Verbesserung Booking | Ignjatovic David

GetRoomsAsync()

alter Code:

```
public async Task<List<RoomDTO>> GetRoomsAsync()
{
    return await _dbContext.Rooms.Select(r => new RoomDTO()
    {
        RoomNumber = r.RoomNumber,
        RoomType = r.RoomType,
        From = r.Bookings.OrderByDescending(b=> b.From.Year).Where(b => b.RoomId == r.Id && b.From.Year >= DateTime.Now.Year).Select
        To = r.Bookings.OrderByDescending(b => b.From.Year).Where(b => b.RoomId == r.Id && b.To > DateTime.Now).Select(i => i.To).Fi
        IsEmpty = r.Bookings.OrderByDescending(b => b.From.Year).Where(b => b.RoomId == r.Id).Select(i => i.From.Day > DateTime.Now.
        CurrentBooking = r.Bookings.Single(b=> b.RoomId == r.Id)
    }).ToListAsync();
}
```

Meine Lösung für *GetRoomsAsync()* war leider nicht sehr schön.

neuer Code:

```
public async Task<List<RoomDTO>> GetRoomsAsync()
{
    return await _dbContext.Rooms.Include(r => r.Bookings).ThenInclude(b => b.Customer).Select(r => new RoomDTO()
    {
        RoomNumber = r.RoomNumber,
        RoomType = r.RoomType,
        From = r.Bookings != null ? r.Bookings.OrderByDescending(b => b.From).First().From : null,
        To = r.Bookings != null ? r.Bookings.OrderByDescending(b => b.From).First().To : null,
        IsEmpty = r.Bookings != null ? CheckIsEmpty(r.Bookings.OrderByDescending(b => b.From).First()) : true,
        CurrentBooking = r.Bookings != null ? r.Bookings.OrderByDescending(b => b.From).First() : null
    })
        .ToListAsync();

}

public static bool CheckIsEmpty(Booking booking)
{
    if (booking.To == null || booking.To > DateTime.Now)
    {
        if (booking.From < DateTime.Now)
        {
            return false;
        }
        else
        {
            return true;
        }

    }
    else
    {
        return true;
    }
}
```

Die neue Methode ermöglicht mir ein besseres Überprüfen des Datums. Mit *booking* als Parameter lässt sich feststellen, ob der Raum am heutigen Tag zur Verfügung steht.

Es wurde berücksichtigt, dass ein Customer im Voraus buchen kann.

Mit Hilfe der Elvis-Operatoren konnte ich leicht überprüfen, ob die Buchung null ist.

Index.cshtml (Main page)

alter Code:

```
<div class="row">
    <table class="table">
        <thead>
            <tr>
                <th>RoomNumber</th>
                <th>Roomtype</th>
                <th>IEmpty</th>
                <th>From</th>
                <th>To</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model.Rooms) {
                <tr>
                 <td>@item.RoomNumber</td>
                 <td>@item.RoomType</td>
                 <td>@item.IsEmpty</td>
                 <td>@item.From</td>
                 <td>@item.To</td>

                </tr>
            }
        </tbody>
    </table>
</div>
```

Die Klasse **index.cshtml** habe ich während der Prüfung aufgrund von Zeitmangel nur grob zusammengestellt. In der Verbesserung sieht die Tabelle der Angabe entsprechend aus:

neuer Code:

```
<div class="row">
    <table class="table">
        <thead>
            <tr>
                <th>Nachname</th>
                <th>Vorname</th>
                <th>Anzahl Buchungen</th>
                <th>Bearbeiten</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model.Customers) {
                <tr>
                 <td>@item.LastName</td>
                 <td>@item.FirstName</td>
                 <td>@item.CountBookings</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

Nachnamen, Vornamen und die Anzahl der Buchungen werden nun angezeigt.

Auch die Liste, welche die Tabelle befüllt, hat sich geändert. Jetzt verwende ich eine neu erstelltes DTO namens *CustomerDTO*:

CustomerDTO:

```
public class CustomerDTO
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public int CountBookings { get; set; }
    public int CustomerId { get; set; }
}
```

neuer Code:

```
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;
    private IUnitOfWork _unitOfWork;

    [BindProperty]
    public List<CustomerDTO> Customers { get; set; } // neue Liste


    public IndexModel(ILogger<IndexModel> logger, IUnitOfWork unitOfWork)
    {
        _logger = logger;
        _unitOfWork = unitOfWork;
        Customers = new List<CustomerDTO>();
    }

    public async Task OnGet()
    {
        Customers = await _unitOfWork.Customers.GetCustomersForView(); // neue Methode im Repository

    }
}
```

Das Abfragen der Customer passiert mit einem einfachen Select, welches dann die DTO-Liste befüllt:

neuer Code:

```
public async Task<List<CustomerDTO>> GetCustomersForView()
{
    return await _dbContext.Customers.Select(c => new CustomerDTO
    {
        CustomerId = c.Id,
        LastName = c.LastName,
        FirstName = c.FirstName,
        CountBookings = _dbContext.Bookings.Where(b => b.CustomerId == c.Id).Count()
    }).ToListAsync();
}
```

Filter

Um die Filter-Option zu erfüllen, fügen wir folgenden Code hinzu:

neuer Code:

```
<form method="post" class="form-inline">
    <div class="form-group">
        <label>Filter Name:</label>
        <input  class="form-control" type="text"/> @*asp-for="  "*@ <!--werte setzen in NameFiltered-->

        <label>Nur mit aktuellen Buchungen</label>
        <input class="form-control" type="checkbox"/> @*asp-for="actualBookingsChecked"*@ <!--werte setzen in ActualBookingsChecked--

     </div>
    <div class="form-group">
        <input type="submit" value="Aktualisieren" class="btn btn-primary" />
    </div>
</form>
```

Die Form wird verwendet, um die Tabelle zu filtern. Leider funktioniert sie noch nicht ganz.

Die Model-Klasse wurde an die Form angepasst:

neuer Code:

```
private readonly ILogger<IndexModel> _logger;
private IUnitOfWork _unitOfWork;

[BindProperty]
public List<CustomerDTO> Customers { get; set; }

[BindProperty]
public string NameFiltered { get; set; }


public IndexModel(ILogger<IndexModel> logger, IUnitOfWork unitOfWork)
{
    _logger = logger;
    _unitOfWork = unitOfWork;
    Customers = new List<CustomerDTO>();
    NameFiltered = "alle";
}

public async Task OnGet()
{
    var list = await _unitOfWork.Customers.GetCustomersForView();
    foreach (var item in list)
    {
        if (NameFiltered == "alle")
        {
            Customers.Add(item);

        }
        if (item.FirstName.Contains(NameFiltered) ||item.LastName.Contains(NameFiltered))
        {
            Customers.Add(item);
        }
    }

}
```

Was hier noch nicht funktioniert ist das Leeren der Tabelle via Buttonclick. (leider keine Kraft mehr um den Fehler zu suchen)

Edit

Um ein Feld zu editieren fügen wir eine weiter Spalte in die Tabelle ein:

```
<td><a asp-page="/Customers/Edit" asp-route-id="@item.CustomerId">Bearbeiten</a></td>
```

Dieser Link leitet uns auf unsere Edit-Seite und übergibt die CustomerId, welche benötigt wird, um einen Customer zu bearbeiten.

Edit.cshtml

Edit

In der Edit-Model-Klasse haben wir die Methode *OnGetAsync()*, welche mit dem Edit-Link aufgerufen wird. Dadurch wird die ID mitgeliefert.

So können wir dann auf den entsprechenden Customer zugreifen:

neuer Code:

```
[BindProperty]
public Customer CurrentCustomer { get; set; }


public async Task<IActionResult> OnGetAsync(int? id)
{
    CurrentCustomer = await _uow.Customers.GetByIdAsync((int)id);

    return Page();
}
```

In *Edit.cshtml* können wir den Customer mit Hilfe der Form anzeigen und auch bearbeiten. Das Speichern passiert über einen Button.

neuer Code:

```
<div class="row">
<div class="col-md-4">
    <form method="post">
        <div class="form-group">
            <label>Vorname</label>
            <input asp-for="CurrentCustomer.FirstName" class="form-control" type="text"/>
            <span asp-validation-for="CurrentCustomer.FirstName" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label>Nachname4</label>
            <input asp-for="CurrentCustomer.LastName" class="form-control" type="text"/>
            <span asp-validation-for="CurrentCustomer.LastName" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label>Email-Adresse</label>
            <input asp-for="CurrentCustomer.EmailAddress" class="form-control" type="email"/>
            <span asp-validation-for="CurrentCustomer.EmailAddress" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label>Vorname</label>
            <input asp-for="CurrentCustomer.CreditCardNumber" class="form-control" type="text"/>
            <span asp-validation-for="CurrentCustomer.CreditCardNumber" class="text-danger"></span>
        </div>
        <div class="form-group">
            <input type="submit" value="Speichern" class="btn btn-primary" />
        </div>
    </form>
</div>
</div>
```

Nachdem der Button gedrückt wurde, wird ein Post-Request ausgeführt. Die Methode *OnPostAsync()* wird "gestartet".

neuer Code:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    try
    {
        await _uow.SaveChangesAsync();
        return Page();
    }
    catch (ValidationException e)
    {

        var validationResult = e.ValidationResult;
        foreach (var field in validationResult.MemberNames)
        {
            ModelState.AddModelError(field, validationResult.ToString());
        }

        ModelState.AddModelError("", e.Message);

        CurrentCustomer = await _uow.Customers.GetByIdAsync(CurrentCustomer.Id);

        return Page();
    }
}
```

Bookings anzeigen

Um die Bookings eines Customers anzuzeigen, fügen wir zur unserer *OnGetAsync()* Methode noch etwas hinzu:

neuer Code:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    CurrentCustomer = await _uow.Customers.GetByIdAsync((int)id);
    CustomerBookings = await _uow.Bookings.GetBookingsForCustomer((int)id);

    return Page();
}
```

Die Methode *GetBookingsForCustomer()* holt sich alle Bookings, welche zu dem jeweiligen Customer gehören.

neuer Code:

```
public async Task<List<BookingForCustomerDTO>> GetBookingsForCustomer(int customerId)
        {
            return await _dbContext.Bookings
                .Where(b => b.CustomerId == customerId)
                .Include(b => b.Customer)
                .Include(b => b.Room)
                .Select(b => new BookingForCustomerDTO
                {
                    From = b.From,
                    To = b.To,
                    RoomNumber = b.Room.RoomNumber,
                    RoomType = b.Room.RoomType.ToString(),
                    DaysSleeped = b.To != null ? (b.To - b.From).Value.TotalDays : -1
                }).ToListAsync();
        }
```

DaysSleep stellt ein Problem dar, da es eine Kommazahl ausgibt. Um das zu umgehen, habe ich die Zahl gerundet:

```
DaysSleeped = b.To != null ? Math.Round((b.To - b.From).Value.TotalDays,0) : -1
```

Die Buchungen werden in einer Tabelle angezeigt, nachdem sie in die Liste *CustomerBookings* gespeichert worden sind.

```
<h4>Zimmerbuchungen</h4>
<table class="table">
 <thead>
        <tr>
            <th>
                Von
            </th>
            <th>
                Bis
            </th>
            <th>
                Zimmer
            </th>
            <th>
                Zimmertyp
            </th>
              <th>
                Anzahl Übernachtungen
            </th>
        </tr>
</thead>
<tbody>
    @foreach (var item in Model.CustomerBookings) {
            <tr>
                <td>@item.From</td>
                <td>@item.To</td>
                <td>@item.RoomNumber</td>
                <td>@item.RoomType</td>
                @if(@item.DaysSleeped == -1)
                {
                    <td></td>
                }
                @if(@item.DaysSleeped != -1)
                {
                    <td>@item.DaysSleeped</td>
                }
            </tr>
        }
</tbody>
</table>
```

In *OnPostAsync()* fügen wir noch diese Zeile hinzu, um die Tabelle nach dem Bearbeiten eines Customers zu updaten.

```
CustomerBookings = await _uow.Bookings.GetBookingsForCustomer(CurrentCustomer.Id);
```

Edit

Das Editieren passiert, in dem man auf den "Speichern" Button drückt. Dieser funktioniert aber leider nicht.

Fügt man allerdings `await _uow.Customers.AddAsync(CurrentCustomer);` hinzu, müsste man die E-Mail ändern, dann wäre ein neuer Customer in unserer Tabelle.