

Web API powerful Custom Model Validation with FluentValidation

BY [CHRISTOS S.](#) on [JANUARY 17, 2015](#) ·  (2)

Since **DataAnnotations** were introduced, developers found an easy way for setting up their database (especially those who use Code First development) while adding at the same time **validation** logic to their domain objects. In the Web API side, when an object is posted to one of it's actions, you can check if the object posted is valid with something like this..

```
1  if (!ModelState.IsValid)
2      {
3          return BadRequest(ModelState);
4      }
5  else
6      {
7          //
8      }
```

While all these sounds great, what happens when you have to add custom validation to your objects? What happens, if you want a **User** object posted with a FirstName property to be marked as Invalid when it's first letter is not capital? Can you do this with Attributes? This very question reveals the real problem here, that is it's not your domain models that needs to be validated but the **ViewModel**, the one that is transferred between the client and the server. Some times, your ViewModel objects may be the same with your domain objects which is something I always try to avoid. Anyway, I thought it would be good to post about this and make things a little bit clearer for those who are confused.

A clean solution to this problem is to split the logic where you define your database configuration (with DataAnnotation attributes or not) and the logic that validates your ViewModel objects. Here's what we 're gonna see on this post:

1. **Set up the database** (Code First development)
2. **Add database configuration using** Fluent API
3. **Add custom validation configuration using** FluentValidation

For this solution I chose on purpose not to use custom ViewModel objects so that you can see that **model validation** can be much more things than database configuration. Let's start.

I created a new black solution in VS 2013 and added three projects:

1. **Domain** (Class Library)
2. **Data Access** (Class Library)
3. **Web API** Empty Web Application with MVC / Web API templates checked

Suppose that we want to create a Registration form so that user can sign up to our application. Starting from the Domain class library project add the following classes.

User.cs

```
1  public class User
2  {
3      public int ID { get; set; }
4
5      public string FirstName { get; set; }
6
7      public string LastName { get; set; }
8
9      public string Username { get; set; }
10
11     public string Password { get; set; }
12
13     public string BirthDate { get; set; }
14
15     public string EmailAddress { get; set; }
16
17     // Relationship with custom Foreign Key name
18     public Gender Sex { get; set; }
19     public int SexID { get; set; }
20 }
```

Gender.cs

```
1  public class Gender
2  {
3      public int ID { get; set; }
4      public string Name { get; set; }
5
6      public List<User> Users { get; set; }
7  }
```

You can see that I have defined a Gender class which is going to hold only two records, one for Male and another one for Female users. There will be a **One to Many** relationship between them (not defined yet) and we will declare that SexID will be the User's foreign key to Gender table. Switch to the DataAccess project and install **Entity Framework** using Nuget Packages. Also make sure you add reference to the Domain project. First of all, we need to introduce the Entity Framework our classes. Add a UsersContext class as follow:

```
1  public class UsersContext : DbContext
2  {
3      public DbSet<User> Users { get; set; }
4      public DbSet<Gender> Genders { get; set; }
```

```
5 | }
```

To setup Database configurations you have two options. The one is to use DataAnnotations and the other is to use Fluent API. I prefer the latter cause I like to keep my domain objects clean. I usually add a “Configurations” folder in the project and setup a specific configuration for each of my domain objects. Do this and add two classes that inherits the **EntityTypeConfiguration** class as follow:

figuration.cs

```
public class UserConfiguration : EntityTypeConfiguration<User>
{
    public UserConfiguration()
    {
        HasKey(u => u.ID);

        Property(u => u.ID).HasColumnName("UserID");
        Property(u => u.FirstName).IsRequired().HasMaxLength(50);
        Property(u => u.LastName).IsRequired().HasMaxLength(50);
        Property(u => u.Username).IsRequired().HasMaxLength(100);
        Property(u => u.BirthDate).IsRequired().HasMaxLength(20);
        Property(u => u.EmailAddress).IsRequired().HasMaxLength(100);

        // Our custom foreign key
        HasRequired(u => u.Sex).WithMany(s => s.Users).HasForeignKey(u => u.S
    }
}
```

GenderConfiguration.cs

```
1 public class GenderConfiguration : EntityTypeConfiguration<Gender>
2 {
3     public GenderConfiguration()
4     {
5         HasKey(g => g.ID);
6
7         // We will set ID values manually
8         Property(g => g.ID).HasDatabaseGeneratedOption(DatabaseGener
9     }
10 }
```

As you can see these are basic database configurations that will have impact in the way Entity Framework Code First will set the database for us. You need to add those two configurations in the UsersContext class:

```
1 public class UsersContext : DbContext
2 {
3
4     protected override void OnModelCreating(DbModelBuilder modelBuilder)
5     {
6         modelBuilder.Configurations.Add(new UserConfiguration());
7         modelBuilder.Configurations.Add(new GenderConfiguration());
8     }
9 }
```

```

8         base.OnModelCreating(modelBuilder);
9     }
10
11     public DbSet<User> Users { get; set; }
12     public DbSet<Gender> Genders { get; set; }
13 }

```

While at the same project, you need to set an initializer class that will always re-create our database, seeding two records for the Gender table. On production projects you will probably need migration logic but this is enough for this post.

DbInitializer.cs

```

1 public class DbInitializer : DropCreateDatabaseAlways<UsersContext>
2 {
3     protected override void Seed(UsersContext context)
4     {
5         GetGenders().ForEach(g => context.Genders.Add(g));
6     }
7
8     private static List<Gender> GetGenders()
9     {
10         return new List<Gender>
11         {
12             new Gender {
13                 ID = 1,
14                 Name = "Male"
15             },
16             new Gender {
17                 ID = 2,
18                 Name = "Female"
19             }
20         };
21     }
22 }

```

Now switch to the WebAPI project and for start install Entity Framework through Nuget Packages. Then add references to both of the class library projects. Open Global.asax.cs and add the following line into the **Application_Start** method, in order to set our database initializer.

```

1 void Application_Start(object sender, EventArgs e)
2 {
3     // Code that runs on application startup
4     AreaRegistration.RegisterAllAreas();
5     GlobalConfiguration.Configure(WebApiConfig.Register);
6     RouteConfig.RegisterRoutes(RouteTable.Routes);
7
8     // Initialize the users database.
9     Database.SetInitializer(new DbInitializer());
10 }

```

Adding a **connectionString** element in to the Web.config file and declare where the database will be created.

```

1 <connectionStrings>
2   <add name="UsersContext" providerName="System.Data.SqlClient" connectio
3 </connectionStrings>

```

Add a new Web API Controller class named UsersController for retrieving and posting User objects. As soon as we invoke one of this controller actions, our database will be created.

UsersController.cs

```

1 public class UsersController : ApiController
2 {
3     private UsersContext db = new UsersContext();
4
5     // GET: api/Users
6     public IQueryable<User> GetUsers()
7     {
8         return db.Users;
9     }
10
11    // GET: api/Users/5
12    [ResponseType(typeof(User))]
13    public async Task<IHttpActionResult> GetUser(int id)
14    {
15        User user = await db.Users.FindAsync(id);
16        if (user == null)
17        {
18            return NotFound();
19        }
20
21        return Ok(user);
22    }
23
24    // POST: api/Users
25    [ResponseType(typeof(User))]
26    public async Task<IHttpActionResult> PostUser(User user)
27    {
28        if (!ModelState.IsValid)
29        {
30            return BadRequest(ModelState);
31        }
32        else
33        {
34            try
35            {
36                db.Users.Add(user);
37                await db.SaveChangesAsync();
38            }
39            catch (Exception ex)
40            {
41                // Log
42            }
43            return CreatedAtRoute("DefaultApi", new { id = user.ID }, us
44        }
45    }
46

```

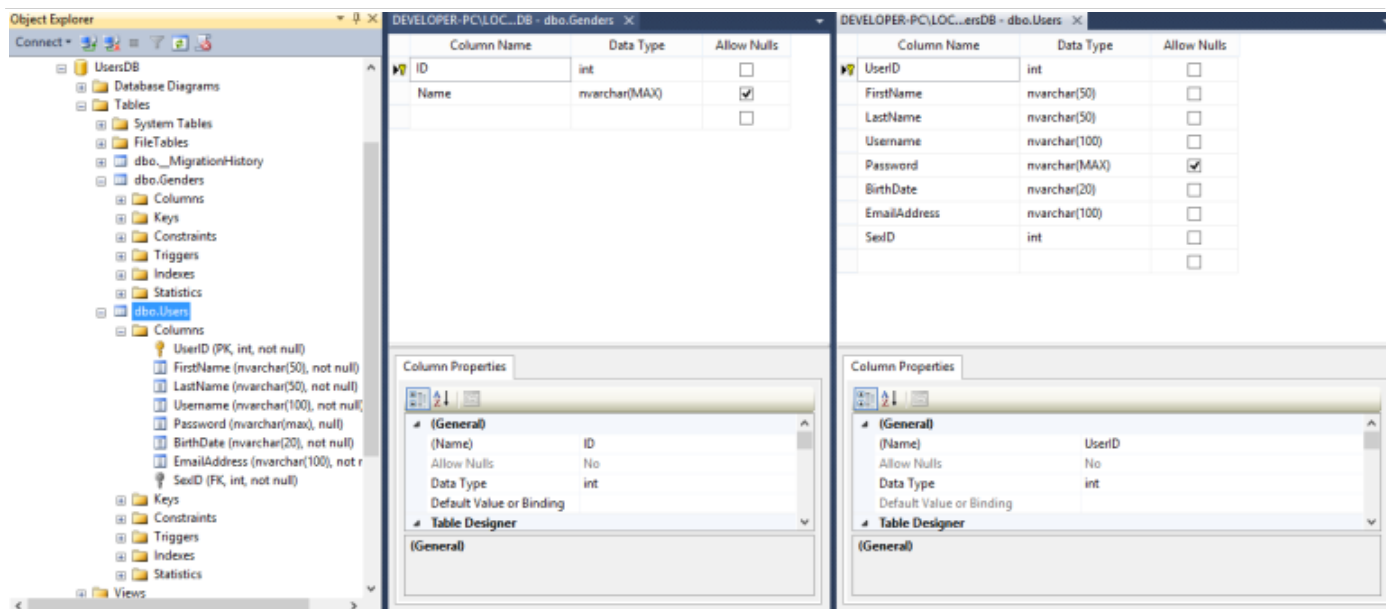
```

47     protected override void Dispose(bool disposing)
48     {
49         if (disposing)
50         {
51             db.Dispose();
52         }
53         base.Dispose(disposing);
54     }
55
56     private bool UserExists(int id)
57     {
58         return db.Users.Count(e => e.ID == id) > 0;
59     }
60 }

```

Before starting your application you may have to set **System.Web.Mvc** reference, Copy Local property to True, from the properties window.

Fire your application and invoke /api/users action through your browser. You should see the UsersDB database in your local server as follow:



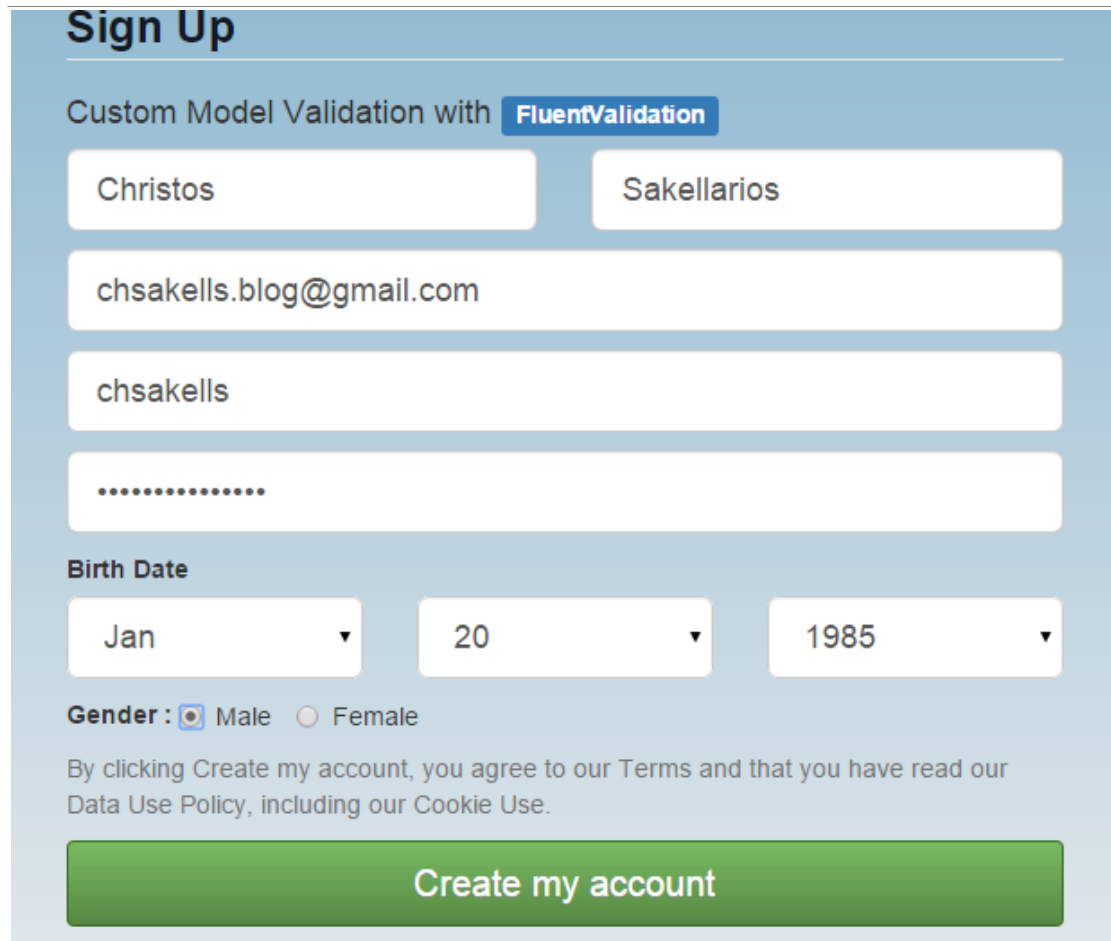
If you get any errors never mind, you can download this solution from the bottom of this post. We started this post to set custom validations to our object without affecting our database configurations. To check this, create an MVC Controller named HomeController with an Index method.

```

1     public class HomeController : Controller
2     {
3         // GET: Home
4         public ActionResult Index()
5         {
6             return View();
7         }
8     }

```

Right click inside the View and select Add view. This will create an Index.cshtml file. I prefer not to paste all of it's code here since it's quite big. I will explain though what I did and what I am posting from this page. You can create your own implementation for this or just open the attached solution at the bottom of this post. I have created a registration form using **AngularJS** and **Bootstrap** which simply posts form values to our UsersController post method.



If you write your own implementation just make sure on **Create my Account** button click event, to post all form values without running any javascript validation. Here is the AngularJS function running when the button is clicked.

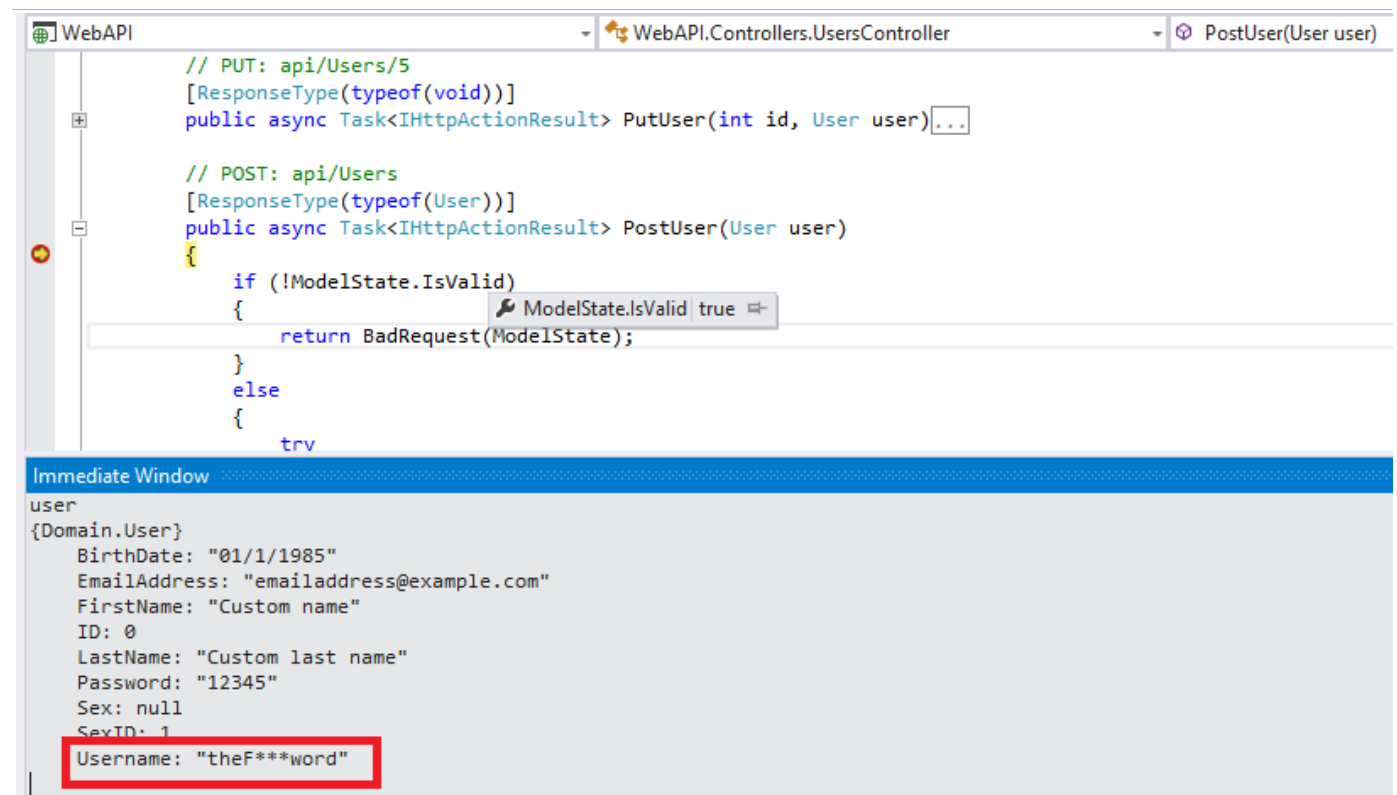
```
1  $scope.registerUser = function () {
2
3      $scope.modelSubmitted = false;
4
5      var newUser = {
6          FirstName: $scope.firstName,
7          LastName: $scope.lastName,
8          Username: $scope.userName,
9          Password: $scope.password,
10         BirthDate: $scope.birthMonth + '/' + $scope.birthDay
11         EmailAddress: $scope.emailAddress,
12         SexID: $scope.sex
13     };
14
15     $http.post(apiUrl, newUser)
16         .success(function (data, status, headers, config) {
17             $scope.newUserLocation = headers('Location');
18             $scope.hasModelErrors = false;
19             $scope.modelSubmitted = true;
20         });
```

```

21         // reset $scope values
22         $scope.firstName = '';
23         $scope.lastName = '';
24         $scope.userName = '';
25         $scope.password = '';
26         $scope.birthMonth = '01';
27         $scope.birthDay = 1;
28         $scope.birthYear = 1985;
29         $scope.emailAddress = '';
30     })
31     .error(function (error) {
32         console.log(error);
33         $scope.error = JSON.stringify(error.ModelState);
34         $scope.hasModelErrors = true;
35     }).finally(function () {
36     });
37
38 }

```

It's time to setup the Validation logic. We saw that the User.Username property has been created from Code First configuration as an `[nvarchar(100), not null]` property in the database. Hence, if the user selects a username with inappropriate or offensive words, the username of course will be valid and the stored in the database.



The screenshot shows the Visual Studio IDE with the `WebAPI.Controllers.UsersController` file open. The `PostUser` method is selected, and the `ModelState.IsValid` property is being debugged. The `ModelState.IsValid` property is set to `true`, and the `return BadRequest(ModelState);` statement is highlighted. Below the code editor, the `Immediate Window` displays the `user` object, which is an instance of `Domain.User`. The `Username` property is highlighted in red, showing the value `"theF***word"`.

```

// PUT: api/Users/5
[ResponseType(typeof(void))]
public async Task<IHttpActionResult> PutUser(int id, User user)

// POST: api/Users
[ResponseType(typeof(User))]
public async Task<IHttpActionResult> PostUser(User user)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    else
    {
        trv
    }
}

```

Immediate Window

```

user
{Domain.User}
  BirthDate: "01/1/1985"
  EmailAddress: "emailaddress@example.com"
  FirstName: "Custom name"
  ID: 0
  LastName: "Custom last name"
  Password: "12345"
  Sex: null
  SexID: 1
  Username: "theF***word"

```

What we want to do is to add validation logic so that if certain words found on the username property, automatically mark the `ModelState` as invalid. This can be done using the **FluentValidation**. You can see that we try to inject a model validation logic that has nothing to do with database table properties. Switch to the Domain class library and add a new folder named

Validations. Install the FluentValidation package running the following command:

1 | Install-package FluentValidation

Add a UserValidator class that inherits the **AbstractValidator** class.

UserValidator.cs

```
1  public UserValidator()
2  {
3      RuleFor(user => user.FirstName)
4          .Length(10, 50)
5          .WithMessage("First name must be between 10 to 50 characters");
6
7      RuleFor(user => user.FirstName)
8          .Must(HasNotDigits)
9          .WithMessage("First name cannot contain digits");
10
11     RuleFor(user => user.SexID)
12         .NotEqual(0)
13         .WithMessage("Please select a Gender");
14
15
16     RuleFor(user => user.Username).NotNull().WithMessage("Username is required");
17     RuleFor(user => user.FirstName).NotNull().WithMessage("First name is required");
18
19     RuleFor(user => user.Username)
20         .Must(NotOffensive)
21         .WithMessage("Username has invalid keywords. Try another one");
22
23     RuleFor(user => user.EmailAddress).EmailAddress();
24
25
26 }
27
28 private bool HasNotDigits(string name)
29 {
30     if (!string.IsNullOrEmpty(name))
31         return !name.Any(char.IsDigit);
32     else return true;
33 }
34
35 private bool NotOffensive(string username)
36 {
37     if (!string.IsNullOrEmpty(username))
38     {
39         bool isOffensive = InvalidKeywords.Contains(username.ToLower());
40         return !isOffensive;
41     }
42     else return true;
43 }
44
45 // Pool from Database..
46 private static List<string> InvalidKeywords = new List<string>()
47 {
48     "thef***word",
```

```

49         "keyword_02",
50         "keyword_03",
51         "keyword_04"
52     };
53 }

```

I have highlighted only the validations added for the username. I added some more for other properties so you can study them later. I declared a **Rule** for the username property which says that username must not be offensive and if so, return the relative message. You need to attach the validation configuration to the specific object you want to validate, that is the User class. This is the updated version.

```

1  public class User : IValidatableObject
2  {
3      public int ID { get; set; }
4
5      public string FirstName { get; set; }
6
7      public string LastName { get; set; }
8
9      public string Username { get; set; }
10
11     public string Password { get; set; }
12
13     public string BirthDate { get; set; }
14
15     public string EmailAddress { get; set; }
16
17     // Relationship with custom Foreign Key name
18     public Gender Sex { get; set; }
19     public int SexID { get; set; }
20
21     public IEnumerable<ValidationResult> Validate(ValidationContext
22     {
23         var validator = new UserValidator();
24         var result = validator.Validate(this);
25         return result.Errors.Select(item => new ValidationResult(it
26     }
27 }

```

You can see how many we can achieve using FluentValidation. In the solution you will find that I defined that First name cannot contain Digits and if so return an appropriate message.

Custom Model Validation with **FluentValidation**

Birth Date

Gender : ☒ Male ☐ Female

By clicking Create my account, you agree to our Terms and that you have read our Data Use Policy, including our Cookie Use.

Create my account

{"user.Username":["Username has invalid keywords. Try another one."]}

```
1 RuleFor(user => user.FirstName)
2     .Must(HasNotDigits)
3     .WithMessage("First name cannot contain digits");
4
5 private bool HasNotDigits(string name)
6 {
7     if (!string.IsNullOrEmpty(name))
8         return !name.Any(char.IsDigit);
9     else return true;
10 }
```

Custom Model Validation with **FluentValidation**

first name 10 last name

email@example.com

theF***word

.....

Birth Date

Jan 1 1985

Gender : ☒ Male ☐ Female

By clicking Create my account, you agree to our Terms and that you have read our Data Use Policy, including our Cookie Use.

Create my account

`{"user.FirstName":["First name cannot contain digits"],"user.Username":["Username has invalid keywords. Try another one."]}`

Food for thought

The reason I decided not to use custom ViewModel objects is to highlight the opposite: That is validation logic shouldn't necessary depend (only) on domain/entity objects but also in custom ViewModel objects. You can create custom lightweighted ViewModel classes for exchanging data between client and server and only if their ModelState is valid proceed to further processing. This way you prevent useless interaction with your database repository.

What comes next

In the next two 2 posts we will see how to create a full functional SPA application using [AngularJS](#) and pure Web API. Make sure you follow this blog in order to get notified when this occurs!

You can download the project we built from [here](#).

Advertisements

