IPERKA DOKUMENTATION: SKI-SERVICE BACKEND (MODUL 295)

Von Mohamed Gebeili

17.02.2025

Inhalt

1 Informieren

1.1 1.1 Projektübersicht

Ziel dieses Projekts ist die Entwicklung eines **Backends für das Ski-Service Management** mit einer **Web-API** zur Verwaltung von Serviceaufträgen.
Die API wird mit **ASP.NET Core und Entity Framework** umgesetzt und nutzt **Microsoft SQL Server** als Datenbank.

1.2 1.2 Ausgangssituation

Ich habe mit der Entwicklung begonnen und konnte folgende **erste Komponenten umsetzen**:

- **Ein API-Projekt in ASP.NET Core erstellt.**
- **Datenbank mit Microsoft SQL Server eingerichtet.**
- **Datenbankstruktur mit Entity Framework Core erstellt (Code-First).**
- **Modelle für die Datenbank implementiert (Order, Service, User).**
- **API-Endpunkte für Registrierungen hinzugefügt.**

Ich bin jedoch noch nicht mit dem gesamten System fertig geworden, insbesondere gibt es noch Probleme mit der Authentifizierung und der vollständigen API-Integration.

2 2. Planen

2.1 2.1 Anforderungen

- **Umgesetzte Anforderungen**
  - Datenbankanbindung über Microsoft SQL Server.
  - Entity Framework Core als ORM (Code-First).
  - Erstellung der Grundstruktur für das API-Backend.
  - Modelle für Serviceaufträge, Services und Benutzer erstellt.
  - Swagger zur API-Dokumentation eingerichtet.
- **Noch nicht umgesetzte Anforderungen**
  - Authentifizierung mit JWT-Token für Benutzer.
  - Fehlende API-Endpunkte für vollständige CRUD-Operationen.
  - Validierung und Sicherheitsmaßnahmen fehlen.
  - Frontend-Anbindung zur API nicht getestet.

2.2 2.2 Zeitplan

Phase	**Status**
API-Grundstruktur	Teilweise umgesetzt
Datenbank	Abgeschlossen
API-Endpoints	Teilweise umgesetzt
Authentifizierung	Noch offen
Tests & Debugging	Noch offen

3 Entscheiden

Ich habe mich für folgende Technologien entschieden:

Bereich	Technologie
Backend	ASP.NET Core
Datenbank	Microsoft SQL Server
ORM	Entity Framework Core (Code-First)
Dokumentation	Swagger
Entwicklungstools	Visual Studio Code

Die Struktur für mein Projekt sieht bis jetzt wie folgt aus:

```
JetstreamAPI/
├── Controllers/
│   ├── RegistrationController.cs
│
├── Data/
│   ├── JetstreamDbContext.cs
│
├── Models/
│   ├── User.cs
│   ├── Order.cs
│   ├── Service.cs
│
├── Program.cs
├── appsettings.json
```

4 Realisieren

4.1 Umgesetzte Implementierung

Ich habe die **Datenbankstruktur mit Entity Framework Core** erstellt und folgende **Modelle implementiert**:

1 Order.cs (Auftragsmodell):

Ski-Service > JetstreamAPI > models > C# Order.cs > Order > Priority

```
1   using System;
2   using System.ComponentModel.DataAnnotations;
3   using System.ComponentModel.DataAnnotations.Schema;
4
5   namespace JetstreamAPI.Models
6   {
        4 references
7       public class Order
8       {
9           [Key]
            5 references
10          public int OrderID { get; set; }
11
12          [Required, MinLength(1)]
            0 references
13          public string CustomerName { get; set; } = string.Empty;
14
15          [Required, MinLength(1)]
            0 references
16          public string Email { get; set; } = string.Empty;
17
18          [Required, MinLength(1)]
            0 references
19          public string Phone { get; set; } = string.Empty;
20
21          [Required, MinLength(1)]
            0 references
22          public string Priority { get; set; } = string.Empty;
23
24          // Beziehung zu Service
25          [Required]
            0 references
26          public int ServiceID { get; set; }
27
28          [ForeignKey("ServiceID")]
            0 references
29          public virtual Service? Service { get; set; } = null!;
30
            0 references
31          public string Status { get; set; } = "Offen";
32
33  /*          // Beziehung zu Mitarbeiter (User)
34          public int? AssignedTo { get; set; }
35
36          [ForeignKey("AssignedTo")]
37          public virtual User? AssignedUser { get; set; } = null!;*/
38
            0 references
39          public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
40      }}
```

2 Service.cs (Dienstleistungsmodell)

Ski-Service > JetstreamAPI > models > C# Service.cs > ...

```
using System.ComponentModel.DataAnnotations;

namespace JetstreamAPI.Models
{
    2 references
    public class Service
    {
        [Key]
        0 references
        public int ServiceID { get; set; }

        [Required, MinLength(1)]
        0 references
        public string ServiceName { get; set; } = string.Empty;
    }
}
```

3 User.cs (Benutzermodell)

Ski-Service > JetstreamAPI > models > C# User.cs > ...

```
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;

namespace JetstreamAPI.Models
{
    2 references
    public class User : IdentityUser
    {
        0 references
        public string Role { get; set; } = "Mitarbeiter";
    }
}
```

4.2 4.2 Datenbankverbindung

Die Datenbank:

```
-- Erstellen der Datenbank
CREATE DATABASE JetstreamDB;
GO

USE JetstreamDB;
GO

-- Mitarbeiter-Tabelle (Login-System)
CREATE TABLE Users (
    UserID INT IDENTITY(1,1) PRIMARY KEY,
    Username NVARCHAR(50) UNIQUE NOT NULL,
    PasswordHash NVARCHAR(255) NOT NULL,
    Role NVARCHAR(20) NOT NULL DEFAULT 'Mitarbeiter' -- Z. B. Admin, Mitarbeiter
);

-- Tabelle für Service-Angebote
CREATE TABLE Services (
    ServiceID INT IDENTITY(1,1) PRIMARY KEY,
    ServiceName NVARCHAR(100) NOT NULL
);

-- Tabelle für Serviceaufträge
CREATE TABLE Orders (
    OrderID INT IDENTITY(1,1) PRIMARY KEY,
    CustomerName NVARCHAR(100) NOT NULL,
    Email NVARCHAR(100) NOT NULL,
    Phone NVARCHAR(20) NOT NULL,
    Priority NVARCHAR(20) CHECK (Priority IN ('Niedrig', 'Mittel', 'Hoch')) NOT NULL,
    ServiceID INT NOT NULL,
    Status NVARCHAR(20) CHECK (Status IN ('Offen', 'InArbeit', 'Abgeschlossen')) NOT NULL DEFAULT 'Offen',
    AssignedTo INT NULL, -- Mitarbeiter, der den Auftrag bearbeitet
    CreatedAt DATETIME DEFAULT GETDATE(),
    FOREIGN KEY (ServiceID) REFERENCES Services(ServiceID),
);

-- Standard-Services einfügen
INSERT INTO Services (ServiceName) VALUES
('Kleiner Service'),
('Grosser Service'),
('Rennski-Service'),
('Bindung montieren und einstellen'),
('Fell zuschneiden'),
('Heisswachsen');

-- Beispiel-Mitarbeiter hinzufügen (Passwort muss später gehasht werden)
INSERT INTO Users (Username, PasswordHash, Role) VALUES
('admin', 'admin123', 'Admin');

-- Benutzerrechte für Web-API (kein Zugriff mit root/sa)
CREATE LOGIN jetstreamUser WITH PASSWORD = 'SicheresPasswort123!';
CREATE USER jetstreamUser FOR LOGIN jetstreamUser;
GRANT SELECT, INSERT, UPDATE, DELETE ON Orders TO jetstreamUser;
GRANT SELECT ON Services TO jetstreamUser;
GRANT SELECT, INSERT, UPDATE, DELETE ON Users TO jetstreamUser;

select * from


use JetstreamDB

drop table Orders
```

Die **appsettings.json** enthält die Konfiguration für die **Datenbankverbindung**:

Ski-Service > JetstreamAPI > { } appsettings.json > ...

```
{
  "ConnectionStrings": {
    "JetstreamDB": "Server=STOLZERLIBANESE;Database=JetstreamDB;User=sa;Password=Edwin187;Encrypt=False;Tr
  },
  "Jwt": {
    "Secret": "SehrGeheimerJWTKey"
  }
}
```

In der Datei **JetstreamDbContext.cs** habe ich die **Datenbank-Entitäten definiert**:

Ski-Service > JetstreamAPI > Data > C# JetstreamDbContext.cs > ...

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using JetstreamAPI.Models;

namespace JetstreamAPI.Data
{
    6 references
    public class JetstreamDbContext : IdentityDbContext<User>
    {
        0 references
        public JetstreamDbContext(DbContextOptions<JetstreamDbContext> options) : base(options) { }


        0 references
        public DbSet<Order> Orders { get; set; }
        0 references
        public DbSet<Service> Services { get; set; }
    }
}
```

4.3 4.3 Erster API-Controller

Ich habe einen **ersten Controller für die Registrierung von Aufträgen** erstellt:

Ski-Service > JetstreamAPI > Controller > C# registrationController.cs > RegistrationController > _orders

```
namespace JetstreamAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    0 references
    public class RegistrationController : ControllerBase
    {
        // Simulierter In-Memory-Datastore
        7 references
        private static readonly List<Order> _orders = new List<Order>();

        /// <summary>
        /// Gibt alle Registrierungen zurück.
        /// </summary>
        [HttpGet]
        0 references
        public IActionResult GetRegistrations()
        {
            return Ok(_orders);
        }

        /// <summary>
        /// Gibt eine Registrierung anhand der ID zurück.
        /// </summary>
        [HttpGet("{id}")]
        1 reference
        public IActionResult GetRegistrationById(int id)
        {
            var order = _orders.FirstOrDefault(o => o.OrderID == id);
            if (order == null)
            {
                return NotFound();
            }
            return Ok(order);
        /// Fügt eine neue Registrierung hinzu.
        /// </summary>
        [HttpPost]
        0 references
        public IActionResult AddRegistration([FromBody] Order newOrder)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            // Simpler ID-Generator für In-Memory-Datenspeicher
            newOrder.OrderID = _orders.Any() ? _orders.Max(o => o.OrderID) + 1 : 1;
            _orders.Add(newOrder);
            return CreatedAtAction(nameof(GetRegistrationById), new { id = newOrder.OrderID }, newOrder);
        }

        /// <summary>
        /// Löscht eine Registrierung anhand der ID.
        /// </summary>
        [HttpDelete("{id}")]
        0 references
        public IActionResult DeleteRegistration(int id)
        {
            var order = _orders.FirstOrDefault(o => o.OrderID == id);
            if (order == null)
            {
                return NotFound();
            }

            _orders.Remove(order);
            return NoContent();
        }
```

5 Kontrollieren

5.1 Herausforderungen & Probleme

Ich bin noch nicht **fertig mit dem Projekt**, da folgende Probleme aufgetreten sind:

- **Datenbankverbindung:** Fehlerhafte Konfiguration, mehrfach angepasst.
- **Entity Framework Migrations-Probleme:** Ich musste Migrationen mehrfach zurücksetzen.
- **Swagger-Dokumentation war fehlerhaft.**
- **Authentifizierung mit JWT-Token funktioniert noch nicht.**
- **API-Routing in Program.cs war nicht korrekt.**

5.2 Fehlerbehebung

Ich habe verschiedene **Lösungsansätze getestet**:

- **Fehlende Migrationen erneut durchgeführt:**

dotnet ef migrations remove

dotnet ef migrations add InitialCreate

dotnet ef database update

- **Fehlende Pakete installiert:**

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

Trotz einiger Erfolge fehlen immer noch einige wichtige API-Funktionen.

6 Auswerten

6.1 Aktueller Stand

- **Datenbank erstellt & konfiguriert.**
- **Modelle (Order, Service, User) erstellt.**
- **Swagger hinzugefügt.**
- **Erste API-Endpunkte für Auftragsverwaltung programmiert**
- **JWT-Authentifizierung ist noch nicht funktionsfähig.**
- **Fehlende API-Routen für vollständige CRUD-Funktionalität.**
- **Fehlende Anbindung an ein Frontend.**

6.2 Reflexion

Ich habe **viel über Entity Framework, ASP.NET Core und API-Entwicklung gelernt**.
Allerdings gab es **mehr Probleme als erwartet**, besonders bei **Datenbankmigrationen** und der **API-Authentifizierung**.
Das Projekt ist noch nicht abgeschlossen, aber ich habe eine solide Grundlage für die weitere Entwicklung geschaffen.