1 MovieService mit CRUD-Operationen

> **Hinweis:** Dieser Auftrag setzt auf erledigtem Auftrag *03-AA-Minimal-API-MongoDB* auf.

1.1 Ziele

- Sie erstellen einen MovieService mit CRUD-Operationen für eine MongoDB
- Sie verwenden den MovieService in den Endpunkten

1.2 Umgebung

Die Übung wird auf der VM LP-22.04 durchgeführt.

1.3 Aufgaben

Aufgabe 1: MovieService | Einzelarbeit | 15'

Alle Zugriffe auf die MongoDb sollen nun in eine eigene Komonente ausgelagert werden. In ASP.NET Core werden dazu Services verwendet, die dank dependency injection in der ganzen App verfügbar sind. Siehe auch ASP.NET Core fundamentals.

Erstellen Sie dazu unter *minimal-api-with-mongodb/WebApi* ein neues File *MovieService.cs* mit folgendem Codegerüst:

```
using Microsoft.Extensions.Options;
using MongoDB.Driver;

public class MovieService
{
    // Constructor.
    // Settings werden per dependency injection übergeben.
    public MovieService(IOptions<DatabaseSettings> options)
    {
    }

    public string Check()
    {
        return "Zugriff auf MongoDB ...";
    }
}
```

Übernehmen Sie die Logik der Verbindungsprüfung aus *app.MapGet("/check" ...)* in *MovieService.Check()*.

Registrieren Sie den Service in WebApi/Program.cs als Singleton:

```
builder.Services.AddSingleton<MovieService>();
```

MovieService lässt sich nun in jede Map-Methode oder auch in jeden anderen Service injecten.

Passen Sie *app.MapGet("/check" ...)* so an, dass beim Aufruf von */Check* der neue Service aufgerufen wird:

```
app.MapGet("/check", (MovieService movieService) => {
    return movieService.Check();
});
```

Vergewissern Sie sich, dass der Aufruf von (http://localhost:5001/check) wieder zum gleichen Ergebnis führt, wie vor dem Refactoring.

Erweitern Sie nun alle app.Map-Methoden so, dass Sie den Service in der Methode zur Verfügung haben.

Aufgabe 2: CRUD-Operationen | Einzelarbeit | 15'

Attributieren Sie Property Id der Klasse Movie mit Attribut *BsonId*. Damit markieren Sie das Property *Id* als Repräsentation des MongoDB _id Attributs.

```
using MongoDB.Bson.Serialization.Attributes;

public class Movie
{
    [BsonId]
    public string Id { get; set; } = "";

    ...
}
```

Erweitern Sie *MovieService* mit folgendem Code-Gerüst.

```
public void Create(Movie movie)
{
    throw new NotImplementedException();
}

public IEnumerable<Movie> Get()
{
```

```
    throw new NotImplementedException();
}

public Movie Get(string id)
{
    throw new NotImplementedException();
}

public void Update(string id, Movie movie)
{
    throw new NotImplementedException();
}

public void Remove(string id)
{
    throw new NotImplementedException();
}
```

Passen Sie die Endpunkte in Program.cs so an, dass die entsprechende Service-Methode aufgrufen wird.

Implementieren Sie die Service-Methoden nun so, dass die Persistenz der Filme von der MongoDB übernommen wird. Verwenden Sie dazu eine Datenbank *gbs* mit einer Collection *movies*.

Eine gute Hilfestellung über den C# MongoDB-Treiber gibt die Quick-Referenz.

Mit folgendem Beispielcode wird ein Movie-Document in Collection *movies* der Datenbank *mydatabase* eingefügt.

```
var movie = new Movie()
{
    Id = "100",
    Title = "No Time to Die"
};

var mongoClient = new MongoClient(mongoDbConnectionString);
var database = _mongoClient.GetDatabase("mydatabase");
var movieCollection = database.GetCollection<Movie>("movies");

movieCollection.InsertOne(movie);
```

Aufgabe 3: Test | Einzelarbeit | 30’

Testen Sie die Anwendung sowohl mit *dotnet run* als auch mit *docker compose up*. Prüfen Sie jeweils alle Endpunkte mit Postman-Requests. Testen Sie nebst korrekten Reuqests immer auch Ausnahmesi-

tuationen, die zu einem Fehlercode führen.

Abbildung 1: Postman POST-Request

> **Hinweis** Bravo! Mit dem erfolgreichen Test der Anwendung haben Sie das Ende dieses kleinen Praxis-Projekts erreicht. :-)