
    _context.SaveChanges();

    return Ok(id);
}
```

Probamos y verificamos que todo esté funcionando como se supone que deba funcionar. Ya tenemos la aplicación funcionando como debería ahora es momento de refactorizar para que digan que nosotros sabemos programar, aunque sea mentira :D.

Resumen

En este módulo terminamos la implementación de los casos de uso restantes, conocimos como evitar los Magic String dentro de nuestro controlador, echamos un vistazo a la potencia de los LookUps, sentamos las bases para los procesos venideros de remasterización.