Washington State University Vancouver

SolarWorld Capstone Project

Server Side Development

Author: Tyler Vanderhoef

Contents

February 11, 2017

1 Modifying The Database

1.1 Defining A New Table In Code

Using Entity Framework we are able to define the database tables using C#classes. To add a new database table, all you need to do is add a new class to the *Backend.Schemas* project. Because the code is mapped to a table, there are some standards that must be followed when adding a new class. Below is a list of the standards/mappings between the C# class and the database table. Listing 1 displays how a table may be defined in code.

1. Class Name: The name of the database table.
2. Properties: The name of the columns contained in the table.
3. Property Attributes: These tags dictate *how* the columns are to be defined. Things such as Keys, Foreign Keys, Nullables, Uniques, and Indexes are defined in this way.
4. Virtual Properties: These properties map to a foreign key table. They are auto populated when making database queries.
5. Virtual ICollection Properties: These objects are a collection of table records. ICollection's mean that there is at least a one-to-many mapping between to entities.

```csharp
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Backend.Schemas
{
    public class Region
    {
        //Specifies that this is the primary key, and that it
        //must be unique/indexed.
        [Key]
        [Index(IsUnique = true)]
        public string Name { get; set; }
    }
}
```

Listing 1: Example of a Regions table with a single column called Name.

1.1.1 Adding The Table To The Database

Once you have created your new class like in the previous section, you must tell Entity Framework that you want it added to the database. In order to do this, you need to modify the *DatabaseContext.cs* file located in the *Backend.Schemas* project to have a new IDbSet property of your new class. An example of this can be seen in Listing 2 below.

```
using System.Data.Entity;

namespace Backend.Schemas
{
    /// <summary>
    /// Represents the database tables.
    /// </summary>
    public sealed class DatabaseContext : DbContext
    {
        public DatabaseContext(string nameOrConnectionString)
            : base(nameOrConnectionString)
        {

        }

        public DatabaseContext() : this("DevDatabase") { }

        //add this line to add a new table named Regions using the
        //columns as specified in the Region class.
        public IDbSet<Region> Regions { get; set; }
    }
}
```

Listing 2: Example of adding a new table in the database.

1.2 Updating Existing Tables

Updating existing tables is easy. Just modify the class file and follow the steps outlined for committing changes to the database.

1.3 Committing The Changes To The Database

Once you have made all your changes and are ready to commit them to the database, you only have one more step. To do this, open up the Nuget Package Manager in Visual Studio by navigating to Tools → NuGet Package Manager → Package Manager Console. In the drop down named Default Project, select the *Backend.Schemas* project. The commands to run are listed below. Then, run the command: add-migration AddRegions.

1. Add-Migration MigrationName → Calculates changes to apply to the database.

- The MigrationName is any meaningful name like AddedRegions.
- An example would be: Add-Migration AddedRegions.

2. Update-Database → Commits the changes in the migration to the database.

The first command will create a migration file that shows you what changes will be committed (in the Up() method) to the database. If all the changes look fine, commit it by running second command. Make sure that the parameter after add-migration is a meaningful name. This name is how you can revert database changes if needed.

2 Adding Repositories

All the objects mentioned below can be found in the *Backend.DataAccess* project in their respective folders.

2.1 Adding DataSources

DataSources are the objects that communicate directly with the database. They serve as a barrier between the database objects and the shared models. Typically, the DataSources are only consumed by a Repository object (more on that later). As an example Listing 3 demonstrates what a DataSource typically will look like. Note that this does not have all the code implemented, however the required methods are built out.

```
using Backend.DataAccess.Abstractions;
using Backend.Schemas;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace Backend.DataAccess.Repositories.DataSources
{
    internal sealed class RegionDataSource : AbstractDataSource<Region>
    {
        public RegionDataSource(DatabaseContext ctx) : base(ctx) {}
        public override Task<Region> DeleteAsync(Region toDelete)
        {
            throw new NotImplementedException();
        }
        public override IQueryable<Region> Get()
        {
            return Database.Regions;
        }
        public override IOrderedQueryable<Region> GetOrdered()
        {
            // Return Regions ordered by name.
            return Database.Regions.OrderBy(r => r.Name);
        }
        public override Task<Region> GetSingleAsync()
        {
            //Get a single region
            throw new NotImplementedException();
        }
        public override Task<Region> InsertAsync(Region toCreate)
        {
            //Insert a new region
            throw new NotImplementedException();
        }
        public override Task<Region> UpdateAsync(Region toUpdate)
        {
            //Update an existing region
            throw new NotImplementedException();
        }
    }
}
```

Listing 3: Example Regions DataSource.

Stepping through the code, there are some important things to note:

1. The DataSource is internal. This is so that we constrain them only to the project, and only Repositories know how to use them.
2. The DataSource inherits from *AbstractDataSource<Region>*. This class defines all the standard methods required.

Each of these functions will do work on the database. It does not do anything with shared Models, and any results it returns will be up to the Repository to figure out what to do with the results.

2.2 Creating Repositories

Repositories act as the way to take Models and convert them to their appropriate database table object. Each Repository will contain an internal DataSource which it will use to perform all data operations. Listing 4 demonstrates what a Repository will look like.

Notice how the *DatabaseContext* object gets passed into the constructor, and then into the DataSource. This is done because it allows for testing of any type of database. Unimplemented methods were moved for brevity. Notice we have a BuildViewModel method that takes a database model and converts it to the shared model type. Likewise, there is a shared model to database model method.

```
namespace Backend.DataAccess.Repositories
{
    public sealed class RegionRepository
        : AbstractRepository<RegionModel, Region>
    {
        public RegionRepository(DatabaseContext database)
            : base (new RegionDataSource(database)) {}

        public override async Task<IEnumerable<RegionModel>> GetAsync()
        {
            //TODO: Remove this sample implementation with a real one.
            return await DataSource
                //Get the records in order
                .GetOrdered()
                //convert records to shared model
                .Select(r => new RegionModel
                {
                    IsAdmin = true
                })
                //load the data
                .ToListAsync();
        }
        protected override RegionModel BuildViewModel(Region model)
        {
            /*
             * This is where we will actually convert from a
             * region object to a shared region model object.
             */
            return new RegionModel { Name = model.Name };
        }
        protected override Region BuildModel(RegionModel model)
        {
            /*
             * This is where we will actually convert from a
             * region shared model object to a region database object.
             */
            return new Region { Name = model.Name };
        }
    }
}
```

Listing 4: Example code for a Repository.

3 Adding A New Controller

Controllers are the main entry point from the REST API urls. To add a new url route to the server, add a new file to the *Controllers* folder. Make sure to name the controller something meaningful so it is obvious what routes are contained within. Listing 1 demonstrates the minimum code required to setup a new GET endpoint. The method names are used to determine what type of endpoint it is, such as GET, POST, PUT, DELETE, etc...

Below is a high level overview of the way to add a new Controller/API endpoint.

1. Create a new database table if need be.
2. Create a new DataSource object for said database table.
3. Create a new Repository object using newly created DataSource.
4. Create the new Controller.
5. Add a readonly property for the new Repository in the Controller.
6. Instantiate a new instance of the Repository in the constructor of the Controller.
7. Build methods in the Controller using the Repository for getting/modifying data.

```
using System.Threading.Tasks;
using System.Web;
using System.Web.Http;
using Backend.DataAccess.Abstractions;
using Backend.DataAccess.Repositories;
using Backend.Schemas;
using Mobile_Rounds.ViewModels.Models;

namespace Backend.Controllers
{
    // Controller represents the /api/regions endpoints
    // Lock down the endpoint to require Windows authentication
    [RoutePrefix("api/regions")]
    [Authorize]
    public class RegionsController : ApiController
    {
        // uses the Interface and not the class name because it is more
        // scaleable that way.
        private readonly IRepository<RegionModel> datasource;

        public RegionsController(DatabaseContext database)
        {
            //create our internal repository for fetching data
            this.datasource = new RegionRepository(database);
        }

        //a blank route means this method handles the /api/regions request
        //the method name Get() means it handles GET requests only.
        [Route("")]
        public async Task<IHttpActionResult> Get()
        {
            //fetch data from the database.
            var results = await this.datasource.GetAsync();

            //return the results
            return this.Ok(results);
        }
    }
}
```

Listing 5: Example code for a controller with a single GET endpoint.