

IT-Projekt

„FunEvents GmbH“

Entwicklung des
*Kunden**b**uchungssystem**s** (KBS)*
als Webanwendung
mit ASP.NET Core MVC



ASP.NET
Core
MVC

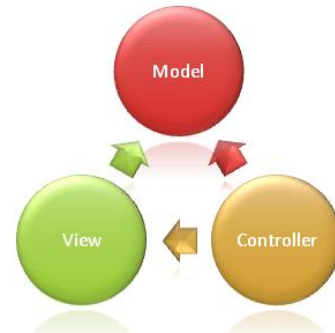
Inhaltsverzeichnis

1. Begriffsbestimmungen	3
1.1. MVC – Model View Controller	3
1.1.1. Model (Modell)	3
1.1.2. View (Ansicht)	3
1.1.3. Controller	3
1.2. Model View ViewModel (MVVM)	3
1.3. Entity Framework Core (EF Core)	4
1.4. Dependency Injection (Abhängigkeitsinjektion)	4
2. Einstieg zur Erstellung des Kunden-Buchungs-System (KBS)	5
2.1. Erstellung Projekt mit Visual Studio	5
2.2. NuGet Pakete des EFCore installieren	5
2.3. Datenbank-Kontext erstellen	6
2.4. Datenbank-Kontext als Service registrieren und Connection-String mitgeben	6
3. Umsetzungsbeispiel MVC bzw. MVVM für den Login-Prozess	7
3.1. Controller festlegen	7
3.2. Action-Methode „Login“ erstellen	7
3.3. View-Model erzeugen	8
3.4. Erstellung der Login-View	9
3.5. Post-Methode „Login“ erstellen	9
3.6. Session mit Cookie erzeugen	11
4. Hilfreiche Codefragmete zur Umsetzung des Projektes	12
4.1. Datensätze aus dem Kontext laden mit LINQ	12
4.2. Daten in Datenbank updaten bzw. neuen Datensatz erstellen	13
5. Quellen / Literatur	14

1. Begriffsbestimmungen

1.1. MVC – Model View Controller

Das Architekturmuster Model-View-Controller (MVC) unterteilt eine Anwendung in drei Hauptkomponentengruppen: Modelle (Models), Ansichten (Views) und Controller (Controllers).



1.1.1. Model (Modell)

Das Modell repräsentiert die Daten und die Geschäftslogik der Anwendung. Es handelt sich um Klassen, die den Zustand und die Funktionalität der Anwendung definieren.

1.1.2. View (Ansicht)

Die Ansicht ist für die Darstellung der Daten und die Benutzeroberfläche zuständig. Hier kommen HTML, CSS und ggf. JavaScript ins Spiel. Views sind normalerweise passiv und zeigen nur die Daten an, die ihnen vom Controller bereitgestellt werden.

1.1.3. Controller

Der Controller ist das Bindeglied zwischen dem Modell und der Ansicht. Er nimmt Benutzereingaben entgegen, verarbeitet sie und aktualisiert das Modell und die Ansicht. Der Controller in .NET Core ist oft eine C#-Klasse, die die Logik für die Interaktion mit dem Benutzer enthält.

1.2. Model View ViewModel (MVVM)

Das Model-View-Controller (MVC) Designmuster ist darauf ausgerichtet, die Trennung von Anwendungslogik (Model), Benutzeroberfläche (View) und Steuerung (Controller) zu fördern. Oftmals wird zusätzlich das Model-View-ViewModel (MVVM) Designmuster verwendet, insbesondere in Zusammenhang mit modernen Benutzeroberflächen, wie sie in Webanwendungen oder Desktopanwendungen mit komplexen Benutzeroberflächen üblich sind.

Gründe, warum MVVM oft in Verbindung mit MVC eingesetzt wird sind

- Trennung von Logik und Präsentation,
- Unterstützung für Datenbindung bzw. Verhinderung von „Overposting“,
- Bessere Testbarkeit,
- Reduzierung von Code in der View und
- Erleichterung der Wartung und Skalierbarkeit.

1.3. Entity Framework Core (EF Core)

Das Entity Framework (EF) ist ein Object-Relational Mapping (ORM)-Framework, das von Microsoft für die .NET-Plattform entwickelt wurde. Das Entity Framework ermöglicht die objektorientierte Programmierung mit Datenbanken, indem es die Abbildung von Datenbankobjekten auf Objekte in der Anwendungslogik erleichtert.

Bei EF Core erfolgt der Datenzugriff über ein Modell. Ein Modell setzt sich aus Entitätsklassen und einem Kontextobjekt zusammen, das eine Sitzung mit der Datenbank darstellt. Das Kontextobjekt ermöglicht das Abfragen und Speichern von Daten.

Database First und Code First sind dabei zwei unterschiedliche Ansätze im Kontext des Entity Frameworks, die sich darauf beziehen, wie die Datenbankmodelle erstellt und verwendet werden.

EF stellt dafür folgende Möglichkeiten zur Verfügung:

- Generieren eines Modells aus einer vorhandenen Datenbank.
- Manuelles Codieren eines Modells, das der Datenbank entspricht.
- Nachdem ein Modell erstellt wurde, kann „EF-Migrations“ verwendet werden, um eine Datenbank aus dem Modell zu erstellen. Migrationen ermöglichen eine Weiterentwicklung der Datenbank, wenn sich das Modell ändert.

1.4. Dependency Injection (Abhängigkeitsinjektion)

Dependency Injection (DI) ist ein Konzept, bei dem Abhängigkeiten einer Klasse von außen als Dienst bereitgestellt werden, anstatt dass die Klasse sie selbst erstellt. In .NET Core MVC ist Dependency Injection ein integraler Bestandteil des Frameworks, denn sie fördert eine lose Kopplung zwischen Komponenten, erleichtert das Testen und verbessert die Wartbarkeit deines Codes.

In .NET Core MVC findet Dependency Injection (DI) hauptsächlich im sogenannten "Dependency Injection Container" statt. Der DI-Container wird in der Startup.cs-Datei konfiguriert, genauer gesagt in der Methode „ConfigureServices“. Hier werden die Dienste registriert, die in der Anwendung benötigt werden.

Die eigentliche Injection erfolgt dann in den Klassen, die die Abhängigkeiten benötigen, wie zum Beispiel in den Controllern, Views oder auch anderen Services.

2. Einstieg zur Erstellung des Kunden-Buchungs-System (KBS)

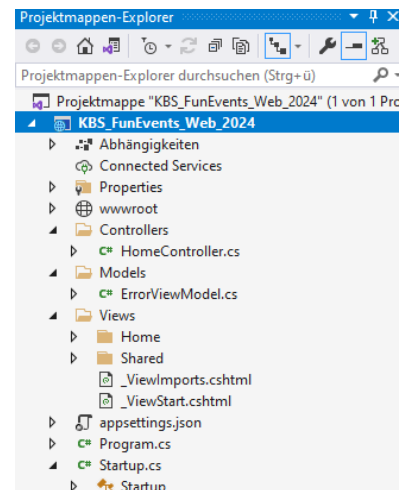
2.1. Erstellung Projekt mit Visual Studio

Starten Sie Visual Studio und erstellen Sie ein neues Projekt. Wählen Sie dabei die Vorlage ASP.NET Core-Web-App (Model View Controller) für C# aus.



Anschließend vergeben Sie den Projektnamen „KBS_FunEvents_Web_2024“. Ändern Sie unbedingt den Speicherort des Projekts auf Ihr Laufwerk ab.

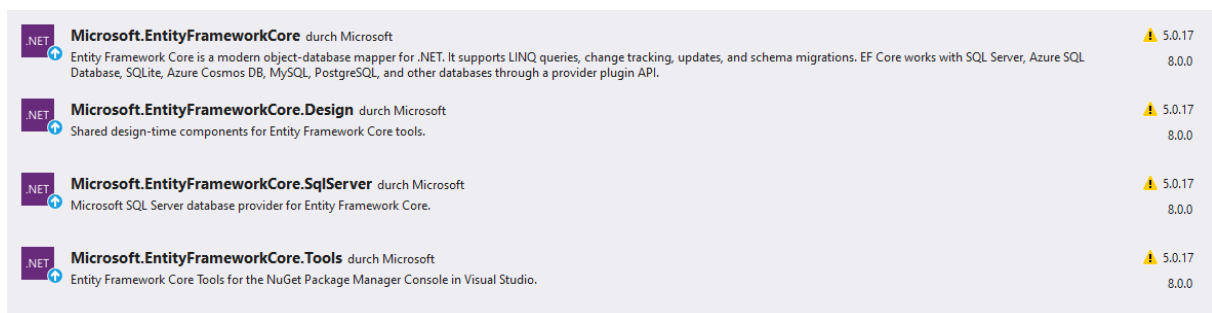
Auf der nächsten Seite wählen Sie nun das Zielframework .NET 5 aus, setzen einen Haken „Für HTTPS konfigurieren“ und lassen den Authentifizierungstyp auf „keine“ stehen. Nun können Sie das Projekt erstellen und Sie erhalten folgende Projektmappe.



2.2. NuGet Pakete des EFCore installieren

Für den Datenaustausch mit der Datenbank benötigen wir vier NuGet-Pakete (siehe Screenshot). Bei der Installation ist zu beachten, dass eine zum Zielframework kompatible Version des Paketes ausgewählt wird.

Die benötigten Pakete sind unter „Extras->NuGet-Paket-Manger->NuGet-Pakte für Projektmappe verwalten...“ zu finden.



2.3. Datenbank-Kontext erstellen

Nun wollen wir den Datenbank-Kontext über das Entity-Framework erzeugen. Dazu müssen wir über die Paket-Manger-Konsole („Extras->NuGet-Paket-Manger->...“) folgenden Befehl eingeben:

```
Scaffold-DbContext „Server=ITSW16SQL2\SQL_A; Database=IHREDatenbank; Trusted_Connection=True;“  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context kbsContext -DataAnnotations
```

Nach erfolgreichem „Build“ sollten Sie nun im Projektextplorer unter Models den „kbsContext.cs“ sowie die Tabellen der Fun-Events-Datenbank als C#-Klasse finden.

2.4. Datenbank-Kontext als Service registrieren und Connection-String mitgeben

Bei Betrachtung der „kbsContext.cs“-Datei fällt sofort auf, dass im Quellcode ein Warnhinweis bzgl. des Connection-Strings vorliegt. Deshalb lagern wir den Connection-String aus und übergeben ihn im Datenbank-Kontext - den wir als Service registrieren - mit.

Folgende Schritte sind dabei notwendig:

- a) Connection-String-Eigenschaften in der „appsettings.json“-Datei ablegen

```
"ConnectionStrings": {  
  "DefaultConnection": "Server=ITSW16SQL2\\SQL_A; Database= IHREDatenbank; Trusted_Connection=True;"  
}
```

- b) kbsContext als Service in der „Startup.cs“-Datei registrieren und Connection-String mitgeben

```
0 Verweise  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddControllersWithViews();  
  
    //kbsContext als Service registrieren  
    services.AddDbContext<kbsContext>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));  
}
```

- c) Löschen der „protected override void OnConfiguring(...)“-Methode in der „kbsContext.cs“-Datei

3. Umsetzungsbeispiel MVC bzw. MVVM für den Login-Prozess

Die erste Funktion, die im Projekt umgesetzt werden sollte, ist der Login-Prozess. Dieser wird anhand des MVC bzw. MVVM-Patterns vorgestellt.

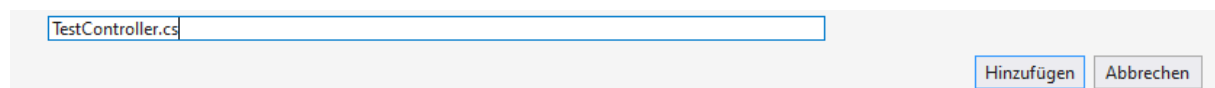
3.1. Controller festlegen

Dabei ist zu überlegen, welcher Controller diese Aktion ausführt. Grundsätzlich ist bei Controllern folgendes zu beachten:

- Jeder Controller sollte eine klare und spezifische Verantwortlichkeit haben. Dies fördert eine saubere Trennung der Anwendungslogik und erleichtert das Verständnis und die Wartung des Codes.
- Es ist oft besser, mehrere kleine, spezialisierte Controller zu haben, anstatt einen großen Controller mit vielen Aktionen. Dadurch verbessert sich die Lesbarkeit und Wartbarkeit des Codes.
- Es ist aber auch darauf zu achten, dass die Anzahl der Controller und Aktionen keine negativen Auswirkungen auf die Performance haben. Zu viele Controller können zu einer unübersichtlichen Routentabelle führen und die Performance beeinträchtigen.

Nun haben Sie die Wahl!

Erstellen Sie für den Login-Vorgang einen eigenen Controller (Rechtsklick auf den Ordner „Controller“ ->Hinzufügen->Controller) oder übernehmen Sie den bereits vorhandenen „HomeController“. Der Controllername ist das Präfix vor dem Wort „Controller“.



D.h. Sie würden in diesem Beispiel eine Klasse TestController.cs erstellen. Mit dem Präfix „Test“ rufen Sie schließlich den „TestController“ über die URL <https://localhost:44311/Test> auf. Der Klassen-Name des Controllers muss deshalb immer mit dem Wort „Controller“ enden.

3.2. Action-Methode „Login“ erstellen

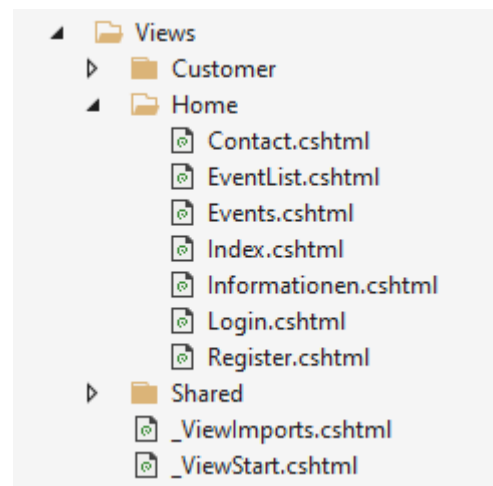
Fügen Sie anschließend Ihrem Controller eine HttpGet-Action-Methode Login hinzu.

```
[HttpGet]
0 Verweise
public IActionResult Login()
{
    return View();
}
```

Die Action „Login“ würde nun eine View aufrufen, die den Namen Login besitzt.

Diese müsste in der Projektmappe auch unbedingt unter „~\Views\IHR-Controllernamen\Login“ abgelegt werden, da sie sonst nicht gefunden wird.

Im angefügten Beispiel wurde die Action-Methode „Login“ im Home-Controller erzeugt und daraus die View automatisiert konfiguriert. Somit wurde die Ansichtsdatei „Login.cshtml“ entsprechend der vordefinierten Ordnerstruktur abgelegt. Dazu später mehr.



Möchten Sie eine View aufrufen, die nicht wie Ihre Action heißt, so müssten Sie unter „return View(„NAME_VIEW““)“ den Namen der View als Parameter mitgeben. Dies funktioniert aber nur, wenn die View im gleichen Controller-Ordner liegt. Möchten Sie eine View eines anderen View-Ordners aufrufen, so müssten Sie den Pfad als Parameter angeben.

3.3. View-Model erzeugen

Für die Login-Ansicht bietet es sich an, ein View-Model für den Login zu verwenden.

Der Grund ist, dass dadurch ein „Overposting“ verhindert werden kann. Overposting (auch als "Mass Assignment" bekannt) bezieht sich auf eine Sicherheitslücke, bei der ein Angreifer versucht, mehr Daten an einen Controller in einem ASP.NET Core MVC-Webanwendungsformular zu übermitteln, als vorgesehen ist. Dies könnte zu unerwünschten Änderungen am Modell bzw. in der Datenbank führen.

```
0 Verweise
public class LoginModelView
{
    [Key]
    0 Verweise
    public int id { get; set; }

    [Required(ErrorMessage = "Email*")]
    [Display(Name = "Email")]
    [DataType(DataType.EmailAddress)]
    0 Verweise
    public string KdEmail { get; set; }

    [Required(ErrorMessage = "Passwort*")]
    [Display(Name = "Passwort")]
    [DataType(DataType.Password)]
    0 Verweise
    public string KdPwHash { get; set; }
}
```

Erstellen Sie einen neuen Ordner in der Projektmappe mit dem Namen „ViewModels“ und fügen Sie anschließend eine Klasse „LoginModelView.cs“ hinzu.

Die Login-View sollte einen Loginnamen (Email), das Passwort sowie eine ID als Property besitzen.

Die Verwendung der Data Annotations macht den Code lesbarer, dokumentiert wichtige Aspekte des Modells und hilft bei der Validierung und Anzeige in der View.

3.4. Erstellung der Login-View

Nun können wir auf Basis des Login-View-Models eine Ansicht automatisiert generieren.

Hierzu gehen Sie in Ihren Controller. Durch Rechtsklick auf die „Login“-Action-Methode können Sie eine neue Razor-Ansicht hinzufügen.

Anschließend nehmen Sie folgende Gerüst-Einstellungen vor und klicken auf „Hinzufügen“.

Razor-Ansicht hinzufügen

Ansichtsnamen

Login

Vorlage

Create

Modellklasse

LoginModelView (KBS_FunEvents_Web_2024.ViewModels)

Datenkontextklasse

kbsContext (KBS_FunEvents_Web_2024.Models)

Optionen

☐ Als Teilansicht erstellen
☐ Auf Skriptbibliotheken verweisen
☒ Layoutseite verwenden

(Leer lassen, wenn der Wert in einer Razor _viewstart-Datei festgelegt wird.)

Hinzufügen

Abbrechen

Nach erfolgreicher Generierung finden Sie in der Projektmappe unter „~\Views\IHR-Controllername\Login“ die dazugehörige „Login.cshtml“-Seite.

Starten Sie das Projekt und rufen Sie die Login-Seite über die URL auf.

Es gilt dabei immer folgende Syntax: <https://localhost:.../IHR-Controllername/IHR-Actionname>

3.5. Post-Methode „Login“ erstellen

Neben der HttpGet-Methode benötigen wir auch eine HttpPost-Methode, um Formulareingaben des Benutzers verarbeiten zu können.

Dazu erstellen Sie ein weitere Action-Methode „Login“, die jedoch als HttpPost-Methode fungieren sollte. Die Methode sollte asynchron ausgeführt werden und ein Login-View-Model als Parameter übergeben bekommen.

```

0 Verweise
public async Task<IActionResult> Login(LoginModelView login)
{
    // ...

    return RedirectToAction();
}

```

Als nächstes wollen wir das übergebene Login-View-Model mit dem Datenbank-Kontext vergleichen und überprüfen, ob Email-Adresse und Passwort des Kunden übereinstimmen. Dazu suchen wir uns zunächst im Datenbank-Kontext, ob es einen Kunden mit den eingegebenen Formulardaten gibt. Hierbei ist jedoch zu beachten, dass in der Datenbank das Passwort als Hashwert abgelegt ist. D.h. es muss das übergebene Passwort als Hashwert umgewandelt werden.

Erstellen Sie deshalb in der Projektmappe einen Ordner „ComputeHash“ und fügen Sie diesen eine Klasse „MD5Generator.cs“ hinzu. Nun ist nur noch die Getter-Methode zu erstellen, die das Passwort als Hashwert zurückgibt.

```
5 Verweise
public class MD5Generator
{
    5 Verweise
    public static string getMD5Hash(string password)
    {
        StringBuilder sb = new StringBuilder();
        var md5provider = new MD5CryptoServiceProvider();
        // computing MD5 hash
        byte[] bytes = md5provider.ComputeHash(new UTF8Encoding().GetBytes(password));
        for (int i = 0; i < bytes.Length; i++)
        {
            sb.Append(bytes[i].ToString("x2"));
        }

        return sb.ToString();
    }
}
```

Sie können diese Methode anschließend in Ihrem Controller benutzen und die HttpPost-Login-Methode entsprechend anpassen. Binden Sie die benötigten „using-direktive“ im Controller ein.

```
0 Verweise
public async Task<IActionResult> Login(LoginModelView login)
{
    if (ModelState.IsValid)
    {
        var email = login.KdEmail;
        var password = MD5Generator.getMD5Hash(login.KdPwHash);

        //Kundendatensatz aus Datenbank-Kontext holen
        TblKunden customer = await _dbContext.TblKundens.FirstOrDefaultAsync(x => x.KdEmail == email && x.KdPwHash == password);

        if (customer != null)
        {
            //Aufrufen einer der Privacy-Seite als Test.
            //Hier müsste dann später auf das Dashboard des Kundenbereiches verwiesen werden
            return RedirectToAction(controllerName:"Home", actionName:"Privacy");
        }
    }
    //Bei Falschen Credentials wird die Login-Seite zurückgegeben
    return View(login);
}
```

Starten Sie das Projekt und testen Sie den Login-Prozess mit folgenden Eingabedaten:

- Email: hm@letsgo.net
- Passwort: Test123

3.6. Session mit Cookie erzeugen

Damit wir später den Benutzer im internen Kundenbereich verwalten können, ist es notwendig eine Cookie-Session zu erstellen, mit der die ID des Kunden verarbeitet werden kann. Dazu registrieren wir zunächst die Cookie-Session als Service in der „Startup.cs“-Datei, in dem wir folgenden Code in der „`public void ConfigureServices`“-Methode hinzufügen.

```
//Cookie als Service registrieren
services.AddSession(options =>
{
    options.Cookie.Name = "CustomerCookie";
    options.IdleTimeout = TimeSpan.FromMinutes(30); // oder die gewünschte Timeout-Zeit
});
```

Ebenfalls muss in der „`public void Configure`“-Methode die „`app.UseSession()`“-Methode angemeldet werden.

```
// Cookie Session
app.UseSession();
app.UseAuthorization();
```

Nun ist alles eingerichtet und Sie können in den einzelnen Controller-Methoden auf das Cookie zugreifen. In unserer Anwendung benötigen wir das Cookie nach erfolgreicher Anmeldung. D.h. wir übergeben in der Login-Methode dem Cookie hierbei die Kunden-ID sowie die Email-Adresse des Kunden. Sie könnten auch noch weitere Eigenschaften übergeben.

```
if (customer != null)
{
    //KundenId und Email in Session-Cookie hinzufügen
    //using Microsoft.AspNetCore.Http; einfügen
    HttpContext.Session.SetInt32("KundenID", customer.KdKundenId);
    HttpContext.Session.SetString("Email", customer.KdEmail);

    //Aufrufen der Privacy-Seite als Test.
    //Hier müsste dann später auf das Dashboard des Kundenbereiches verwiesen werden
    return RedirectToAction(controllerName:"Home", actionName:"Privacy");
}
```

Damit Sie die Funktionsweise der Cookie-Session testen können, erstellen Sie im Home-Controller unter der Privacy-Action folgenden Code.

```
0 Verweise
public IActionResult Privacy()
{
    ViewBag.kundenId = HttpContext.Session.GetInt32("KundenID");
    ViewBag.email = HttpContext.Session.GetString("Email");
    return View();
}
```

Hier ist auch beispielhaft dargestellt, wie Sie die Eigenschaften des Cookies abrufen können. Sie benötigen diese später, um einzelne Ansichten nur für den angemeldeten Benutzer sichtbar zu machen. Um das Ganze ebenfalls testen zu können, machen wir die Eigenschaften des Cookies sichtbar. Ändern Sie deshalb die Ansichtsdatei „Privacy.cshtml“ wie folgt ab. Alternativ könnten Sie auch eine eigene Seite bauen (z.B. das Dashboard des Kunden 😊)

```
@{  
    ViewData["Title"] = "Privacy Policy";  
}  
<h1>@ViewData["Title"]</h1>  
  
<p>Email: @ViewBag.email</p>  
<p>KundenID: @ViewBag.kundenId</p>
```

Starten Sie nun die Anwendung und überprüfen Sie die Funktionalität.

4. Hilfreiche Codefragmete zur Umsetzung des Projektes

4.1. Datensätze aus dem Kontext laden mit LINQ

LINQ (Abkürzung für Language Integrated Query) ist ein programmtechnisches Verfahren von Microsoft zum Zugriff auf Daten und im .NET Framework integriert. Mit LINQ-Anweisungen wollen wir Daten aus unserem Datenbank-Kontext an die View übergeben. Die Syntax ist dabei sehr ähnlich zu SQL.

Beim Login-Prozess haben Sie bereits die erste LINQ-Anweisung verwendet. Hier mussten Sie den Kunden mit den richtigen Anmeldedaten finden. Sehen wir uns dies nochmal im Detail an.

```
TblKunden customer = await _dbContext.TblKundens  
.FirstOrDefaultAsync(x => x.KdEmail == email && x.KdPwHash == password);
```

Sie greifen über den Datenbank-Kontext (`_dbContext`) auf die Tabelle Kunden zu und suchen sich den ersten Kunden raus (`.FirstOrDefaultAsync`), bei dem die Email und das Passwort übereinstimmen. Der Zugriff auf einzelne Attribute innerhalb des Models funktioniert mit der sogenannten Lambda-Expression (`x => x.Attribut`). Das Schlüsselwort „await“ bedeutet, dass Sie die Aktion asynchron ausführen.

Alternativ hätten Sie auch folgende Anweisung schreiben können, die ein identisches Ergebnis liefert.

```
TblKunden customer = await _dbContext.TblKundens  
    .Where(x => x.KdEmail == email && x.KdPwHash == password)  
    .FirstOrDefault();
```

Möchten Sie nun mehrere Tabellen miteinander verknüpfen, können Sie das Schlüsselwort „include“ in Verbindung mit der Methode „ToList()“ verwenden. In dem folgenden Codebeispiel wollen wir alle Events mit den dazugehörigen Veranstaltern und Kategorien als Liste darstellen.

```
var events = _dbContext.TblEvents
    .Include(t => t.EkEvKategorie)
    .Include(t => t.EvEvVeranstalter)
    .ToList();
```

Die „Include“-Anweisung funktioniert ähnlich wie die JOIN-Anweisung in SQL. Mit der Lambda-Expression innerhalb der „Include“-Anweisung wird angegeben, dass das Ergebnis der Abfrage auch die mit der Tabelle „EkEvKategorie“ verknüpften Daten mitliefern soll. D.h. Sie „verjoinen“ die Tabelle „Events“ mit der Tabelle „Kategorie“ über die Property „`public virtual TblEvKategorie EkEvKategorie { get; set; }`“.

Gleiches gilt für die Tabelle Veranstalter. Nun stellt sich die Frage, wo ist die Verbindung von Primär- und Fremdschlüssel. Dies ist ganz einfach zu beantworten, in dem wir uns das Model „TblEvents.cs“ ansehen und bemerken, dass die Beziehungseigenschaft bereits automatisiert über „Data-Annotations“ vom Entity-Framework vorgenommen wurde.

```
[ForeignKey(nameof(EkEvKategorieId))]
[InverseProperty(nameof(TblEvKategorie.TblEvents))]
11 Verweise
public virtual TblEvKategorie EkEvKategorie { get; set; }

[ForeignKey(nameof(EvEvVeranstalterId))]
[InverseProperty(nameof(TblEvVeranstalter.TblEvents))]
11 Verweise
public virtual TblEvVeranstalter EvEvVeranstalter { get; set; }
```

4.2. Daten in Datenbank updaten bzw. neuen Datensatz erstellen

Grundsätzlich gibt es drei Operationen, die auf Datensätze in der Datenbank ausgeführt werden können. Für unser Projekt benötigen Sie i.d.R. das Hinzufügen von neuen Datensätzen (Add-Methode) und das Updaten von vorhandenen Datensätzen (Update-Methode). Das Löschen von Datensätzen (Delete-Methode) ist in unserer Anwendung (aktuell) nicht notwendig.

Der Datenbankzugriff funktioniert grundsätzlich über das Model und dem dazugehörigen Datenbank-Kontext. Am Beispiel der Buchung eines neuen Events (Eventdatums) wird dies nun verdeutlicht.

Beim Erstellen von neuen Datensätzen benötigen Sie zunächst immer ein Objekt, das Sie neu erzeugen und anschließend mit Attributen befüllen.

```
TblBuchungen buchung = new TblBuchungen();
buchung.BuBezahlt = false;
buchung.BuStorniert = false;
buchung.BuRechnungErstellt = false;
buchung.EdEvDatenId = int.Parse(evID);
buchung.BuGebuchtePlaetze = int.Parse(anzahlPlaetze);
buchung.KdKundenId = kdID;
```

Bei der Buchung ist noch eine Besonderheit zu beachten, da die Plätze in der Tabelle Eventdatum manuell bearbeitet werden müssen. D.h. wir müssen anhand der Anzahl der gebuchten Plätze die aktuelle Teilnehmerzahl in der Tabelle „EventDaten“ updaten. Dazu holen wir uns den entsprechenden Datensatz aus dem Datenbank-Kontext und modifizieren die aktuelle Teilnehmerzahl.

```
TblEventDaten eventDatum = await _dbContext.TblEventDatens
    .Where(t => t.EdEvDatenId == int.Parse(evID)).FirstOrDefaultAsync();

eventDatum.EdAktTeilnehmer += int.Parse(anzahlPlaetze);
```

Nun können wir beide modifizierten Objekte unserem Datenbank-Kontext wieder übergeben und anschließend die Änderungen in der Datenbank vornehmen. Dazu sind folgende Anweisungen nötig.

```
_dbContext.TblBuchungen.Add(buchung);

_dbContext.TblEventDatens.Update(eventDatum);

await _dbContext.SaveChangesAsync();
```

Bei der Stornierung von Buchungen müssen Sie ähnlich vorgehen, um die Daten integer zu halten.

5. Quellen / Literatur

- <https://learn.microsoft.com/de-de/aspnet/core/mvc/overview?view=aspnetcore-5.0>
- <https://learn.microsoft.com/de-de/dotnet/architecture/maui/mvvm>
- <https://learn.microsoft.com/de-de/ef/core/>
- <https://learn.microsoft.com/de-de/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-5.0>