### REST nedir?

Client-Server arasındaki iletişimi HTTP protokolüyle kolay ve hafif (lightweight) bir şekilde sağlanmasına olanak veren bir yapıdır. REST (**Re**presentational **S**tate **T**ransfer) tanımı ilk defa, 2000 yılında Apache HTTP Server'ın mucitlerinden Roy Fielding tarafından yapılmıştır. REST yapısıyla implemente edilen servislere RESTful servis denir ve bu servislerin sunucu tarafında kurulumu, konfigürasyonu ve yönetimi diğer servis yapılarına göre daha kolaydır. RESTful servislerin amacı temelde sunucu ve istemci arasındaki veri alışverişini platform bağımsız ve olabilecek en az veri yüküyle sağlamak olduğundan JSON, HTML, XML gibi bir çok response tipiyle çalışabilmektedir. JSON (JavaScript-Object-Notation) yapısı bir çok mobil platformda ve Web üzerinde kullanışlı ve az yer kaplamasından dolayı sıklıkla tercih edilir.

Bir servis, RESTful sıfatını kazanabilmek için aşağıdaki koşulları sağlamasi gerekmektedir;

* **Client–server (Seperation of Cöncerns)**

İstemcilerin sunucu tarafında herhangi bir iç yapıyı bilmemesi, sunucuların da istemci hakkında kullanıcı arabirimini veya durumunu bilmemesi gibi, her iki yapının da birbirinden tamamen bağımsız, ancak ortak bir interface ile iletişimini sağlayabilmesi mantığıdır. Bu şekilde aralarındaki interface aynı kalmasi koşuluyla sunucu ve istemci tarafı birbirinden bağımsız bir şekilde geliştirilebilir ve düzenlenebilir. Client tarafındaki kod farklı platformlara uyarlanabilir ve sunucu tarafındaki kod da daha basitleştirilmiş olur.

* **Stateless**

İstekler esnasında sunucuda client ile ilgili bir bilginin tutulmamasıdır. RESTful servis, istemcilerin request anında gönderdığı bir token, kimlik bilgisi veya benzeri ayırıcı bilgi ile client ayırımini yapar ve ilgili yanıtı kendisine gönderir.

* **Cacheable**

İstemcilerin sunucudan dönen yanıtı cache'leyebilir. Bu nedenle gönderilen her response'un, saklanabilir (Cacheable) olup olmadığının sunucu tarafından belirtilmesi gerekir. Aksi durumda yanlış bir veri alışverişi sağlanabilir.

* **Layered system**

İstemcinin sunucudaki yapı hakkında hangi katmana veya hangi servise baglanacağını bilmemesi, istemciyi karşılayan yapının da iç katmanlardan ve servislerden ayrıştırılmış olmasi gerekir. Bu ayrıca güvenlik önlemleri açısından da ayrı bir önem taşımakta.

* **Uniform interface**

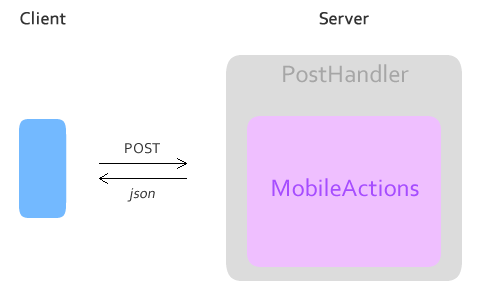
Ortak bir yapı üzerinden client ve server iletişiminin gerçeklestirilmesi, her iki tarafında bu ortak interface arkasında ayrı yapılarla geliştirilmesi açısından önem taşır.

* **Code on demand (opsiyonel)**

Javascript uygulamarında goruleceği gibi, sunucular gecici olarak client tarafında bazi işleyişleri modifiye etmek ya da genişletmek amacıyla response olarak client-side script gönderebilir. Ancak bu koşul tum client'lar için geçerli olamayacağından dolayı opsiyoneldir, ve duruma göre farklılık gösterebilir.

### REST with IHttpHandler

Bu örnekte ASP.NET IHttpHandler kullanarak bir RESTful servis oluşturacağız. Servisimiz içerisinde Search ve List metodları olacak ve bu metodları *SERVER/HttpRest/methodName* şeklinde çağıracağız.

Kısaca nasıl çalışacağını açıklamak gerekirse;

Oluşturduğumuz HttpHandler içerisinde client tarafından yollanan request'i karşılayacağız. Request'in yapıldığı URI üzerinden hangi metodun çağırılacağını Uri'deki path ile anlayacağız, ve içerisindeki JSON formatında serialize edilmiş post verisini metodlarımızın içerisinde deserialize edip ilgili tiplere dönüştüreceğiz. İşlemin sonunda response nesnesini ise JSON formatında göndermek üzere serialize edeceğiz.

**Demo:**

Öncelikle bir library projesi oluşturalım. Daha kapsamlı projelerde business ve repository'ler farklı katmanda bulunabilir, ancak bu küçük örnekte hepsini Core olarak adlandırdığım projeye yerlestireceğim.

Modellerimizi oluşturalım. Search ve User modellerimiz için aşağıdaki gibi iki class'a gereksinimimiz var;

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public string Surname { get; set; }

public string Email { get; set; }

public string DeviceId { get; set; }

}

public class Search

{

public string Email { get; set; }

public string Name { get; set; }

public int Id { get; set; }

}

Metodlarımızın yer aldığı MobileActions sınıfımız da aşağıdaki gibi. Basit şekilde Search ve List metodlarımız olacak. Örnek projede business ve data katmanları yerine statik veri getiren bir kac metod tanımladim, ekte bulabileceğiniz Utils sınıfında bu metodlar yer alıyor.

internal class MobileActions

{

public object Search(string payload)

{

try

{

Search search = payload.ToModel<Search>();

User user = Utils.SearchUser(search);

return new { Ok = true, User = user };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

}

}

public object List(string payload)

{

try

{

return new { Ok = true, UserList = Utils.Users };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

}

}

}

Sirada, oluşturacağımız singleton bir Invoker sınıfi ile client tarafından istenen metodun çalışmasını sağlayacağız. Burada hangi metodu çalıştıracağımızı client'in gönderdığı request uri üzerindeki path'i bularak yapacağımızı belirtmiştim. Bunun için reflection kullanarak yukarıdaki MobileActions sınıfında yer alan metodları bulacağız ve path ile eşleşen metodu gönderilen parametre ile çalıştıracağız. Büyük projelerde her defasında sınıfı reflect edip metodları listelemek performans sorununa yol acabilir, bunun onune geçmek için MobileAction metodlarını da cacheleyebilirsiniz.

internal class Invoker

{

private static readonly object \_lock = new object();

private static Invoker \_invoker;

private MobileActions \_mobileActions;

private Invoker() { }

public static Invoker Instance

{

get

{

if (\_invoker == null)

{

lock (\_lock) {

if (\_invoker == null)

{

\_invoker = new Invoker();

}

}

}

return \_invoker;

}

}

private MobileActions MobileActionInstance

{

get

{

if (\_mobileActions == null)

{

\_mobileActions = new MobileActions();

}

return \_mobileActions;

}

}

public object InvokeMethod(string command, string payload)

{

MethodInfo methodInfo = GetAction(command);

return methodInfo.Invoke(MobileActionInstance, new object[] { payload });

}

private MethodInfo[] MobileActionMethods

{

get { return MobileActionInstance.GetType().GetMethods(); }

}

private MethodInfo GetAction(string command)

{

foreach (MethodInfo method in MobileActionMethods.Where(method => method.Name.Equals( command, StringComparison.InvariantCultureIgnoreCase ))){

return method; }

throw new Exception(string.Format("Invalid action: '{0}'", command));

}

}

Şimdi de istemciden gelen istekleri işleyeceğimiz, IHttpHandler'dan türeteceğimiz PostHandler sınıfını implemente edelim. PostHandler, gelen istekte bulunan path ve post dataya göre client'in istediği komut ve payload değişkenlerini setledikten sonra Invoker aracılığıyla MobileAction içerisinde yer alan metodu çalıştıracaktır.

public class PostHandler : IHttpHandler

{

public bool IsReusable

{

get { return false; }

}

public void ProcessRequest(HttpContext context)

{

string responseData = null;

try

{

string command = context.Request.Path.Substring(1, context.Request.Path.Length - 1).Split('/')[1];

string payload = GetRequestPayload(context);

responseData = JsonConvert.SerializeObject(Invoker.Instance.InvokeMethod(command, payload));

}

catch (Exception ex)

{

object err = new { Ok = false, Message = ex.Message };

responseData = JsonConvert.SerializeObject(err);

}

finally

{

context.Response.ContentType = "application/json";

context.Response.Write(responseData);

context.Response.End();

}

}

private string GetRequestPayload(HttpContext context)

{

StringBuilder sb = new StringBuilder();

int streamLength; int streamRead;

Stream s = context.Request.InputStream;

streamLength = Convert.ToInt32(s.Length);

byte[] streamArray = new byte[streamLength];

streamRead = s.Read(streamArray, 0, streamLength);

for (int i = 0; i < streamLength; i++)

{

sb.Append(Convert.ToChar(streamArray[i]));

}

return sb.ToString();

}

}

Son olarak bos bir web projesi oluşturalım. Yalnızca web config içerisinde yapacağımız ufak bir ekleme ile oluşturduğumuz handler projesini web uygulamasının handler'i olarak belirteceğiz. Bu noktadan sonra web uygulamasına yapacağımız tum HTTP request'ler Core içerisindeki PostHandler.cs tarafından işlenecektir.

<handlers>

<clear/>

<add name="RESTPostHandler" path="\*/\*" verb="\*"

type="Cronom.Demo.HttpHandlerCore.PostHandler, Cronom.Demo.HttpHandlerCore"

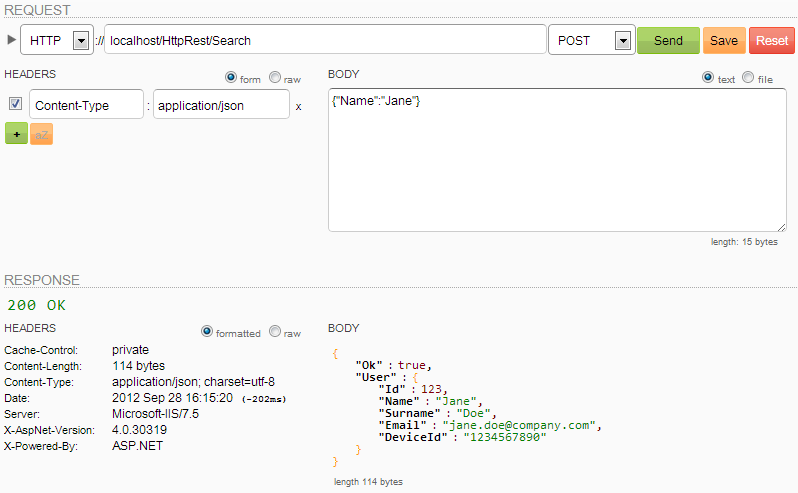
preCondition="integratedMode"/>

</handlers>

Artık elimizde aşağıdaki metodlar mevcut;

SERVER /HttpRest/Search  
SERVER /HttpRest/List

Parametre olarak metodlarda kullandığımız request modellerini business yapısına uygun degerler ile doldurduktan sonra json formatında göndererek istekte bulunabiliriz. Oluşturduğumuz RESTful servisin işleyişini görmek için metodları aşağıdaki gibi çağırabiliriz;



WebAPI nedir?

WCF development ekibi, WCF servislerine REST özelligi katmak amacıyla WCF Web API'yi geliştirmişti. Bu süreçe paralel olarak, basit bir şekilde Controller'dan json verisi dönen ve REST yapısına uygun bir yapı zaten ASP.NET MVC'de mevcut bulunuyordu. Ortadaki karışıklık daha fazla büyümeden, Microsoft tarafından WCF Web API ve ASP.NET 'in geliştirme ekibi birleştirilmesiyle beraber ASP.NET Web API ortaya çıkmış oldu.

Kısaca, MVC'nin WCF Web API ile olan birleşiminden doğan, RESTful servislerin geliştirilmesi için kullanilabilecek güzel bir framework diyebiliriz.

ASP.NET Web API nin standart bir HttpHandler'a göre su avantajları vardir;

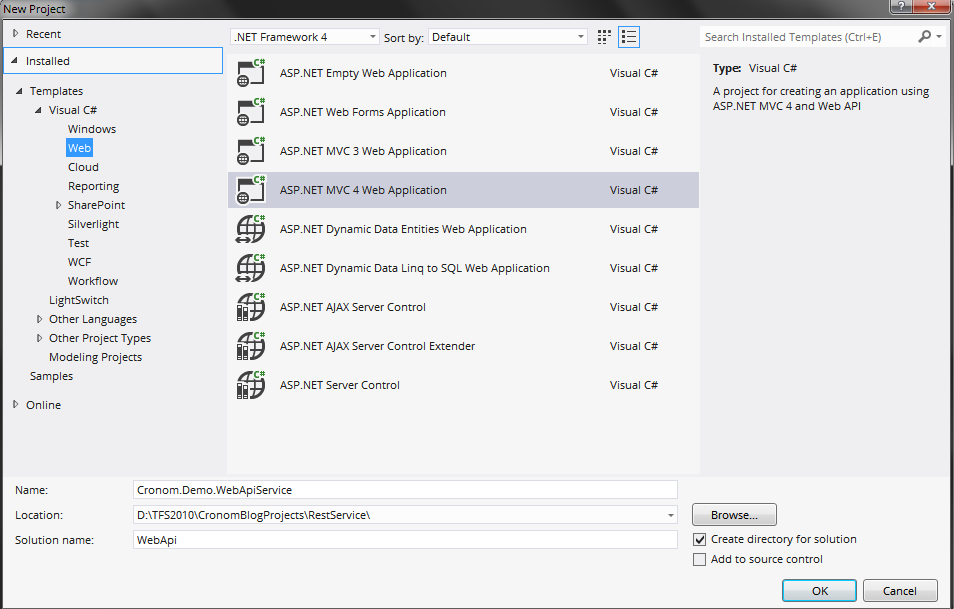
* + MVC de olduğu gibi, routing işlemlerinde kolay, kod tabanli konfigürasyon sağlar
  + Request-Response işleminde header isteğine göre içerik tipi belirlenebilir (Content-Type: application/json)
  + JSON, XML ve Atom gibi response formatlarını destekler
  + HttpRequestMessage ve HttpResponseMessage sınıflarıyla daha kullanışlı ve yönetilmesi kolay yapısı vardir, request durumuna özellestirilebilir
  + Filter ve Formatter modelleriyle genişletilebilir bir yapıya sahiptir.
  + WCF servislerindeki gibi self-host özelligi bulunuyor
  + Unit test kolaylıgı

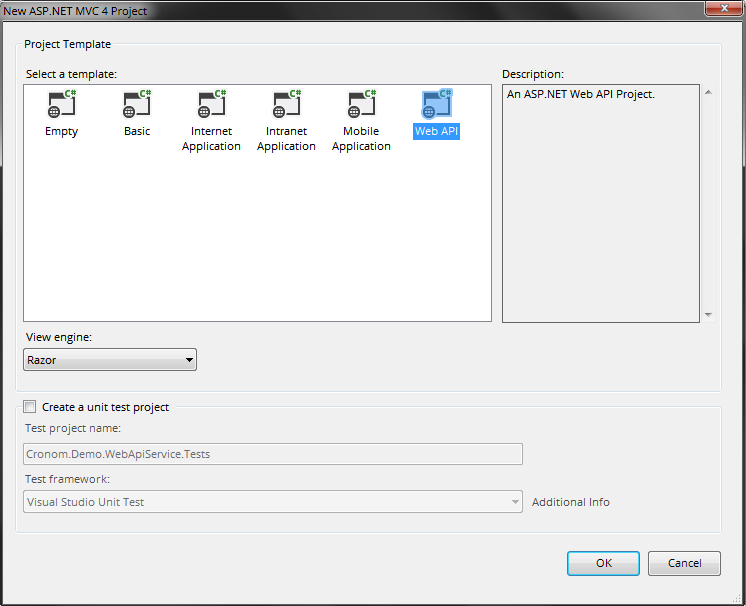
.Net 4.5 ve MVC 4, ASP.NET Web API’yi beraberinde getirmektedir. Ayrıca NuGet paket yöneticisiyle de daha sonradan sisteme yüklenebilir, bunun için **PM> Install-Package AspNetWebApi** komutunun çalıştırılmasi yeterlidir.

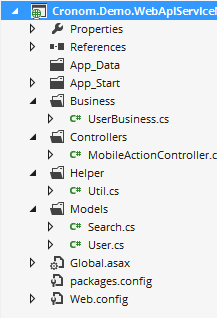
### REST with WebAPI

Yukarida HttpHandler ile oluşturduğumuz RESTful servisi bu defa ASP.NET Web API kullanarak tekrardan yapacağız. Islevsellik olarak aynı yapıda olacak, ve yine Search ve List metodlarını kullanacak; fakat Web API'nin sundugu kolayligi hep birlikte gormus olacağız.

**Demo:**Öncelikle bir MVC 4 projesi oluşturarak başlayalım. Project template olarak da Web API yi seçip ise koyulalım.





Örneğimizde herhangi bir web arayüzüne ihtiyacımız olmayacak.

Yalnızca bir ApiController kullanacağımız için mevcut template içerisinde gelen Content, Script, View klasörlerini ve var olan controller'ları silebiliriz.

HttpHandler ile yaptığımız örnekteki benzer adlandırmalar üzerinden gidecek olursak, bir tane MobileActionController adiyla boş bir web api kontrolü ekleyelim.

Ardından modellerimiz olarak yine yukarıda yer alan Search ve User sınıflarını oluşturalım.

Bir business klasörü oluşturup içerisine UserBusiness sınıfını da tanımladiktan sonra, bu noktada proje yapısı yandakine benzer bir şekilde gorunecek.

Şimdi MobileActionController içerisindeki metodlarımızı hazırlayalım. Search ve List için iki ayrı metoda gereksinimimiz var.

public class MobileActionController : ApiController

{

private UserBusiness \_userBusiness = new UserBusiness();

public object Search(Search search)

{

try

{

User user = \_userBusiness.SearchUser(search);

return new { Ok = true, User = user };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

}

}

public object List()

{

try

{

return new { Ok = true, UserList = Util.Users };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

}

}

}

Bu örnekte de verileri yine Util sınıfi içerisindeki statik değişkenler ile sağlıyoruz. Basit bir örnek olduğundan, business ve repository yapıları üzerinde çok fazla durmamıza gerek yok, ancak büyük projelerde bu yapılar duyulan ihtiyaci karşılayacak ve ileriye dönük bir şekilde implemente edilmelidir.

Global.asax üzerinde tanımli start metodu, uygulama basladığı zaman çalışir. Start metodunun içerisinde MVC'den de hatırlayacağımız route, filter, bundle konfigürasyonları için çağırımlar bulunuyor. Burada WebApiConfig bizim için önemli olan anahtar konfigürasyondur, ve aşağıdaki gibi olmalıdır;

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{action}/{id}",

defaults: new { id = RouteParameter.Optional }

);

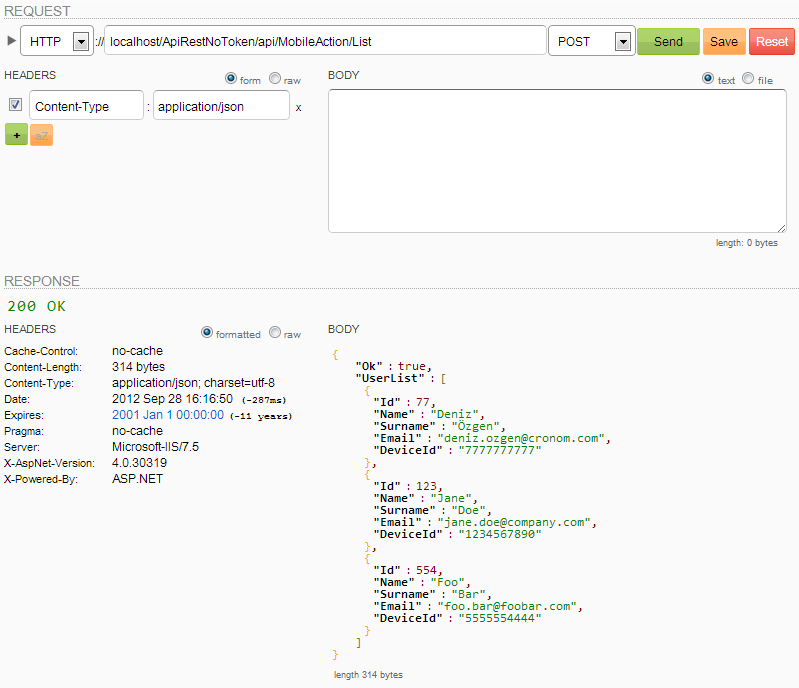
}

}

Web arayüzü kullanmayacağımız için RouteConfig ve BundleConfig ile bir işimiz olmayacak, dolayısıyla burada yapılan routing ve bundling konfigürasyonlarını temizleyebiliriz. Hazir temizlik yapıyorken, web.config de bulunan entity framework ve connection string ile ilgili alanları da kafa karıştırmaması açısından silebiliriz.

Projemizi derleyelim, RESTful servisimiz artık hazır. Herhangi bir denetim mekanizmasi olmadan, Search ve List metodlarımız artık erişilebilir durumda. Metod çağırımlarını su şekilde yapıyoruz;

**SERVER/ApiRest/api/MobileAction/List  
SERVER/ApiRest/api/MobileAction/Search , POSTDATA = {"Id":123}**



REST in Secure!

Yukarıda Web API ile oldukça basit bir RESTful servis oluşturduk ancak bu servis ile veri alışverişinde bulunacak kötü amaçlı kullanıcıların da olabileceğini aklımızdan çıkarmamamız gerekiyor, bu nedenle bir güvenlik kontrolümüzün olması şart. SSL gibi bir sifreleme, veya veri trafiginin farklı bir şekilde encrypt edilmesi, login kontrolü sağlayarak token bazli iletişim gibi bir çok alternatiflerimiz mevcut.

Biz de yukarıdaki örneğimize yine basit bir token kontrolü ekleyerek yalnizca login olan kullanıcıların api metodlarına erişmesini sağlayabilir, diğerleri için bu erişimi sınırlandırabiliriz.

Yapacağımız özetle su şekilde;

* Login metodu oluşturacağız, ve girilen email adresine göre login işlemi sonrasında client'a bir token döneceğiz.
* Client, kendisine gönderilen token'i request header içerisinde sunucuya iletecek.
* Sunucu, istemci ile haberleşirken kimin erişim izni olup olmadığını bu token ile anlayabilecek, ve buna göre yanıt dönecek.

Kontroldeki her metod içerisinde request üzerindeki token bilgisini alip islem yapmak daha fazla kod yazmamıza ve bir çok yerde kod tekrarlanmasına neden olabileceğinden, bu tur işlemler için action filter kavrami tam olarak gereksinimimizi karşılayacak anahtar bir yapı.

**Action Filter**

ASP.NET MVC yapısında bulunan Action Filter, controller içerisindeki metodların işleyişini request-response arasındaki pipeline içerisinde yönetmeyi sağlayan mekanizmadir. Authorization işleminin yani sira, loglama, validation, cache işlemleri, localization gibi işlemlerde de yararı büyüktur.

Örneğimize geri dönüp az önce bahsettiğimiz token yapısını bir TokenRequiredAttribute sınıfi yaratarak sağlayalım.

public class TokenRequiredAttribute : ActionFilterAttribute

{

private UserBusiness \_userBusiness = new UserBusiness();

public override void OnActionExecuting(HttpActionContext actionContext)

{

string token = "";

try

{

token = actionContext.Request.Headers.GetValues("Api-Token").First();

}

catch

{

actionContext.Response = CreateFailResponse();

return;

}

bool isValid = \_userBusiness.IsValid(token);

if (isValid)

{

base.OnActionExecuting(actionContext);

}

else

{

actionContext.Response = CreateFailResponse();

return;

}

}

private HttpResponseMessage CreateFailResponse()

{

return new HttpResponseMessage()

{

StatusCode = HttpStatusCode.Forbidden,

Content = new ObjectContent(typeof(object),

new { Ok = false, Message = "Unauthorized." },

new JsonMediaTypeFormatter())

};

}

}

Ve MobileActionController içerisinde Login metodunu implemente edelim.

public class MobileActionController : ApiController

{

private UserBusiness \_userBusiness = new UserBusiness();

public object Login(User user)

{

try

{

string token = \_userBusiness.Login(user.Email);

return new { Ok = true, Token = token };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

}

}

[TokenRequired]

public object Search(Search search)

{

try

{

User user = \_userBusiness.SearchUser(search);

return new { Ok = true, User = user };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

}

}

[TokenRequired]

public object List()

{

try

{

return new { Ok = true, UserList = Util.Users };

}

catch (Exception ex)

{

return new { Ok = false, Message = ex.Message };

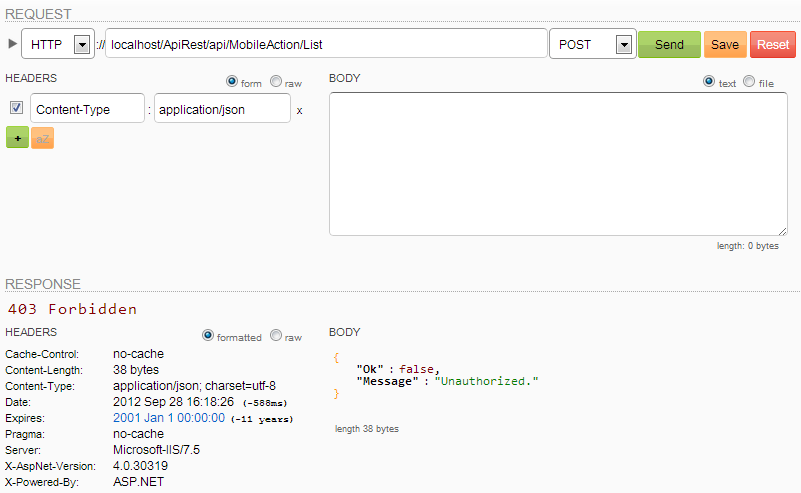
}

}

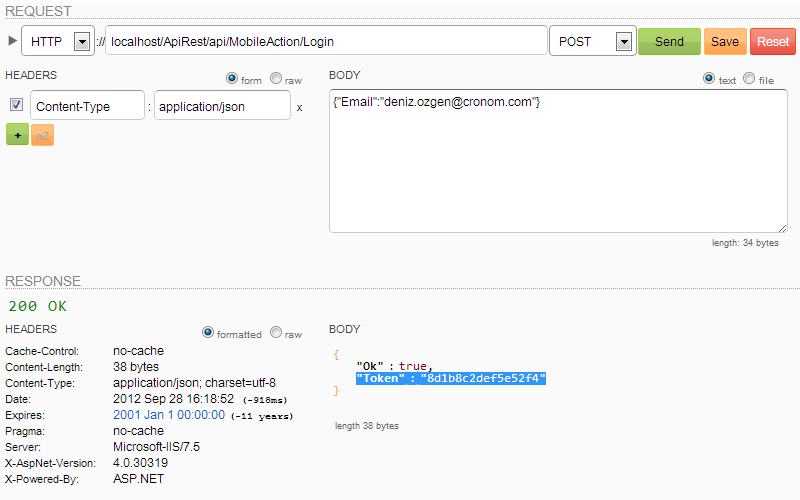
}

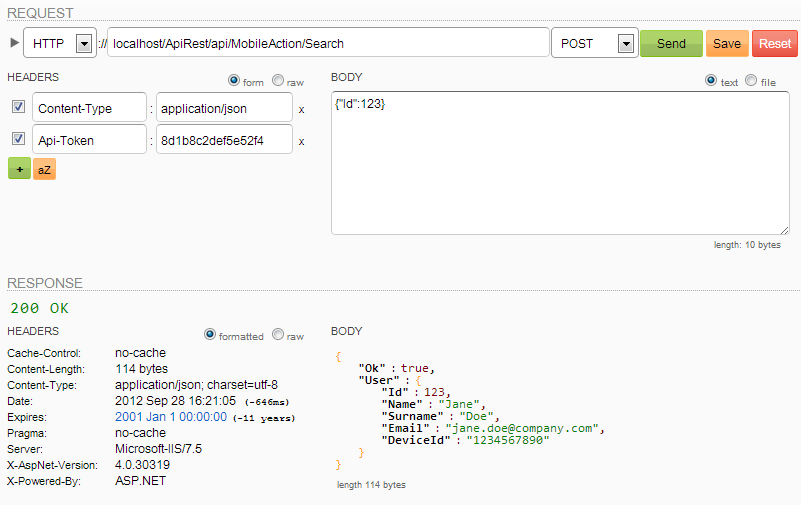
**[TokenRequired]** attribute'unu koyduğumuz her metod çalıştırilmadan önce TokenRequiredAttribute içerisindeki akişa dahil olacak ve client'in header içerisinde gönderdiği token bilgisini denetleyecektir.

Artık login olmadan veya token göndermeden api metodlarına erişmek istediğimizde şöyle bir yanıt alacağız.



Login olduğumuzda bu şekilde token alacağız ve bunu yapacağımız her isteğin header'ında tanımlayacağız.





Burada örnek olarak gösterilen her kod ve proje ekte yer almaktadir, projeleri indirip derleyerek daha detaylı olarak bu işleyişi görebilirsiniz.

Örnek projeler:  
[\_HttpRest-Project]  
[\_ApiRestNoToken-Project]  
[\_ApiRest-Project]