C# Anleitung - REST + Klienten

Richard Bäck

2015-01-31 Sa

Contents

1 Vorwort

Diese Anleitung beschreibt, wie man eine Datenschicht, einen REST-Dienst, einen WPF-Klienten und ein ASP.NET MVC4 Klienten mit JavaScript anlegt. Als Beispiel dient eine Musikverwaltung mit Interpret, Album und Liedern.

2 Datenschicht

2.1 Lösung anlegen

1. Visual Studio starten
2. „FILE"
3. „New"
4. „Project. . . "
5. „Templates" → „Visual C#" → „Windows" → „Class Library"
6. Variablen setzen:
   (a) Name = MusicDataLayer
   (b) Location = Save/Path/For/Your/Solution
   (c) Solution name = MusicManager
   (d) Create directory for solution = true
7. „OK"

2.2 Entity Framework installieren

1. Rechtsklick auf die Lösung „MusicManager"
2. „Manage NuGet Packages"
3. „Online"
4. Nach „EntityFramework" suchen
5. Den ersten Vorschlag installieren
6. Den Dialog durchführen und danach schließen

2.3 Referenz zu System.Data.Entity setzen

1. Rechtsklick auf das Project „MusicDataLayer”
2. „Add Reference...”
3. „System.Data.Entity” anhacken
4. Dialog schließen

2.4 Entitätsmengen erstellen

1. Neuen Ordner stellen mit dem Namen „Entities”
2. Die generierte Datei umbennen auf „Entities.cs”
3. Die Datei muss nun berarbeitet werden und soll danach so aussehen:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicDataLayer.Entities
{
    public class Song
    {
        public int SongId { get; set; }
        public string Name { get; set; }
        public int Duration { get; set; } // in sec

        public int InterpreterId { get; set; }
        [ForeignKey("InterpreterId")]
        public virtual Interpreter Interpreter { get; set; }

        public int AlbumId { get; set; }
        [ForeignKey("AlbumId")]
        public virtual Album Album { get; set; }
    }
    public class Album
```

```
    {
        public int AlbumId { get; set; }
        public string Name { get; set; }
        public int Year { get; set; }
        public virtual ICollection<Song> Songs { get; set; }
    }
    public class Interpreter
    {
        public int InterpreterId { get; set; }
        public string Name { get; set; }
        public virtual ICollection<Song> Songs { get; set; }
    }
}
```

2.5 DbContext ableiten

1. Neuen Ordner erstellen mit dem Namen „Database"
2. Eine neue C#-Datei namens „MusicDbContext.cs" anlegen
3. Die generierte Datei in den neu erstellten Ordner verschieben
4. Die Datei sollte dann verändert werden, dass sie so aussieht:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicDataLayer
{
    public class MusicDbContext : DbContext
    {
        public DbSet<Song> Songs { get; set; }
        public DbSet<Album> Albums { get; set; }
        public DbSet<Interpreter> Interpreters { get; set; }

        public MusicDbContext() :
            base(
```

```
"Data Source=(localdb)\\v11.0;Initial Catalog=MusicDb;Integrated Security=True"
            )
        {
        }
    }
}
```

2.6 Repository anlegen

Ein „Repository" dient für einmalige Arbeiten an Objekten. Man kann ein „Update()"-Aufruf vom „Repository" also mit einem „update"-Aufruf in z.B. „sqlplus" mit „AUTOCOMMIT=ON" vergleichen.

1. Neuen Ordner namens „Repository" anlegen
2. Neue C#-Datei in den neu erstellten Ordner anlegen mit dem Namen „Repository"
3. In diese Klasse wird nun die Schnittstelle „IRepository" und die Klasse „Repository" geschrieben:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;
using System.Text;
using System.Threading.Tasks;

namespace MusicDataLayer
{
    public interface IRepository<T>
    {
        void Create(T obj);
        void Update(T obj);
        void Delete(T obj);

        // Parameter is a lambda object which gets passed to LINQ
        IEnumerable<T> Get(Expression<Func<T, bool>> query = null);
        T GetById(int id);
```

```
}

public class Repository<T, TContext> : IRepository<T>
    where T : class
    where TContext : DbContext, new()
{

    public void Create(T obj)
    {
        using (TContext ctx = new TContext())
        {
            ctx.Set<T>().Add(obj);
            ctx.SaveChanges();
        }
    }

    public void Update(T obj)
    {
        using (TContext ctx = new TContext())
        {
            ctx.Set<T>().Attach(obj);
            ctx.Entry<T>(obj).State = EntityState.Modified;
            ctx.SaveChanges();
        }
    }

    public void Delete(T obj)
    {
        using (TContext ctx = new TContext())
        {
            ctx.Set<T>().Attach(obj);
            ctx.Set<T>().Remove(obj);
            ctx.SaveChanges();
        }
    }

    public IEnumerable<T> Get(Expression<Func<T, bool>> query = null)
    {
        using (TContext ctx = new TContext())
        {
```

```
                IQueryable<T> result = ctx.Set<T>();
                if (query != null)
                    result = result.Where(query);
                return result.ToList();
            }
        }

        public T GetById(int id)
        {
            using (TContext ctx = new TContext())
            {
                return ctx.Set<T>().Find(id);
            }
        }
    }

    public class RepositoryMusic<T> : Repository<T, MusicDbContext>
            where T : class
    {
    }
}
```

2.7 Unit of Work

Eine „Unit of Work" dient zur Verarbeitung von Transaktionen. Man kann ein „Update()"-Aufruf aus der „Unit-of-Work" also mit einem „update"-Aufruf in z.B. „sqlplus" mit „AUTOCOMMIT=OFF" vergleichen. Es muss, um eine Transaktion erfolgreich zu beenden, immer „Save()" aufgerufen werden. Dies wird durch das Verwenden der „Unit of Work" mit dem Schlüsselwort „using" automatisch abgesetzt.

1. Neuer Ordner mit dem Namen „UnitOfWork"
2. Anlegen einer C#-Datei names „RepositoryUow.cs" in dem erstellten Ordner. Diese steuert alle Kommandos für die Datenbank.
3. „RepositoryUow.cs" soll nun so aussehen:

```
using System;
using System.Collections.Generic;
```

```
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace MusicDataLayer
{
    public class RepositoryUow<T> : IRepository<T>
        where T: class
    {
        private DbContext _context;

        public RepositoryUow(DbContext context)
        {
            _context = context;
        }

        public void Create(T obj)
        {
            _context.Set<T>().Add(obj);
        }

        public void Update(T obj)
        {
            if (_context.Entry<T>(obj).State == EntityState.Detached)
            {
                _context.Set<T>().Attach(obj);
            }
            _context.Entry<T>(obj).State = EntityState.Modified;
        }

        public void Delete(T obj)
        {
            if (_context.Entry<T>(obj).State == EntityState.Detached)
            {
                _context.Set<T>().Attach(obj);
            }
            _context.Set<T>().Remove(obj);
        }
```

```
        public IEnumerable<T> Get(Expression<Func<T, bool>> query = null)
        {
            IQueryable<T> result = _context.Set<T>();
            if (query != null)
                result = result.Where(query);
            return result.ToList();
        }

        public T GetById(int id)
        {
            return _context.Set<T>().Find(id);
        }
    }
}
```

4. Anlegen einer C#-Datei namens „UnitOfWork.cs" in dem erstellten Ordner. Diese steuert das Ende der Transaktion und somit das Festsetzen der Daten in einen konsistenten Zustand.

5. Die Datei „UnitOfWork.cs" soll nun so aussehen:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicDataLayer.UnitOfWork
{
    public interface IUnitOfWork : IDisposable
    {
        IRepository<T> GetRep<T>() where T : class;
        void Save();
    }

    public class MusicUnitOfWork : IUnitOfWork
    {
        private MusicDbContext db = new MusicDbContext();
        private bool disposed = false;
```

```
        public IRepository<T> GetRep<T>() where T : class
        {
            var rep = new RepositoryUow<T>(db);
            return rep as IRepository<T>;
        }

        public void Save()
        {
            db.SaveChanges();
        }

        private void Dispose(bool disposing)
        {
            if (!disposed)
            {
                if (disposing)
                    db.Dispose(); // free the locked ressources
            }
            disposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this); // suppress calling the destructor
        }
    }
}
```

2.8 Anlegen von DTOs

Um einen reibungslosen Datentransfer zu gewährleisten ist es ratsam DTOs (= „Data Transfer Objects" = Objekte für den Datentransfer) anzulegen.

1. Im Ordner „Entities" eine neue C#-Datei namens „DTOs.cs" anlegen
2. Diese sollte dann so aussehen:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicDataLayer
{
    public class SongDTO
    {
        public int SongId { get; set; }
        public string Name { get; set; }
        public int Duration { get; set; } // in seconds

        public int InterpreterId { get; set; }

        public int AlbumId { get; set; }
        public AlbumDTO Album { get; set; }
    }
    public class AlbumDTO
    {
        public int AlbumId { get; set; }
        public string Name { get; set; }
        public int Year { get; set; }
    }
    public class InterpreterDTO
    {
        public int InterpreterId { get; set; }
        public string Name { get; set; }
    }
}
```

2.9 Datenbank anlegen

1. „VIEW" → „Other Windows" → „Package Manager Console"

2. In das neu geöffnete Terminal „enable-migrations". Mit diesem Kommando wird „Migrations/Configuration.cs" genriert. In dieser Klasse wird die Datenbank mit der Methode „Seed()" befüllt.

3. „Migrations/Configuration.cs" sollte umgeändert werden, damit sie etwa so aussieht (= Einspielen von Testdaten):

```
namespace MusicDataLayer.Migrations
{
    using System;
    using System.Collections.Generic;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
       DbMigrationsConfiguration<MusicDataLayer.MusicDbContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(MusicDataLayer.MusicDbContext context)
        {
            // Delete old stuff
            context.Albums.RemoveRange(context.Albums.ToList());
            context.Songs.RemoveRange(context.Songs.ToList());
            context.Interpreters.RemoveRange(context.Interpreters.ToList());

            // Make some fresh stuff
            Interpreter inter1 = new Interpreter()
            {
                Name = "Gordon Goodwin Big Phat Band",
                Songs = new List<Song>()
                {
                    new Song()
                    {
                        Name = "Backrow Politics",
                        Duration = 180,
                        Album = new Album()
                        {
                            Name = "Album1", Year = 2015
                        }
                    },
                    new Song()
                    {
```

```
            Name = "The Jazz Police",
            Duration = 180,
            Album = new Album()
            {
                Name = "Album2", Year = 2014
            }
        }
    }
};

Album albumMingus = new Album()
{
    Name = "Mingus Album",
    Year = 2015
};
Interpreter inter2 = new Interpreter()
{
    Name = "Charles Mingus Big Band",
    Songs = new List<Song>()
    {
        new Song()
        {
            Name = "Moanin'",
            Duration = 180,
            Album = albumMingus
        },
        new Song()
        {
            Name = "Ecclusiastics",
            Duration = 180,
            Album = albumMingus
        },
        new Song()
        {
            Name = "Invisible lady",
            Duration = 180,
            Album = albumMingus
        }
    }
};
```

```
            Interpreter inter3 = new Interpreter()
            {
                Name = "Dave Brubeck",
                Songs = new List<Song>()
                {
                    new Song()
                    {
                        Name = "Take five",
                        Duration = 177,
                        Album = new Album()
                        {
                            Name = "Dave", Year = 1975
                        }
                    }
                }
            };

            // Populate the database with the fresh stuff
            context.Interpreters.AddOrUpdate(inter1);
            context.Interpreters.AddOrUpdate(inter2);
            context.Interpreters.AddOrUpdate(inter3);
            context.SaveChanges();
        }
    }
}
```

4. „add-migration" eingeben. Wenn eine Parameter Aufforderung kommt, einfach „MusicDb" eintippen. Mit diesem Kommando wird eine neue Klasse generiert, mit der die Datenbank dann schlussendlie erstellt werden kann.

5. „update-database" eingeben. Nun wird die Datenbank generiert und unter „C:/users/youruser/MusicDb.mdf" gespeichert.

2.10 Datebankverbindung

Für Testzwecke kann auch eine Verbindung zu der neu erstellten Datenbank aufgebaut werden.

1. „Server Explorer”
2. Rechtsklick auf „Data Connections”
3. „Add Connection. . . ”
4. „Microsft SQL Server Database File” auswählen und weiter
5. Die Verbindungszeichenkette „C:/users/youruser/MusicDb.mdf” eintragen

2.11 Ninject-Factory erstellen

1. Ninject installieren. Dies wird wie im Kapitel 2.2 abgewickelt.
2. Neuen Ordner namens „Factory” anlegen
3. Neue C#-Datei namens „Factory.cs” in dem Ordner anlegen
4. „Factory.cs” sollte dann so aussehen:

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicDataLayer
{
    public static class Factory
    {
        private static IKernel kernel;

        static Factory()
        {
            kernel = new StandardKernel();
            CreateBindings();
            //CreateMockBindings();
        }

        private static void CreateMockBindings()
        {
            //kernel.Bind<IUnitOfWork>()
```

```
            //    .To<MockUnitOfWork>();
            //kernel.Bind<IRepository<Song>>()
               //.To<MockRepository<Song>>()
               //.WithConstructorArgument("init", songs)
        }

        private static void CreateBindings()
        {
            kernel.Bind<IUnitOfWork>()
                .To<MusicUnitOfWork>();

            kernel.Bind<IRepository<Song>>().To<RepositoryMusic<Song>>();
            kernel.Bind<IRepository<Album>>().To<RepositoryMusic<Album>>();
            kernel.Bind<IRepository<Interpreter>>()
                  .To<RepositoryMusic<Interpreter>>();
        }

        public static T Get<T>()
        {
            return kernel.Get<T>();
        }
    }
}
```

3 REST-Dienst

3.1 Neues Project anlegen

1. Rechtsklick auf Lösung
2. „Add" → „New" → „Project…"
3. „Visual C#" → „Web" → „ASP.NET MVC 4 Web Application"
4. Variablen setzen:
   (a) Name = MusicService
5. „OK"
6. Ein neuer Dialog öffnet sich
7. „Web API" auswählen

8. „OK”

3.2 Referenz zur Datenschicht erstellen

1. Rechtsklick auf das Project „MusicService”
2. „Project Dependencies”
3. Bei der Liste „Depends on” das Project „MusicDataLayer” anhacken
4. Rechtsklick auf das Project „MusicService”
5. „Add reference”
6. „Solution” → „Projects”
7. Bei der Liste das Project „MusicDataLayer” anhacken
8. „OK”

3.3 Erstellen des SongController

Es gibt entweder schon einen Controller (z.B.: Controllers/ValuesController.cs), dann kann dieser benutzt werden (nur umbennen zu „SongController”) und es kann die nächste Aufzählung übersprungen werden. Sollte dieser Controller noch nicht existieren, kann man ihn erstellen durch:

1. Rechtsklick auf „Controllers”
2. „Add” → „Controller. . . ”
3. Variablen setzen:
   (a) Controller name = SongController
   (b) Template = API Controller with empty read/write actions
4. „Add”

Der „SongController” soll nun so aussehen:

```
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
```

```
using System.Net.Http;
using System.Web.Http;

namespace MusicService.Controllers
{
    public class SongController : ApiController
    {
        private IRepository<Song> rep = Factory.Get<RepositoryMusic<Song>>();

        private SongDTO createDTO(Song song)
        {
            return new SongDTO
            {
                SongId = song.SongId,
                Name = song.Name,
                Duration = song.Duration,
                AlbumId = song.AlbumId,
                InterpreterId = song.InterpreterId,
                //Album = new AlbumDTO
                //{
                //    AlbumId = song.Album.AlbumId,
                //    Name = song.Album.Name,
                //    Year = song.Album.Year
                //}
            };
        }

        private Song createModel(SongDTO songDTO)
        {
            return new Song
            {
                SongId = songDTO.SongId,
                Name = songDTO.Name,
                Duration = songDTO.Duration,
                AlbumId = songDTO.AlbumId,
                InterpreterId = songDTO.InterpreterId
            };
        }

        // GET api/song
```

```
        public IEnumerable<SongDTO> Get()
        {
            return rep.Get().Select(song => createDTO(song));
        }

        // GET api/song?interpreterid=5
        public IEnumerable<SongDTO> GetByInterpreterId(int interpreterid)
        {
            return rep.Get(song => song.InterpreterId == interpreterid)
                      .Select(song => createDTO(song));
        }


        // GET api/song/5
        public SongDTO Get(int id)
        {
            Song song = rep.GetById(id);
            return createDTO(song);
        }

        // POST api/song
        public void Post([FromBody]SongDTO value)
        {
            rep.Create(createModel(value));
        }

        // PUT api/song/5
        public void Put(int id, [FromBody]SongDTO value)
        {
            rep.Update(createModel(value));
        }

        // DELETE api/song/5
        public void Delete(int id)
        {
            rep.Delete(new Song { SongId = id });
        }
    }
}
```

3.4 Erstellen des InterpreterController

Es muss ein Controller wie im Kapitel 3.3 erstellt werden, nur mit dem Namen „InterpreterController". Die Quelldatei sollte dann so aussehen:

```
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace MusicService.Controllers
{
    public class InterpreterController : ApiController
    {
        private IRepository<Interpreter> rep =
            Factory.Get<RepositoryMusic<Interpreter>>();

        private InterpreterDTO createDTO(Interpreter interpreter)
        {
            return new InterpreterDTO
            {
                InterpreterId = interpreter.InterpreterId,
                Name = interpreter.Name
            };
        }

        private Interpreter createModel(InterpreterDTO interpreterDTO)
        {
            return new Interpreter
            {
                InterpreterId = interpreterDTO.InterpreterId,
                Name = interpreterDTO.Name
            };
        }

        // GET api/interpreter
        public IEnumerable<InterpreterDTO> Get()
        {
```

```
            return rep.Get().Select(i => createDTO(i));
        }

        // GET api/interpreter/5
        public InterpreterDTO Get(int id)
        {
            return createDTO(rep.GetById(id));
        }

        // POST api/interpreter
        public void Post([FromBody]InterpreterDTO value)
        {
            rep.Create(createModel(value));
        }

        // PUT api/interpreter/5
        public void Put(int id, [FromBody]InterpreterDTO value)
        {
            rep.Update(createModel(value));
        }

        // DELETE api/interpreter/5
        public void Delete(int id)
        {
            rep.Delete(new Interpreter { InterpreterId = id });
        }
    }
}
```

3.5 Erstellen des AlbumController

Es muss ein Controller wie im Kapitel 3.3 erstellt werden, nur mit dem Namen „AlbumController". Die Quelldatei sollte dann so aussehen:

```
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
```

```
using System.Web.Http;

namespace MusicService.Controllers
{
    public class AlbumController : ApiController
    {
        private IRepository<Album> rep =
            Factory.Get<RepositoryMusic<Album>>();

        private AlbumDTO createDTO(Album album)
        {
            return new AlbumDTO
            {
                AlbumId = album.AlbumId,
                Name = album.Name,
                Year = album.Year
            };
        }

        private Album createModel(AlbumDTO albumDTO)
        {
            return new Album
            {
                AlbumId = albumDTO.AlbumId,
                Name = albumDTO.Name,
                Year = albumDTO.Year
            };
        }

        // GET api/album
        public IEnumerable<AlbumDTO> Get()
        {
            return rep.Get().Select(a => createDTO(a));
        }

        // GET api/album/5
        public AlbumDTO Get(int id)
        {
            return createDTO(rep.GetById(id));
        }
```

```
        // POST api/album
        public void Post([FromBody]AlbumDTO value)
        {
            rep.Create(createModel(value));
        }

        // PUT api/album/5
        public void Put(int id, [FromBody]AlbumDTO value)
        {
            rep.Update(createModel(value));
        }

        // DELETE api/album/5
        public void Delete(int id)
        {
            rep.Delete(new Album { AlbumId = id });
        }
    }
}
```

4 WPF-Klient

4.1 Neues Project anlegen

1. Rechtsklick auf Lösung
2. „Add" → „New" → „Project…"
3. „Visual C#" → „Windows" → „WPF Application"
4. Variablen setzen:
   (a) Name = MusicClient
5. „OK"

4.2 Referenz zur Datenschicht erstellen

Siehe Kapitel 3.2.

4.3 RESTRepository

Mit dem „RESTRepository" findet eine Implementierung von „IRepository" statt, die es erlaubt „CRUD" über das Netzwerk auszuführen.

1. „System.Net.Http" wie in Kapitel 2.3 referenzieren - Einbindung für die Verwendung von „HttpClient"
2. „System.Web.Extensions" wie in Kapitel 2.3 - Einbindung für Serialisierung referenzieren
3. Neue C#-Datei „RESTRepository" anlegen
4. Diese soll dann etwa so aussehen:

```
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using System.Web.Script.Serialization;

namespace MusicClient
{
    public class RESTRepository
    {
        private HttpClient client;
        private JavaScriptSerializer serializer = new JavaScriptSerializer();

        public RESTRepository(string baseUrl)
        {
            client = new HttpClient { BaseAddress = new Uri(baseUrl) };
        }

        // e.g.: InterpreterDTO -> Interpreter
        private string RemoveDTO<T>()
        {
            var name = typeof(T).Name;
```

```
            name = name.Substring(0, name.Length - 3);
            return name;
        }

        public void Create<T>(T obj)
        {
            var json = serializer.Serialize(obj);
            var content = new StringContent(json, Encoding.UTF8, "application/json");
            client.PostAsync(RemoveDTO<T>(), content);
        }

        public void Update<T>(T obj, int id)
        {
            var json = serializer.Serialize(obj);
            var content = new StringContent(json, Encoding.UTF8, "application/json");
            client.PutAsync(RemoveDTO<T>() + "/" + id.ToString(), content);
        }

        public void Delete<T>(T obj, int id)
        {
            client.DeleteAsync(RemoveDTO<T>() + id.ToString());
        }

        public IEnumerable<T> Get<T>(string query = "")
        {

            var result = client.GetStringAsync(RemoveDTO<T>() + query).Result;
            return serializer.Deserialize<IEnumerable<T>>(result);
        }

        public T GetById<T>(int id)
        {
            var result = client.GetStringAsync(
                    RemoveDTO<T>() + "/" + id.ToString()).Result;
            return serializer.Deserialize<T>(result);
        }
    }
}
```

4.4 Das ViewModel

Das ViewModel ist das Bindeglied zwischen Datenverarbeitung und Datenpräsentation. XAML nimmt z.B. eine Liste von Liedern und zeigt diese an. Jedes Lied-Objekt (= „ViewModelManageSong") enthält Attribute, die auf das Lied-Datenobjekt zugreifen. Diese Attribute können von XAML aus abgerufen und zur Anzeige gebracht werden.

4.4.1 ViewModelBase

1. Neuen Ordner „ViewModel" erstellen
2. In dem Ordner „ViewModel" eine neue C#-Datei namens „ViewModelBase.cs" anlegen
3. „ViewModelBase.cs" (Hinweis: bleibt immer gleich!):

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicClient.ViewModel
{
    public class ViewModelBase : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        public virtual void OnPropertyChanged(string property)
        {
            if (this.PropertyChanged != null)
                this.PropertyChanged(this,
                new PropertyChangedEventArgs(property));
        }
    }
}
```

4.4.2 ViewModelManageSong

1. In dem Ordner „ViewModel" eine neue C#-Datei namens „RelayCommand.cs" anlegen

2. Die Datei sollte dann so umgeschrieben werden (Hinweis: die Klasse „RelayCommand" ist immer gleich!):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

namespace MusicClient.ViewModel
{
    public class RelayCommand : ICommand
    {
        private Action<object> execute;
        private Predicate<object> canExecute;

        public RelayCommand(Action<object> execute,
                            Predicate<object> canExecute = null)
        {
            this.execute = execute;
            this.canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            if (this.canExecute != null)
                return this.canExecute(parameter);
            return true;
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public void Execute(object parameter)
        {
            this.execute(parameter);
```

```
        }
    }
}
```

3. In dem Ordner „ViewModel" eine neue C#-Datei namens „ViewModel-ManageSong" anlegen, mit dem folgenden Inhalt:

```
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicClient.ViewModel
{
    public class ViewModelManageSong : ViewModelBase
    {
        // To estimate the right port run your project!
        private RESTRepository rest =
            new RESTRepository("http://localhost:6008/api/");

        private ViewModelBase vmParent;
        public ViewModelBase ViewModelParent
        {
            set { vmParent = value; }
        }

        private SongDTO song = new SongDTO();
        public SongDTO Song
        {
            get { return song; }
            set { song = value ?? new SongDTO(); }
        }

        public string Name
        {
            get { return song.Name; }
            set { song.Name = value; }
        }
```

```
public int Duration
{
    get { return song.Duration; }
    set { song.Duration = value; }
}

public int InterpreterId
{
    get { return song.InterpreterId; }
    set { song.InterpreterId = value; }
}

public int AlbumId
{
    get { return song.AlbumId; }
    set { song.AlbumId = value; }
}

public AlbumDTO Album
{
    get { return song.Album; }
    set { song.Album = value; }
}

// The next to properties enable editing those
// values in a user control (seeh ManageSongUserControl.xml)
public IEnumerable<AlbumDTO> Albums
{
    get
    {
        return rest.Get<AlbumDTO>();
    }
}

public IEnumerable<InterpreterDTO> Interpreters
{
    get
    {
        return rest.Get<InterpreterDTO>();
```

```
    }
}

public ICommand Save
{
    get
    {
        return new RelayCommand(
            p =>
            {
                rest.Update<SongDTO>(song, song.SongId);

                if (vmParent != null)
                    vmParent.OnPropertyChanged("Songs");
            }
        );
    }
}

public ICommand Delete
{
    get
    {
        return new RelayCommand(
            p =>
            {
                rest.Delete<SongDTO>(song, song.SongId);

                if (vmParent != null)
                    vmParent.OnPropertyChanged("Songs");
            }
        );
    }
}

public ICommand Create
{
    get
    {
        return new RelayCommand(
```

```
                    p =>
                    {
                        rest.Create<SongDTO>(song);

                        if (vmParent != null)
                            vmParent.OnPropertyChanged("Songs");
                    }
                );
            }
        }
    }
}
```

4.4.3 ViewModelList hinzufügen

Diese Klasse dient um Listen von Datenobjekten XAML bereitzustellen. Außerdem dient es als Speicher für in XAML selektierte Daten. Dabei löst eine Selektion der Daten mit der Methode „OnPropertyChanged()" eine andere Funktionskette aus.

1. In dem Ordner „ViewModel" eine neue C#-Datei namens „ViewModelSongList.cs" anlegen
2. Die Datei sollte dann so aussehen:

```
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MusicClient.ViewModel
{
    public class ViewModelSongList : ViewModelBase
    {
        private RESTRepository rest =
            new RESTRepository("http://localhost:24135/api/");

        public IEnumerable<InterpreterDTO> Interpreters
```

```
{
    get
    {
        return rest.Get<InterpreterDTO>();
    }
}

public IEnumerable<SongDTO> Songs
{
    get
    {
        return selectedInterpreter != null ?
            rest.Get<SongDTO>("?interpreterid=" +
              selectedInterpreter.InterpreterId.ToString()) :
            new SongDTO[0];
    }
}

// stores a selection coming from the XAML
private InterpreterDTO selectedInterpreter;
public InterpreterDTO SelectedInterpreter
{
    get { return selectedInterpreter; }
    set
    {
        selectedInterpreter = value;
        OnPropertyChanged("SelectedInterpreter");
        OnPropertyChanged("Songs");
    }
}

// stores a selection coming from the XAML
private SongDTO selectedSong;
public SongDTO SelectedSong
{
    get { return selectedSong; }
    set
    {
        selectedSong = value;
        OnPropertyChanged("SelectedSong");
```

```
                OnPropertyChanged("ViewModelManageSong");
            }
        }

        public ViewModelManageSong ViewModelManageSong
        {
            get
            {
                var vmpm = Factory.Get<ViewModelManageSong>();
                vmpm.Song = selectedSong;
                vmpm.ViewModelParent = this;
                return vmpm;
            }
        }
    }
}
```

4.5 XAML - MainWindow

1. Die „MainWindow.xaml" soll nun umgeändert werden. Anmerkung: Die XAML Datei bindet mit „local:ManageSongUserControl" eine andere XAML-Datei namens „ManageSongUserControl.xaml" ein. Diese wird im Kapitel 4.6 erstellt.

```
<Window x:Class="MusicClient.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:MusicClient.ViewModel"
        xmlns:local="clr-namespace:MusicClient"
        Title="MusicManager" Height="350" Width="525">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Label>Interpreters</Label>
            <ComboBox DisplayMemberPath="Name"
                      SelectedValuePath="InterpreterId"
                      ItemsSource="{Binding Interpreters}"
                      SelectedItem="{Binding SelectedInterpreter}">
             </ComboBox>
        </StackPanel>
        <local:ManageSongUserControl DockPanel.Dock="Right" Width="250"
```

```
                                        DataContext="{Binding ViewModelManageSong}">
        </local:ManageSongUserControl>
        <DataGrid AutoGenerateColumns="False" IsReadOnly="True"
                  ItemsSource="{Binding Songs}"
                  SelectedItem="{Binding SelectedSong}"
                  MouseDoubleClick="DataGrid_MouseDoubleClick">
            <DataGrid.Columns>
                <DataGridTextColumn Header="Name" Binding="{Binding Name}">
                </DataGridTextColumn>
                <DataGridTextColumn Header="Duration"
                                    Binding="{Binding Duration}">
                </DataGridTextColumn>
            </DataGrid.Columns>
        </DataGrid>
    </DockPanel>
</Window>
```

2. Die „MainWindow.xaml.cs" soll nun auch angepasst werden. Anmerkung: Diese C#-Klasse verwendet die Klasse „ManageSongDialog", die zu diesem Zeitpunkt noch nicht existiert. Diese wird erst im Kapitel 4.7 angelegt.

```
using MusicClient.ViewModel;
using MusicDataLayer;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace MusicClient
```

```
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext = Factory.Get<ViewModel.ViewModelSongList>();
        }

        private void DataGrid_MouseDoubleClick(object sender, MouseButtonEventArgs
        {
            ManageSongDialog dlg = new ManageSongDialog
            {
                DataContext =
                ((ViewModelSongList)DataContext).ViewModelManageSong
            };
            dlg.ShowDialog();
        }
    }
}
```

4.6 XAML - ManageSongUserControl

1. Beim Projekt „MusicClient" → „Add" → „New Item. . . "
2. „Visual C#" → „WPF" → „User Control (WPF)"
3. Variablen setzen:
   (a) Name = ManageSongUserControl
4. „Add"
5. Die neu erstellte „ManageSongUserControl.xaml" sollte nun so umgeändert werden:

```
<UserControl x:Class="MusicClient.ManageSongUserControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006
          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
          mc:Ignorable="d"
          d:DesignHeight="300" d:DesignWidth="300">
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>

    <Label Grid.Row="0" Grid.Column="0">Name</Label>
    <TextBox Grid.Row="0" Grid.Column="1" Margin="3"
             Text="{Binding UpdateSourceTrigger=PropertyChanged, Path=Name}">
    </TextBox>

    <Label Grid.Row="1" Grid.Column="0">Duration</Label>
    <TextBox Grid.Row="1" Grid.Column="1" Margin="3"
    Text="{Binding UpdateSourceTrigger=PropertyChanged, Path=Duration}">
    </TextBox>

    <Label Grid.Row="2" Grid.Column="0">Album</Label>
    <ComboBox Grid.Row="2" Grid.Column="1" Margin="3"
              DisplayMemberPath="Name" SelectedValuePath="AlbumId"
              ItemsSource="{Binding Albums}"
              SelectedValue="{Binding Song.AlbumId}">
    </ComboBox>

    <Label Grid.Row="3" Grid.Column="0">Interpreter</Label>
    <ComboBox Grid.Row="3" Grid.Column="1" Margin="3"
              DisplayMemberPath="Name" SelectedValuePath="InterpreterId"
              ItemsSource="{Binding Interpreters}"
              SelectedValue="{Binding Song.InterpreterId}">
```

```
        </ComboBox>

        <StackPanel Grid.Row="4" Grid.ColumnSpan="2"
                    Orientation="Horizontal" HorizontalAlignment="Right">
            <Button Margin="3" Command="{Binding Save}" Width="93">Save</Button>
            <Button Margin="3" Command="{Binding Delete}" Width="93">Delete</Butto
            <Button Margin="3" Command="{Binding Create}" Width="93">Add</Button>
        </StackPanel>
    </Grid>
</UserControl>
```

4.7 XAML - ManageSongDialog

1. Beim Projekt „MusicClient" → „Add" → „New Item. . . "
2. „Visual C#" → „WPF" → „Window (WPF)"
3. Variablen setzen:
   (a) Name = ManageSongDialog
4. „Add"
5. Die neu erstellte „ManageSongDialog.xaml" sollte nun so umgeändert werden:

```
<Window x:Class="MusicClient.ManageSongDialog"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MusicClient"
        Title="ManageSongDialog" Height="300" Width="300">
    <local:ManageSongUserControl></local:ManageSongUserControl>
</Window>
```

5 ASP.NET MVC4 Klient

5.1 Aufgabe

Die Aufgabenstellung ist eine Webapplikation zu erstellen, die dynamisch mit JavaScript Interpreten nach ihren Liedern abfragen kann. Danach sollte das Lied ausgewählt werden können und dessen Daten bearbeitet werden können. Dies alles soll innerhalb einer HTML-Datei passieren.

5.2 Projekterstellung

Es wird Einfachheit halber im bereits bestehenden Projekt „MusicService" weitergearbeitet. Es kann ansonsten auch ein neues Projekt hierfür wie im Kapitel 3.1 beschrieben angelegt werden.

5.3 Ändern der Steuerungnamens von „Home" auf „Music"

Mit der Erstellung des Projekts wurden die Dateien „Controllers/HomeController.cs" und „Views/Home/Index.cshtml" automatisch generiert. Somit kann der virtuelle URL-Pfad http://localhost/Home/Index angesprochen werden. Um nun „Home" mit „Music" zu ersetzen, muss folgendes erledigt werden:

1. Öffnen der Datei „Controllers/HomeController.cs"
2. Refakturierung des Klassennamens:
   (a) Rechtsklick auf die Zeichenkette „HomeController"
   (b) „Refactor" → „Rename"
   (c) Variablen setzen:
      i. New name = MusicController
   (d) „Ok"
3. Die Datei „Controllers/HomeController.cs" selbst zu „Controllers/MusicController.cs" umbennen
4. Den Ordner „Views/Home" zu „Views/Music" umbennen.

Nun kann der URL-Pfad http://localhost/Music/Index angesprochen werden.

5.4 Vorbereiten der Index.cshtml

Der Inhalt der Date „Views/Music/Index.cshtml" wird durch dem folgenden Code ersetzt:

```
<h2>Interpreterlist</h2>

<div>
    <label>Interpreters</label>
    <select id="selInterpreter"></select>
</div>
```

```
<div style="float:left; width:400px; min-height: 300px">
    <table id="tableSongs">
        <thead>
            <tr>
                <th>
                    Name
                </th>
                <th>
                    Duration
                </th>
            </tr>
        </thead>
    </table>
</div>

<div>
    <div>
        <label>Name</label>
        <input id="name" type="text" />
    </div>
   <div>
        <label>Duration</label>
        <input id="duration" type="text" />
    </div>
    <input id="saveButton" type="button" value="Save" />
</div>
```

5.5 Anlegen der JavaScript-Quelldatei

1. Rechtsklick auf den Ordner „Scripts”
2. „Add” → „New Folder” → benennen mit „MusicScripts”
3. Rechtsklick auf den neu erstellten Ordner
4. „Add” → „New Item. . . ” → „Web” → „JavaScript File”
5. Variablen setzen
   (a) Name = MusicController.js
6. „Ok”

5.6 JavaScript-Quelldatei als Skriptbündel registrieren

Um eine bestimmte JavaScript-Quelldatei an den Browserklienten senden zu können, muss es in der „BundleConfig.cs" registriert sein:

1. Öffnen der Datei „$\text{App}_{\text{Start}}$/BundleConfig.cs"
2. In der Methode „RegisterBundles()" muss nun der folgende Code eingefügt werden (am besten nach den Registrierungen von Bibliotheken → nach jquery):

```
bundles.Add(new ScriptBundle("~/bundles/music").Include(
                        "~/Scripts/MusicScripts/MusicController.js"));
```

5.7 Das Skriptbündel übertragen lassen

Abschließend muss das Skriptbündel noch bei einer Ansichtsdatei angegeben werden, damit diese auch tastächlich übertragen wird. Dies wird durch die Codezeile:

```
@Scripts.Render("~/bundles/music")
```

ermöglicht. Wichtig ist aber, wo diese Zeile plaziert wird. Dazu gibt es zwei mögliche Übertragungsarten:

1. Skript nur für eine Ansichtsdatei übertragen: Für dies wird die Zeile in der speziellen Ansichtsdatei (hier: „Views/Music/Index.cshtml") plaziert.

2. Skript global für alle Ansichtsdateien übertragen: Für dies wird die Zeile in der Ansichtsdatei „Views/Share/$_Layout.cshtml''innerhalbdesHTML - Abgrenzers,,body''eingetragen$. Auf jeden Fall ist die Reihenfolge der Skriptbündel zu beachten! Wenn zB jQuery eine Abhängigkeit von meiner eigenen JavaScript-Datei ist, dann muss zuerst jQuery eingebunden werden. Daraus lässt sich schließen, dass benötigte Bibliotheken als ersters und die Hauptfunktion als letzters eingebunden werden muss.

   Für dieses Beispiel würde das bei den Dateien nun so aussehen:

   - _Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    @Scripts.Render("~/bundles/jquery")

    @RenderBody()

    @RenderSection("scripts", required: false)
</body>
</html>
```

- Index.cshtml

```
@Scripts.Render("~/bundles/music")

<h2>Interpreterlist</h2>

<div>
    <label>Interpreters</label>
    <select id="selInterpreter"></select>
</div>

<div style="float:left; width:400px; min-height: 300px">
    <table id="tableSongs">
        <thead>
            <tr>
                <th>
                    Name
                </th>
                <th>
                    Duration
                </th>
            </tr>
```

```
            </thead>
        </table>
    </div>

    <div>
        <div>
            <label>Name</label>
            <input id="name" type="text" />
        </div>
       <div>
            <label>Duration</label>
            <input id="duration" type="text" />
        </div>
        <input id="saveButton" type="button" value="Save" />
    </div>
```

5.8 Die JavaScript-Quelldatei

5.8.1 Die gesamte Quelldatei

Es wird nun nacheinander abgearbeitet, wie der Code die HTML-Datei beeinflusst bzw. wie der Code zusammenspielt. Die gesamte Quelldatei sieht dann wiefolgt aus:

```
var selectedSong = null;
var baseurl = 'http://localhost:6008/api/';

$(function () {
    loadInterpreters('#selInterpreter');
    $('#selInterpreter').change(function () {
        loadSongs(this.value);
    });
    $('#saveButton').click(function () {
        selectedSong.Name = $('#name').val();
        selectedSong.Duration = $('#duration').val();
        saveSong(selectedSong);
    });
})

function loadInterpreters(select) {
    $.getJSON(baseurl + 'interpreter', function (data) {
```

```
        $.each(data, function (index, item) {
            $('<option value = ' +
              item.InterpreterId +
              '>' +
              item.Name +
              '</option>')
                .appendTo(select);
        });
    });
}

function loadSongs() {
    $('#tableSongs tbody tr').remove();
    $.getJSON(baseurl + 'song?interpreterid=' + $('#selInterpreter').val(),
      function (data) {
        $.each(data, function (index, item) {
            $('<tr><td>' + item.Name +
                '</td><td>' + item.Duration +
                '</td><td>' + //item.Ablum.Name +
                '</td></tr>')
                .appendTo('#tableSongs')
                .hover(function () {
                    $(this).css('background-color', 'lightblue');
                }, function () {
                    $(this).css('background-color', '');
                })
                .click(function () {
                    editSong(item);
                });
        });
    });
}

function editSong(song) {
    selectedSong = song;
    $('#name').val(song.Name);
    $('#duration').val(song.Duration);
}

function saveSong(song) {
```

```
    var url = baseurl + 'song/' + song.SongId;
    $.ajax({
        url: url,
        type: 'PUT',
        data: song,
        success: function () {
            loadSongs($('#selInterpreters').val());
        },
        error: function (xhr, status, msg) {
            alert(msg);
        }
    });
}
```

5.8.2 Die Konstanten

```
var selectedSong = null;
var baseurl = 'http://localhost:6008/api/';
```

selectedSong Dies ist das aktive Lied, das von einem Interpreten ausgewählt ist und editiert werden kann.

baseurl Dies ist die Basis URL, von der Webdienststelle.

5.8.3 Laden von Interpreten

Es wird das JSON-Objekt für die Liste von Interpreten geladen. In der Zeile 4 wird dann ein neuer HTML-Abgrenzer erzeugt und danach an eine HTML-Liste angehängt:

```
function loadInterpreters(select) {
    $.getJSON(baseurl + 'interpreter', function (data) {
        $.each(data, function (index, item) {
            $('<option value = ' +
              item.InterpreterId +
              '>' +
              item.Name +
              '</option>')
                .appendTo(select);
        });
    });
}
```

5.9 Laden von Liedern

Es wird zuerst die HTML-Tabelle für die Lieder gelöscht (Zeile 2). Danach wird wieder ein JSON-Objekt mit der Liste von Liedern zu dem ausgewählten Interpreten im HTML-Abgrenzer „selInterpreter". Schlussendlich wird die Liste durchgelaufen und jedes Lied nacheinander der Tabelle als Tabellenzeile angehängt. In Zeile 11 bis 15 folgt eine Stileinbettung statt. Mit der Funktion in Zeile 16 und folgende wird bei einem Anklicken eines Liedes automatisch die Funktion „editSong()" aufgerufen.

```
function loadSongs() {
    $('#tableSongs tbody tr').remove();
    $.getJSON(baseurl + 'song?interpreterid=' + $('#selInterpreter').val(),
      function (data) {
        $.each(data, function (index, item) {
            $('<tr><td>' + item.Name +
                '</td><td>' + item.Duration +
                '</td><td>' + item.Ablum.Name +
                '</td></tr>')
                .appendTo('#tableSongs')
                .hover(function () {
                    $(this).css('background-color', 'lightblue');
                }, function () {
                    $(this).css('background-color', '');
                })
                .click(function () {
                    editSong(item);
                });
        });
    });
}
```

5.10 Die Liededitierung

Zuerst wird das aktuell ausgewählte Lied gesetzt. Dies wird später für die Speicherung der Änderungen benötigt. Mit der folgenden Funktion wird ein ausgewähltes Lied in das Formular geladen.

```
function editSong(song) {
    selectedSong = song;
    $('#name').val(song.Name);
```

```
    $('#duration').val(song.Duration);
}
```

5.11 Die Speicherung

Die folgende Funktion ruft die REST-URL zur Aktualiserung eines Liedes auf. Bei einer erfolgreichen HTTP-Verbindung wird anschließend die Liederliste neu geladen, um die Änderung gleich für den Benutzer anzuzeigen (Zeile 8). Sollte ein Fehler auftreten wird die Fehlermeldung ausgegeben (Zeile 11).

```
function saveSong(song) {
    var url = baseurl + 'song/' + song.SongId;
    $.ajax({
        url: url,
        type: 'PUT',
        data: song,
        success: function () {
            loadSongs($('#selInterpreters').val());
        },
        error: function (xhr, status, msg) {
            alert(msg);
        }
    });
}
```

5.11.1 Die Hauptfunktion

Die Hauptfunktion stellt den Einsprungspunkt für den ganzen JavaScript-Code der Seite dar. Außerdem verbindet es die ganzen Funktionen miteinander. In der Zeile 2 wird zuerst die Liste der Interpreten gefüllt. Mit der Funktionsverknüpfung in Zeile 3 erfolgt ein automatisches Laden der Lieder eines Interpreten, wenn dieser in der Interpetenliste ausgewählt wird. Zuletzt wird mit der Funktionsverknüpfung in Zeile 6 gewährleistet, dass das drücken des Speichernknopfes die veränderten Daten von HTML in das JavaScript Liederobjekt geschrieben werden und dann die Aktualisierungsfunktion aufgerufen wird.

```
$(function () {
    loadInterpreters('#selInterpreter');
    $('#selInterpreter').change(function () {
```

```
        loadSongs(this.value);
    });
    $('#saveButton').click(function () {
        selectedSong.Name = $('#name').val();
        selectedSong.Duration = $('#duration').val();
        saveSong(selectedSong);
    });
})
```

6 Fehlerlösungen

6.1 Update-Database funktioniert nicht

1. „CTRL + R" → „cmd" → „OK"
2. SqlLocalDb stop v11.0
3. SqlLocalDb delete v11.0