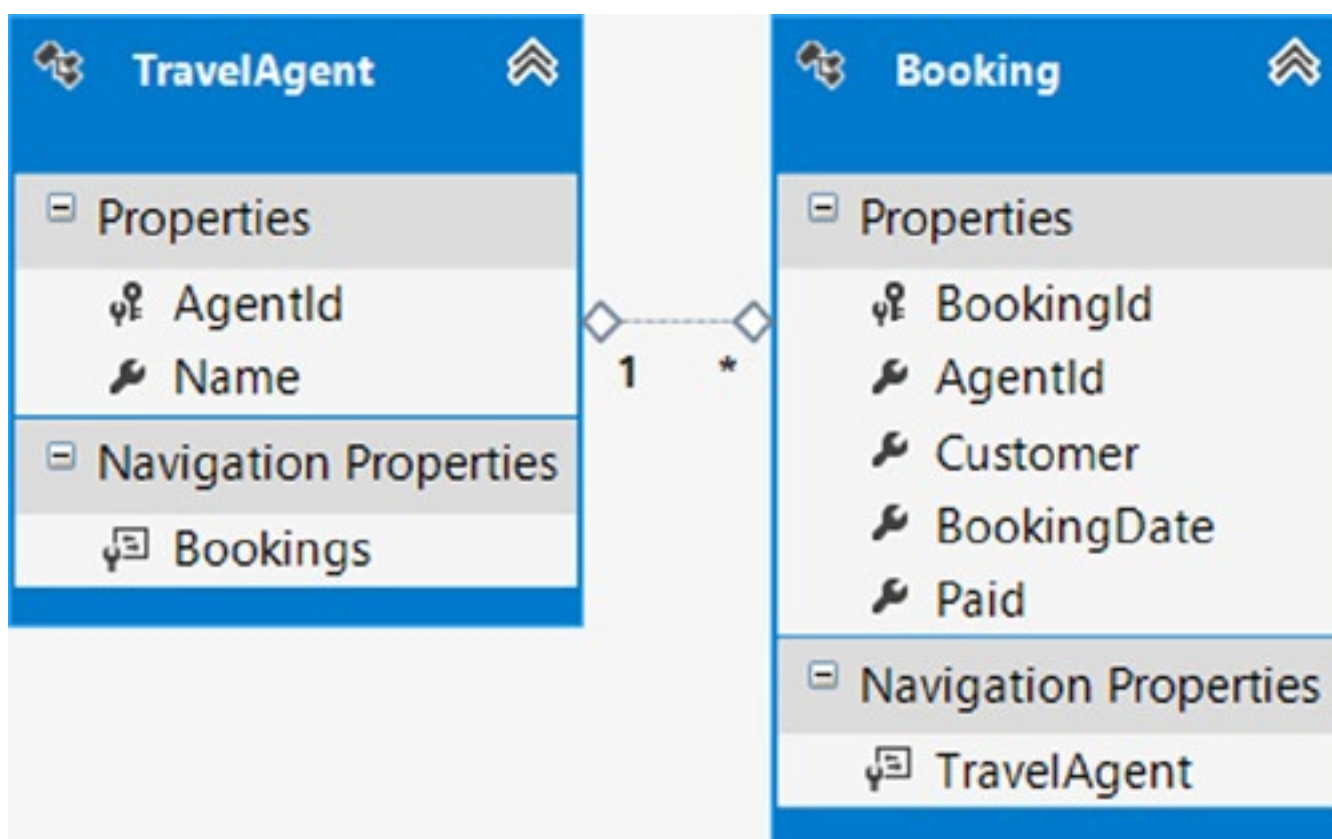


در [قسمت قبلی](#) بروز رسانی موجودیت های منفصل با WCF را بررسی کردیم. در این قسمت خواهیم دید چگونه می توان تغییرات موجودیت ها را تشخیص داد و عملیات CRUD را روی یک Object Graph اجرا کرد.

تشخیص تغییرات با Web API

فرض کنید می خواهیم از سرویس های Web API برای انجام عملیات CRUD استفاده کنیم، اما بدون آنکه برای هر موجودیت متدهایی مجزا تعریف کنیم. به بیان دیگر می خواهیم عملیات مذکور را روی یک Object Graph انجام دهیم. مدیریت داده ها هم با مدل Code-First پیاده سازی می شود. در مثال جاری یک اپلیکیشن کنسول خواهیم داشت که بعنوان یک کلاینت سرویس را فراخوانی می کند. هر پروژه نیز در Solution مجزایی قرار دارد، تا یک محیط n-Tier را شبیه سازی کنیم.

مدل زیر را در نظر بگیرید.



همانطور که می بینید مدل ما آژانس های مسافرتی و رزرواسیون آنها را ارائه می کند. می خواهیم مدل و کد دسترسی داده ها را در یک سرویس Web API پیاده سازی کنیم تا هر کلاینتی که به HTTP دسترسی دارد بتواند عملیات CRUD را انجام دهد. برای ساختن سرویس مورد نظر مراحل زیر را دنبال کنید:

در ویژوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe3.Service تغییر دهید.

کنترلر جدیدی بنام TravelAgentController به پروژه اضافه کنید.

دو کلاس جدید با نام های TravelAgent و Booking بسازید و کد آنها را مطابق لیست زیر تغییر دهید.

```
public class TravelAgent
{
    public TravelAgent()
    {
        this.Bookings = new HashSet<Booking>();
    }

    public int AgentId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Booking> Bookings { get; set; }
}

public class Booking
{
    public int BookingId { get; set; }
    public int AgentId { get; set; }
    public string Customer { get; set; }
    public DateTime BookingDate { get; set; }
    public bool Paid { get; set; }
    public virtual TravelAgent TravelAgent { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید.

کلاس جدیدی بنام Recipe3Context بسازید و کد آن را مطابق لیست زیر تغییر دهید.

```
public class Recipe3Context : DbContext
{
    public Recipe3Context() : base("Recipe3ConnectionString") { }
    public DbSet<TravelAgent> TravelAgents { get; set; }
    public DbSet<Booking> Bookings { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<TravelAgent>().HasKey(x => x.AgentId);
        modelBuilder.Entity<TravelAgent>().ToTable("TravelAgents");
        modelBuilder.Entity<Booking>().ToTable("Bookings");
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه کنید.

```
<connectionStrings>
  <add name="Recipe3ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به متد Application_Start اضافه نمایید. این کد بررسی Model Compatibility در EF را غیرفعال می کند. همچنین به JSON serializer می گوئیم که self-referencing loop خاصیت های پیمایشی را نادیده بگیرد. این حلقه بدلیل ارتباط bidirectional بین موجودیت ها بوجود می آید.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);

    // The bidirectional navigation properties between related entities
    // create a self-referencing loop that breaks Web API's effort to
    // serialize the objects as JSON. By default, Json.NET is configured
    // to error when a reference loop is detected. To resolve problem,
    // simply configure JSON serializer to ignore self-referencing loops.
    GlobalConfiguration.Configuration.Formatters.JsonFormatter
        .SerializerSettings.ReferenceLoopHandling =
        Newtonsoft.Json.ReferenceLoopHandling.Ignore;
    ...
}
```

```
}
```

فایل RouteConfig.cs را باز کنید و قوانین مسیریابی را مانند لیست زیر تغییر دهید.

```
public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "ActionMethodSave",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional });
}
```

در آخر کنترلر TravelAgent را باز کنید و کد آن را مطابق لیست زیر بروز رسانی کنید.

```
public class TravelAgentController : ApiController
{
    // GET api/travelagent
    [HttpGet]
    public IEnumerable<TravelAgent> Retrieve()
    {
        using (var context = new Recipe3Context())
        {
            return context.TravelAgents.Include(x => x.Bookings).ToList();
        }
    }

    /// <summary>
    /// Update changes to TravelAgent, implementing Action-Based Routing in Web API
    /// </summary>
    public HttpResponseMessage Update(TravelAgent travelAgent)
    {
        using (var context = new Recipe3Context())
        {
            var newParentEntity = true;
            // adding the object graph makes the context aware of entire
            // object graph (parent and child entities) and assigns a state
            // of added to each entity.
            context.TravelAgents.Add(travelAgent);
            if (travelAgent.AgentId > 0)
            {
                // as the Id property has a value greater than 0, we assume
                // that travel agent already exists and set entity state to
                // be updated.
                context.Entry(travelAgent).State = EntityState.Modified;
                newParentEntity = false;
            }

            // iterate through child entities, assigning correct state.
            foreach (var booking in travelAgent.Bookings)
            {
                if (booking.BookingId > 0)
                {
                    // assume booking already exists if ID is greater than zero.
                    // set entity to be updated.
                    context.Entry(booking).State = EntityState.Modified;
                }
            }

            context.SaveChanges();
            HttpResponseMessage response;
            // set Http Status code based on operation type
            response = Request.CreateResponse(newParentEntity ? HttpStatusCode.Created :
            HttpStatusCode.OK, travelAgent);
            return response;
        }
    }

    [HttpDelete]
    public HttpResponseMessage Cleanup()
    {
        using (var context = new Recipe3Context())
        {
            context.Database.ExecuteSqlCommand("delete from [bookings]");
            context.Database.ExecuteSqlCommand("delete from [travelagents]");
        }
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}
```

}

در قدم بعدی کلاینت پروژه را می‌سازیم که از سرویس Web API مان استفاده می‌کند.

در ویژوال استودیو پروژه جدیدی از نوع Console application بسازید و نام آن را به Recipe3.Client تغییر دهید.
فایل program.cs را باز کنید و کد آن را مطابق لیست زیر بروز رسانی کنید.

```
internal class Program
{
    private HttpClient _client;
    private TravelAgent _agent1, _agent2;
    private Booking _booking1, _booking2, _booking3;
    private HttpResponseMessage _response;

    private static void Main()
    {
        Task t = Run();
        t.Wait();
        Console.WriteLine("\nPress <enter> to continue...");
        Console.ReadLine();
    }

    private static async Task Run()
    {
        var program = new Program();
        program.ServiceSetup();
        // do not proceed until clean-up is completed
        await program.CleanupAsync();
        program.CreateFirstAgent();
        // do not proceed until agent is created
        await program.AddAgentAsync();
        program.CreateSecondAgent();
        // do not proceed until agent is created
        await program.AddSecondAgentAsync();
        program.ModifyAgent();
        // do not proceed until agent is updated
        await program.UpdateAgentAsync();
        // do not proceed until agents are fetched
        await program.FetchAgentsAsync();
    }

    private void ServiceSetup()
    {
        // set up infrastructure for Web API call
        _client = new HttpClient {BaseAddress = new Uri("http://localhost:6687/")};
        // add Accept Header to request Web API content negotiation to return resource in JSON format
        _client.DefaultRequestHeaders.Accept.Add(new
        MediaTypeWithQualityHeaderValue("application/json"));
    }

    private async Task CleanupAsync()
    {
        // call cleanup method in service
        _response = await _client.DeleteAsync("api/travelagent/cleanup/");
    }

    private void CreateFirstAgent()
    {
        // create new Travel Agent and booking
        _agent1 = new TravelAgent {Name = "John Tate"};
        _booking1 = new Booking
        {
            Customer = "Karen Stevens",
            Paid = false,
            BookingDate = DateTime.Parse("2/2/2010")
        };

        _booking2 = new Booking
        {
            Customer = "Dolly Parton",
            Paid = true,
            BookingDate = DateTime.Parse("3/10/2010")
        };

        _agent1.Bookings.Add(_booking1);
        _agent1.Bookings.Add(_booking2);
    }
}
```

```

}

private async Task AddAgentAsync()
{
    // call generic update method in Web API service to add agent and bookings
    _response = await _client.PostAsync("api/travelagent/update/",
        _agent1, new JsonMediaTypeFormatter());

    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service, which will include
        // database-generated Ids for each entity
        _agent1 = await _response.Content.ReadAsAsync<TravelAgent>();
        _booking1 = _agent1.Bookings.FirstOrDefault(x => x.Customer == "Karen Stevens");
        _booking2 = _agent1.Bookings.FirstOrDefault(x => x.Customer == "Dolly Parton");

        Console.WriteLine("Successfully created Travel Agent {0} and {1} Booking(s)",
            _agent1.Name, _agent1.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private void CreateSecondAgent()
{
    // add new agent and booking
    _agent2 = new TravelAgent {Name = "Perry Como"};
    _booking3 = new Booking {
        Customer = "Loretta Lynn",
        Paid = true,
        BookingDate = DateTime.Parse("3/15/2010")};
    _agent2.Bookings.Add(_booking3);
}

private async Task AddSecondAgentAsync()
{
    // call generic update method in Web API service to add agent and booking
    _response = await _client.PostAsync("api/travelagent/update/", _agent2, new
JsonMediaTypeFormatter());

    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service
        _agent2 = await _response.Content.ReadAsAsync<TravelAgent>();
        _booking3 = _agent2.Bookings.FirstOrDefault(x => x.Customer == "Loretta Lynn");
        Console.WriteLine("Successfully created Travel Agent {0} and {1} Booking(s)",
            _agent2.Name, _agent2.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private void ModifyAgent()
{
    // modify agent 2 by changing agent name and assigning booking 1 to him from agent 1
    _agent2.Name = "Perry Como, Jr.";
    _agent2.Bookings.Add(_booking1);
}

private async Task UpdateAgentAsync()
{
    // call generic update method in Web API service to update agent 2
    _response = await _client.PostAsync("api/travelagent/update/", _agent2, new
JsonMediaTypeFormatter());
    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service, which will include Ids
        _agent1 = _response.Content.ReadAsAsync<TravelAgent>().Result;
        Console.WriteLine("Successfully updated Travel Agent {0} and {1} Booking(s)", _agent1.Name,
            _agent1.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private async Task FetchAgentsAsync()
{
    // call Get method on service to fetch all Travel Agents and Bookings
    _response = _client.GetAsync("api/travelagent/retrieve").Result;
    if (_response.IsSuccessStatusCode)
    {

```

```
// capture newly created travel agent from service, which will include Ids
var agents = await _response.Content.ReadAsAsync<IEnumerable<TravelAgent>>();

foreach (var agent in agents)
{
    Console.WriteLine("Travel Agent {0} has {1} Booking(s)", agent.Name,
agent.Bookings.Count());
}
}
else
    Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}
}
```

در آخر کلاس های TravelAgent و Booking را به پروژه کلاینت اضافه کنید. اینگونه کدها بهتر است در لایه مجزایی قرار گیرند و بین پروژه ها به اشتراک گذاشته شوند.

اگر اپلیکیشن کنسول (کلاینت) را اجرا کنید با خروجی زیر مواجه خواهید شد.

```
Successfully created Travel Agent John Tate and 2 Booking(s)
Successfully created Travel Agent Perry Como and 1 Booking(s)
Successfully updated Travel Agent Perry Como, Jr. and 2 Booking(s)
Travel Agent John Tate has 1 Booking(s)
Travel Agent Perry Como, Jr. has 2 Booking(s)
```

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک کنترلر MVC Web Controller دارد که پس از اجرا شما را به صفحه خانه هدایت می کند. در این مرحله سایت در حال اجرا است و سرویس ها قابل دسترسی هستند.

سپس اپلیکیشن کنسول را باز کنید، روی خط اول کد فایل program.cs یک breakpoint قرار دهید و آن را اجرا کنید. ابتدا آدرس سرویس Web API را نگاشت می کنیم و با تنظیم مقدار خاصیت Accept Header از سرویس درخواست می کنیم که اطلاعات را با فرمت JSON بازگرداند.

بعد از آن با استفاده از آبجکت HttpClient متد DeleteAsync را فراخوانی می کنیم که روی کنترلر TravelAgent تعریف شده است. این متد تمام داده های پیشین را حذف میکند.

در قدم بعدی سه آبجکت جدید می سازیم: یک آژانس مسافرتی و دو رزرواسیون. سپس این آبجکت ها را با فراخوانی متد PostAsync روی آبجکت HttpClient به سرویس ارسال می کنیم. اگر به متد Update در کنترلر TravelAgent یک breakpoint اضافه کنید، خواهید دید که این متد آبجکت آژانس مسافرتی را بعنوان یک پارامتر دریافت می کند و آن را به موجودیت TravelAgents در Context جاری اضافه می نماید. این کار آبجکت آژانس مسافرتی و تمام آبجکت های فرزند آن را در حالت Added اضافه می کند و باعث می شود که context جاری شروع به ردیابی (tracking) آنها کند.

نکته: قابل ذکر است که اگر موجودیت های متعددی با مقداری یکسان در خاصیت کلید اصلی (Primary-key value) دارید باید مجموعه آبجکت های خود را Add کنید و نه Attach. در مثال جاری چند آبجکت Booking داریم که مقدار کلید اصلی آنها صفر است (Bookings with Id = 0). اگر از Attach استفاده کنید EF پیغام خطایی صادر می کند چرا که چند موجودیت با مقادیر کلید اصلی یکسان به context جاری اضافه کرده اید.

بعد از آن بر اساس مقدار خاصیت Id مشخص می کنیم که موجودیت ها باید بروز رسانی شوند یا خیر. اگر مقدار این فیلد بزرگتر از صفر باشد، فرض بر این است که این موجودیت در دیتابیس وجود دارد بنابراین خاصیت EntityState را به Modified تغییر می دهیم. علاوه بر این فیلدی هم با نام newParentEntity تعریف کرده ایم که توسط آن بتوانیم کد وضعیت مناسبی به کلاینت بازگردانیم. در صورتی که مقدار فیلد Id در موجودیت TravelAgent برابر با یک باشد، مقدار خاصیت EntityState را به همان

Added رها می کنیم.

سپس تمام آبجکت های فرزند آژانس مسافرتی (رزرواسیون ها) را بررسی میکنیم و همین منطق را روی آنها اعمال می کنیم. یعنی در صورتی که مقدار فیلد Id آنها بزرگتر از 0 باشد وضعیت EntityState را به Modified تغییر می دهیم. در نهایت متد SaveChanges را فراخوانی می کنیم. در این مرحله برای موجودیت های جدید اسکریپت های Insert و برای موجودیت های تغییر کرده اسکریپت های Update تولید می شود. سپس کد وضعیت مناسب را به کلاینت بر می گردانیم. برای موجودیت های اضافه شده کد وضعیت 201 (Created) و برای موجودیت های بروز رسانی شده کد وضعیت 200 (OK) باز می گردد. کد 201 به کلاینت اطلاع می دهد که رکورد جدید با موفقیت ثبت شده است، و کد 200 از بروز رسانی موفقیت آمیز خبر می دهد. هنگام تولید سرویس های REST-based بهتر است همیشه کد وضعیت مناسبی تولید کنید.

پس از این مراحل، آژانس مسافرتی و رزرواسیون جدیدی می سازیم و آنها را به سرویس ارسال می کنیم. سپس نام آژانس مسافرتی دوم را تغییر می دهیم، و یکی از رزرواسیون ها را از آژانس اولی به آژانس دومی منتقل می کنیم. اینبار هنگام فراخوانی متد Update تمام موجودیت ها شناسه ای بزرگتر از 1 دارند، بنابراین وضعیت EntityState آنها را به Modified تغییر می دهیم تا هنگام ثبت تغییرات دستورات بروز رسانی مناسب تولید و اجرا شوند.

در آخر کلاینت ما متد Retrieve را روی سرویس فراخوانی می کند. این فراخوانی با کمک متد GetAsync انجام می شود که روی آبجکت HttpClient تعریف شده است. فراخوانی این متد تمام آژانس های مسافرتی به همراه رزرواسیون های متناظرشان را دریافت می کند. در اینجا با استفاده از متد Include تمام رکوردهای فرزند را به همراه تمام خاصیت هایشان (properties) بارگذاری می کنیم.

دقت کنید که مرتب کننده JSON تمام خواص عمومی (public properties) را باز می گرداند، حتی اگر در کد خود تعداد مشخصی از آنها را انتخاب کرده باشید.

نکته دیگر آنکه در مثال جاری از قراردادهای توکار Web API برای نگاشت درخواست های HTTP به اکشن متدها استفاده نکرده ایم. مثلاً بصورت پیش فرض درخواست های POST به متدهایی نگاشت می شوند که نام آنها با "Post" شروع می شود. در مثال جاری قواعد مسیریابی را تغییر داده ایم و رویکرد مسیریابی RPC-based را در پیش گرفته ایم. در اپلیکیشن های واقعی بهتر است از قواعد پیش فرض استفاده کنید چرا که هدف Web API ارائه سرویس های REST-based است. بنابراین بعنوان یک قاعده کلی بهتر است متدهای سرویس شما به درخواست های متناظر HTTP نگاشت شوند. و در آخر آنکه بهتر است لایه مجزایی برای میزبانی کدهای دسترسی داده ایجاد کنید و آنها را از سرویس Web API تفکیک نمایید.

نظرات خوانندگان

نویسنده: وحید

تاریخ: ۱۱:۶ ۱۳۹۲/۱۱/۱۱

با سلام شما فرمودید: " و در آخر آنکه بهتر است لایه مجزایی برای میزبانی کدهای دسترسی داده ایجاد کنید و آنها را از سرویس Web API تفکیک نمایید. " برای برقراری امنیت در این سرویس چه باید کرد؟ اگر شخصی آدرس سرویس ما رو داشت و در خواست های را به آن ارسال کرد چگونه آن را نسبت به بقیه کاربران تمیز کند؟ چون در حقیقت webapi را در پروژه جدیدی در solution قرار دادیم و جدا هاست می شود. ممنون

نویسنده: محسن خان

تاریخ: ۱۱:۴۲ ۱۳۹۲/۱۱/۱۱

برای برقراری امنیت، تعیین هویت و اعتبارسنجی در وب API عموماً یا از [Forms authentication](#) استفاده می شود و یا از [ASP.NET Identity](#) . زیر ساخت آن یکی است و مشترک.