

WASHINGTON STATE UNIVERSITY VANCOUVER

SOLARWORLD CAPSTONE PROJECT

Server Side Development

Author: Tyler Vanderhoef

Contents

| | | |
|----------|--|----------|
| 1 | Modifying The Database | 1 |
| 1.1 | Defining A New Table In Code | 1 |
| 1.1.1 | Adding The Table To The Database | 1 |
| 1.2 | Updating Existing Tables | 2 |
| 1.3 | Committing The Changes To The Database | 2 |
| 2 | Adding Repositories | 3 |
| 2.1 | Adding DataSources | 3 |
| 2.2 | Creating Repositories | 5 |
| 3 | Adding A New Controller | 7 |

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.

```
1  using System.ComponentModel.DataAnnotations;
2  using System.ComponentModel.DataAnnotations.Schema;
3
4  namespace Backend.Schemas
5  {
6      public class Region
7      {
8          //Specifies that this is the primary key, and that it
9          //must be unique/indexed.
10         [Key]
11         [Index(IsUnique = true)]
12         public string Name { get; set; }
13     }
14 }
```

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.

```
1  using System.Data.Entity;
2
3  namespace Backend.Schemas
4  {
5      /// <summary>
6      /// Represents the database tables.
7      /// </summary>
8      public sealed class DatabaseContext : DbContext
9      {
10         public DatabaseContext(string nameOrConnectionString)
11             : base(nameOrConnectionString)
12         {
13
14         }
15
16         public DatabaseContext() : this("DevDatabase") { }
17
18         //add this line to add a new table named Regions using the
19         //columns as specified in the Region class.
20         public IDbSet<Region> Regions { get; set; }
21     }
22 }
```

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 view 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.

```
1  using Backend.DataAccess.Abstractions;
2  using Backend.Schemas;
3  using System;
4  using System.Linq;
5  using System.Threading.Tasks;
6
7  namespace Backend.DataAccess.Repositories.DataSources
8  {
9      internal sealed class RegionDataSource : AbstractDataSource<Region>
10     {
11         public RegionDataSource(DatabaseContext ctx) : base(ctx) {}
12         public override Task<Region> DeleteAsync(Region toDelete)
13         {
14             throw new NotImplementedException();
15         }
16         public override IQueryable<Region> Get()
17         {
18             return Database.Regions;
19         }
20         public override IOrderedQueryable<Region> GetOrdered()
21         {
22             // Return Regions ordered by name.
23             return Database.Regions.OrderBy(r => r.Name);
24         }
25         public override Task<Region> GetSingleAsync()
26         {
27             //Get a single region
28             throw new NotImplementedException();
29         }
30         public override Task<Region> InsertAsync(Region toCreate)
31         {
32             //Insert a new region
33             throw new NotImplementedException();
34         }
35         public override Task<Region> UpdateAsync(Region toUpdate)
36         {
37             //Update an existing region
38             throw new NotImplementedException();
39         }
40     }
41 }
```

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 view 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 `View Models` and convert them to their appropriate database model. 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 view model type. Likewise, there is a `view model to database model` method.

```
1 namespace Backend.DataAccess.Repositories
2 {
3     public sealed class RegionRepository
4         : AbstractRepository<RegionsViewModel, Region>
5     {
6         public RegionRepository(DatabaseContext database)
7             : base (new RegionDataSource(database)) {}
8
9         public override async Task<IEnumerable<RegionsViewModel>> GetAsync()
10        {
11            //TODO: Remove this sample implementation with a real one.
12            return await DataSource
13                //Get the records in order
14                .GetOrdered()
15                //convert records to view models
16                .Select(r => new RegionsViewModel
17                {
18                    IsAdmin = true
19                })
20                //load the data
21                .ToListAsync();
22        }
23        protected override RegionsViewModel BuildViewModel(Region model)
24        {
25            /*
26            * This is where we will actually convert from a
27            * region object to a region view model object.
28            */
29            return new RegionsViewModel { IsAdmin = true };
30        }
31        protected override Region BuildModel(RregionsViewModel model)
32        {
33            /*
34            * This is where we will actually convert from a
35            * region view model object to a region database object.
36            */
37            return new Region { Name = "Name from model parameter." };
38        }
39    }
40 }
```

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.


```
1  using System.Threading.Tasks;
2  using System.Web;
3  using System.Web.Http;
4  using Backend.DataAccess.Abstractions;
5  using Backend.DataAccess.Repositories;
6  using Backend.Schemas;
7  using Mobile_Rounds.ViewModels.Admin.Regions;
8
9  namespace Backend.Controllers
10 {
11     // Controller represents the /api/regions endpoints
12     // Lock down the endpoint to require Windows authentication
13     [RoutePrefix("api/regions")]
14     [Authorize]
15     public class RegionsController : ApiController
16     {
17         // uses the Interface and not the class name because it is more
18         // scalable that way.
19         private readonly IRepository<RegionsViewModel> datasource;
20
21         public RegionsController(DatabaseContext database)
22         {
23             //create our internal repository for fetching data
24             this.datasource = new RegionRepository(database);
25         }
26
27         //a blank route means this method handles the /api/regions request
28         //the method name Get() means it handles GET requests only.
29         [Route("")]
30         public async Task<IHttpActionResult> Get()
31         {
32             //fetch data from the database.
33             var results = await this.datasource.GetAsync();
34
35             //return the results
36             return this.Ok(results);
37         }
38     }
39 }
```

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