

İÇİNDEKİLER

SOA NEDİR? (Service Oriented Architecture)	2
SOA'NIN FAYDALARI	2
API NEDİR? (Application Programming Interface)	2
WEB SERVICE NEDİR?	3
RESTFUL SERVICE NEDİR?	3
ASP.NET WEB API.....	3
WEB API ÖZELLİKLERİ	3
Web API Tercih Etmemizin Sebepleri	3
HTTP STATUS KODLARI:	4
HTTP Methods	4
GET ve POST arasındaki Farklar	5
WEB API Projesi	5
Web API Action Result Kullanımı	7
VOID Method Kullanımı	7
ReturnType Kullanımı (Custom Object).....	8
HttpResponseMessage Kullanımı	10
IActionResult Kullanımı	10
WEB API Model Validation.....	16
CORS (Cross Origin Request Sharing)	17
Same Origin	17
CORS WebConfig Ayarı	17
Attribute Based Routing	19
Webapi Sık Kullanılan Attributelar	20
Web Api Authentication	20
OWIN Nedir	21
OAuth Nedir	21
Claim Based Authentication	22
Token Based Authentication	22
Token Based Authentication Çalışma Prensipleri.....	22
Token Based Authentication Uygulama	23
Owin Entegrasyonu	23
Cors Konfigürasyonu	24
Authorization Server Konfigürasyonu.....	25
Web Api Resource Konfigürasyonu	26
Access_Token LocalStorage Barındırma	26
HttpHeaders da Access_Token Konfigürasyonu	26
WEB API ODATA.....	27
OData Global olarak Register Etme.....	27

SOA NEDİR? (Service Oriented Architecture)

Servis Odaklı mimari veya diğer tanımlaması ile hizmet yönelimli mimari bilgisayardaki sistemlerin işlevselliklerini iş süreçleri etrafında guruplayarak sistem geliştirilmesi ve bütünleştirilmesinde yol gösteren bir yazılım tasarım felsefesidir.

Buradaki amaç birbirlerine entegre çalışacak sistemlerin olabildiğince birbirine temasını engellemek ve gevşek bağlı bir yapı oluşturmaktır. Bu hizmetler servisler sayesinde birbirleri ile veri alışverişi yapabilmektedir.



SOA'NIN FAYDALARI

- İş birimi için operasyon maliyeti düşüktür. (Servisler özerk tanımlanır) (Anatomy özelliği)
- Endüstriyel olarak standartmış ilkeleri önerir. (Uyumluluk ve Entegrasyon)
- İş süreçlerinde değişikliklere kolayca adepte olunur, pazara yeni iş fonksiyonelliklerini hızlı sunabilme imkanı vardır
- Yeni sistemlere az eforla bağlanmak mümkündür. İş ortağı firmalarla entegrasyon kolaydır.

API NEDİR? (Application Programming Interface)

Bir yazılımın başka bir yazılımda tanımlanmış işlevleri kullanabilmesi için oluşturulmuş bir tanım bütünüdür. Örneğin Facebook gönderilerimizi kendi uygulamalarımız içerisine çekmek isteyebiliriz. Bu gibi durumlarda api kullanmamız gerekir. Bir başka örnek olarak Facebook verileri ile bir 3rd (third party) Facebook uygulamaları yapmak isteyebiliriz.

Bunlar dışında OpenWeather gibi hava tahmin raporlarını veren siteler vardır. Hava tahmin raporlarını OpenWeather'ın sunmuş olduğu API üzerinden alabilir ve kendi uygulamalarımızda güncel hava tahmin raporlarını gösterebiliriz. Bu hava tahmin raporlarına göre uygulamalarımızda çeşitli işlemler gerçekleştirebiliriz.

Bir başka çok kullanılan API ise Google API servislerinden Google Maps'dir. Uygulamamızda ki restoran ve kafelerin yol tariflerini yapmak gibi bir çok uygulamada Google Maps API kullanabiliriz.

Genel olarak API'lerin Yazılım dilleri ile entegrasyonlarını içeren dökümantasyonları vardır. Bunlara SDK (**Software Development Kit**) adı verilir. Aşağıda birkaç işimize yarayabilecek apilerin linklerine yer verilmiştir.

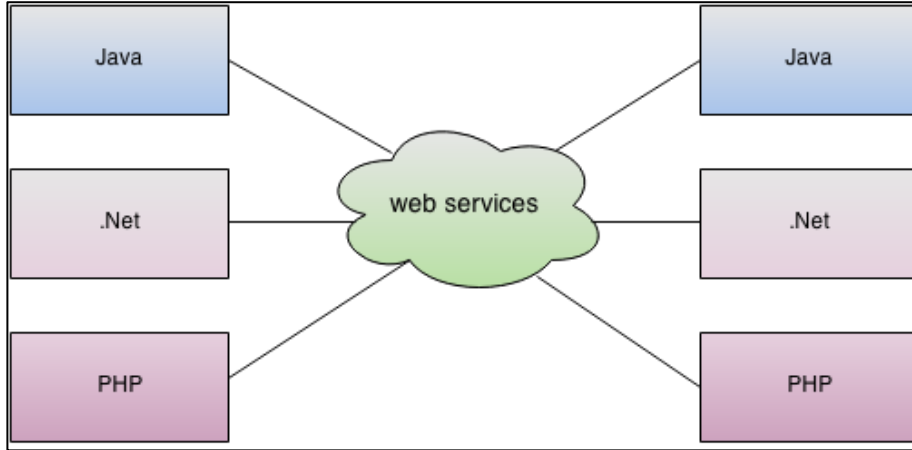
Yayıncı	Açıklaması	API Adresi
Facebook	Sosyal medya devi Facebook tarafından oluşturulmuş bir API'dir. Kullanıcı hesabınız üzerindeki işlemleri gerçekleştirebilmek için kullanılmaktadır.	https://developers.facebook.com/
Google Maps	Google haritalar servisini kullanmak ve haritalar ile ilgili işlemler yapmak için kullanılır.	https://developers.google.com/maps/
Instagram	Instagram hesabınız ile ilgili çeşitli işlemler yapabileceğiniz bir API'dir.	https://www.instagram.com/developer/

OpenWeather Hava tahmin raporlarını uygulamalarınızda kullanabileceğiniz bir API'dir.

<https://openweathermap.org/api>

WEB SERVICE NEDİR?

HTTP protokü üzerinden XML veya JSON medya tipleri ile uzak cihazlar arasındaki iletişimi sağlayan bir haberleşme yöntemidir. XML veya JSON medya tiplerinde olması sayesinde platformlar arası ve programlama dilleri arasında haberleşme sağlanabilir.



RESTFUL SERVICE NEDİR?

Web protokolleri ve teknolojilerini kullanan dağıtık bir sistemdir. Aktarılan verinin formatı HTML,JSON,XML yada farklı bir tipte olabilir. REST, bu konuda bir kısıtma getirmez. Restful servisler çoğunlukla http protokü üzerinden Web tarayıcıları tarafından sayfaların transferinde kullanılan http fiilleri (GET,POST,PUT,DELETE vs) ile haberleşirler. Aktarılan verinin tipi ve özellikleri istemci ve sunucu tarafından http protokolünde yer alan **content-type** ile tanımlanabilir.

ASP.NET WEB API

Asp.Net Web API; Microsoft'un web servis mimarisi için geliştirdiği web ve mobil tabanlı projelerimizde kullanabilmek üzere http servisleri geliştirmemizi sağlayan, platformlar arası veri alış verişini sağlamak için kullanılan bir framework'tür. Asp.Net web Api kullanarak farklı diller ile yazılmış olan uygulamalarımız web'in ortak kabul ettiği medya tiplerine (XML, JSON) çıktılar vererek birbirleri ile veri alışverişinde bulunabilir ve birbirlerine entegre sistemler olarak çalışabilirler.

WEB API ÖZELLİKLERİ

- GET,POST,PUT,DELETE gibi HTTP Methodlarını destekler.
- Kaynaktan alınan cevaplarda (**HttpResponseMessage** ismi ile kullanacağız), başarı ve hata kodları döndürerek istemciyi bilgilendirir. (**HttpStatusCode**)
- Self Hosting dediğimiz uygulama içerisinde barındırılma veya IIS üzerinde barındırma seçenekleri kullanılabilir.
- **ODATA** desteği mevcuttur. Query ile veriyi sorgulamak, filtrelemek oldukça basit ve hızlı bir yöntemdir.
- **OAuth** destekler, bu protokol sayesinde bir çok Open Authentication kullanan uygulama ile haberleşebilir. Sosyal Medya uygulamaları ile entegre çalışabilir.

Web API Tercih Etmemizin Sebepleri

- Geliştirme sürecinde WCF de olduğu kadar zahmetli ve sıkıntılı değildir.
- Http tabanlı olduğundan Restful servis geliştirmek için en iyi seçenektir.
- Open Source bir proje olup, sürekli geliştirilmektedir.
- Exception Handling özelliği WCF göre daha gelişmiştir.
- Sunucu tarafı gelişmiş durum kodlarına sahiptir.

HTTP STATUS KODLARI:

Web İstemci (web-client), web sunucu (web-server) a Http protokolü üzerinden istek gönderdiğinde sunucudan dönen cevap durum kodları içerebilir. En çok karşılaşılabilecek durum kodları ve açıklamalarına aşağıda yer verilmiştir.

Kod	Mesaj	Anlamı
1xx	Başarı Durum Kodları	
100	Continue	Devam
101	Switching Protocols	Anahtarlama Protokolü
102	Processing	İşlem
200	OK	Kaynağa ulaşıldı. Başarı Kodu
201	Created	Kaynak Oluşturuldu
202	Accepted	Web isteği sunucu tarafından kabul edildi.
203	Non-Authoritative Information	İstek Yetersiz bilgi içeriyor
204	No-Content	İçerik Yok (void methodlar)
3xx	Yönlendirme Durum Kodları	
300	Multiple Choices	Çok Seçenek
301	Moved Permanently	Kalıcı Taşındı
302	Moved Temporarily	Geçici Taşındı
303	See Other	Diğer sonuçlara Bak
4xx	İstemci Durum Kodları	
400	Bad Request	Kötü İstek, (yapılan isteğin sunucu daki formata uymadığı durumlarda kullanılır, ModelValidation)
401	Unauthorized	Kişinin kaynağa erişim yetkisi olmadığı durumlarda tercih edilir.
402	Payment Required	Ödeme gerekli, Ödeme işlemlerini servis üzerinden yaptığımız durumlarda istemciye dönen bir mesajdır.
403	Forbidden	Kaynak var ama erişmek için izin var fakat sunucu isteği red ediyor.
404	Not Found	Sayfa Bulunamadı
405	Method not Allowed	Kaynağa yapılan HttpMethod fiili mevcut değil
415	Unsupported Media type	İstemci tarafından desteklenemeyen medya formatı
5xx	Sunucu Durum Kodları	
500	Internal Server Error	Sunucuda Hata oluştu
501	Not Implemented	Sunucudaki kaynak içinde tanımlanmış method içine kod yazılmadığı durumlarda oluşur.
503	Service Unavailable	Sunucuya erişilemiyor (Ya sunucu kapasitesini aştı yada şuan çöktü)

HTTP Methods

GET : Kaynağa url üzerinden erişmek için kullanılan http Method'tur.

POST : İşlenecek verileri belirli bir kayna URI göndermemizi sağlar. Şuan sunucuda bulunmayan ilk defa işlenecek veriler için tercih edilir.

PUT : Belirtilen kaynak URI de işlenecek olan verilen yüklemesini sağlar. Veri güncelleme işlemlerinde tercih edilir. Post'a göre daha performanslı çalışır.

DELETE : Belirtilen kaynak URI de silinecek olan veriler için tercih edilir.

HEAD : GET isteğine benzer fakat, sadece istek ile ilgili veya cevap ile ilgili bilgiler head istediğinde taşınır.

OPTIONS: Sunucunun desteklediği HTTP METHOD'ları döndürür.

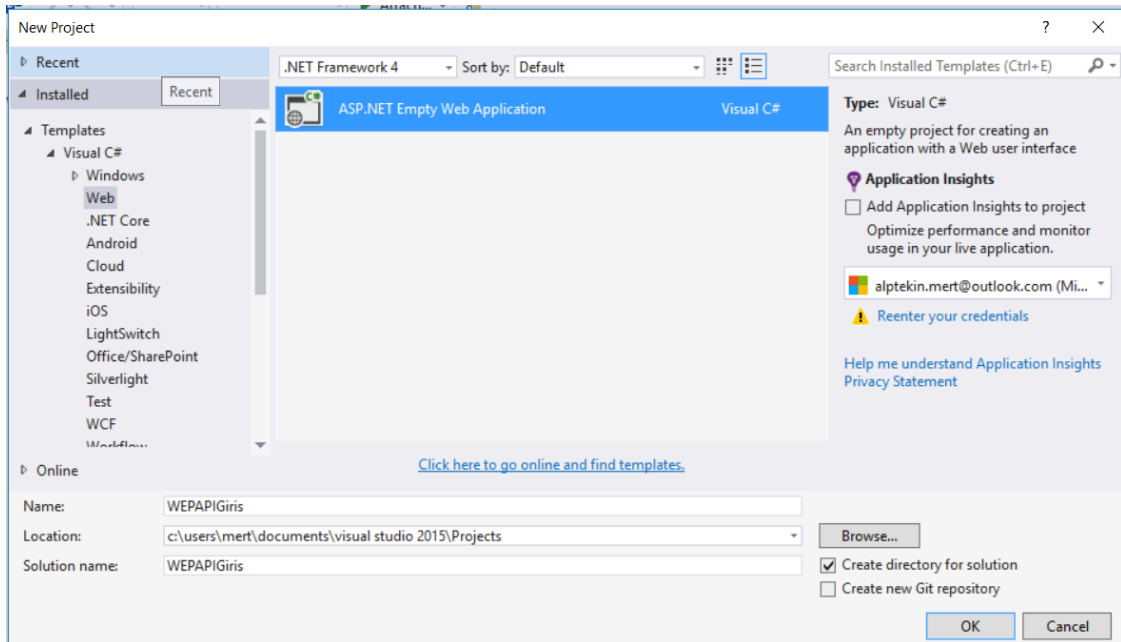
CONNECTS : TCP/IP kanalı üzerinden sunucu ile istemci arasındaki açılan bağlantı bilgisini döndürür.

GET ve POST arasındaki Farklar

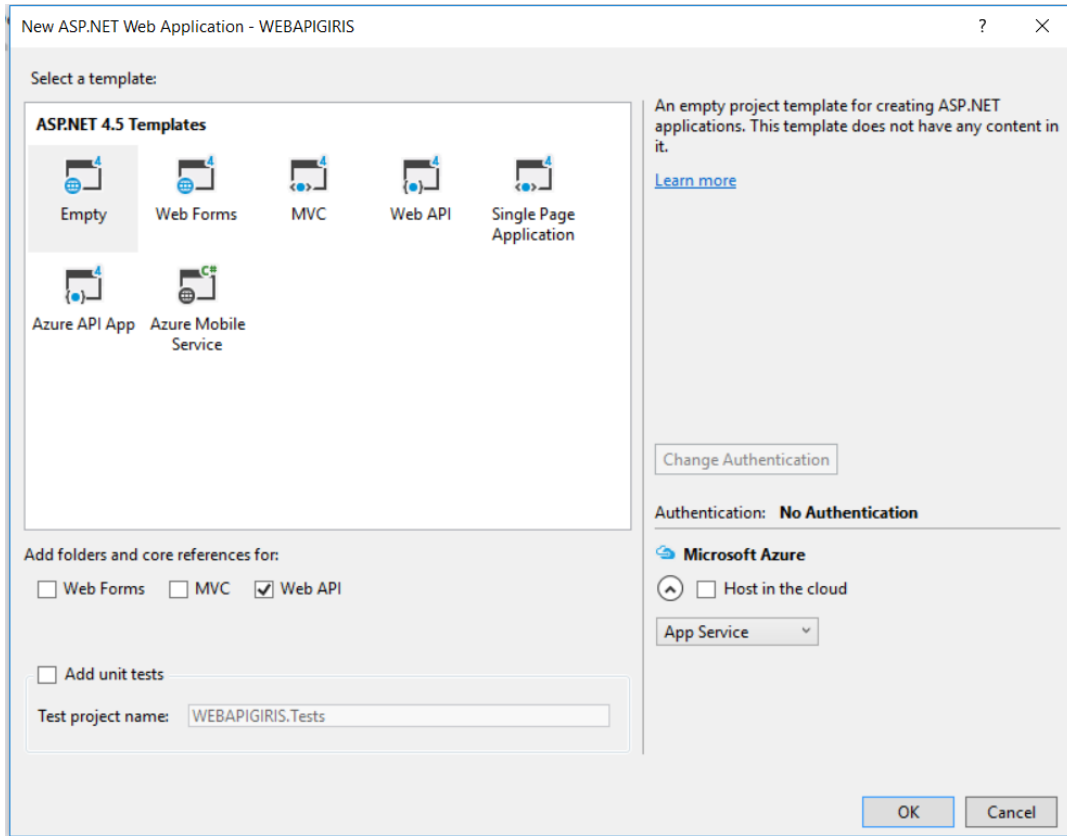
	GET	POST
BACK BUTTON	Geri Butonuna basmanın sunucu ve istemci arasındaki haberleşmeye bir zararı yoktur	Veri yeniden işlenecektir. (DİKKAT)
CACHED	Veri ramde tutulabilir.	Veri ramde tutulamaz
Encoding type	application/x-www-form-urlencoded tipinde istekler yapılabilir.	application/x-www-form-urlencoded veya multipart/form-data
History	Veriler tarayıcının geçmişinde saklı kalır	Veriler Tarayıcı geçmişinde saklanmaz
Security	Post ta göre veri taşımada daha az güvenli bir yöntemdir. Çünkü veri URL üzerinden taşınır. Sensitive veri veya parola gibi önemli verilerimizi taşımak için tercih edilmez	Hassas verilerimizin iletiminde tercih edilir.
Visibility	Veri taşıma sırasında URL den görünür.	Veri URL üzerinden görüntülenemez.

WEB API Projesi

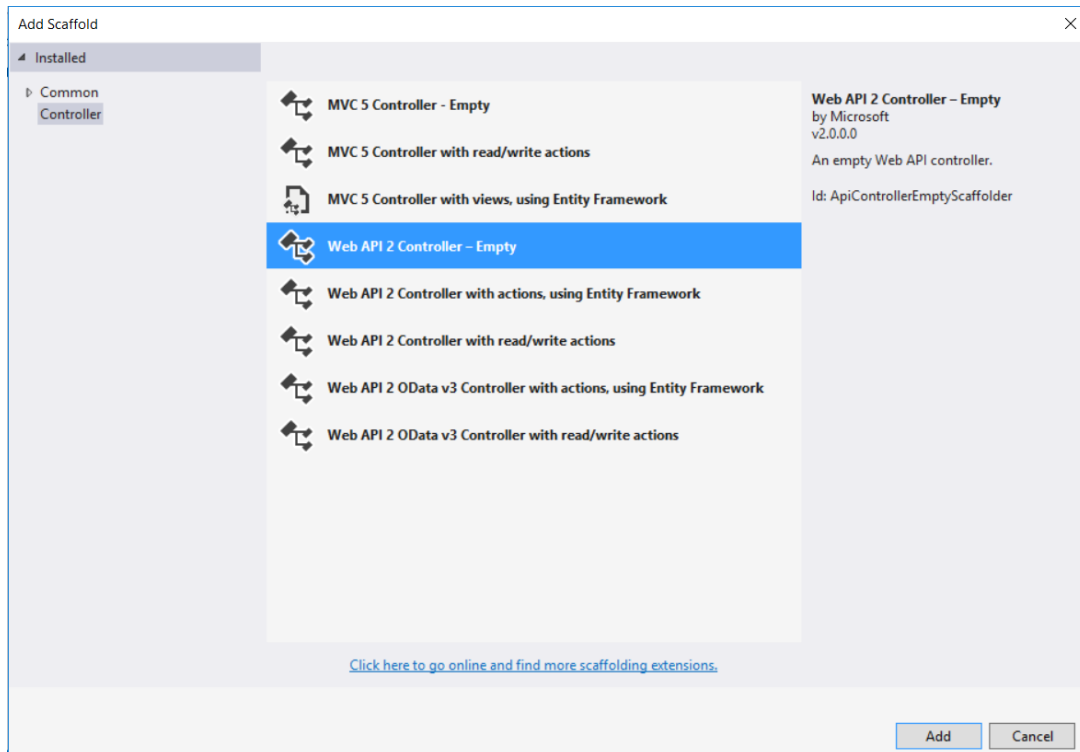
Web API hakkında yeterli bilgiyi edindikten sonra bir proje üzerinde Web API kullanmaya başlayabiliriz. ASP.NET Web api kullanarak proje geliştirmek için aşağıdaki adımları izleyelim.



Daha sonra gelen pencerenden Empty project seçilerek alttaki seçim kutusundan WEB API seçeneğini işaretleyin. Bu seçenek oluşturulan proje içerisinde Web API kullanımına uygun kütüphanelerin referans edilmesini sağlayacaktır. Ayrıca Empty template'ini seçtiğimizden dolayı boş bir Web API projesi oluşturacağız.



Daha sonra açılan projeden Controllers klasörü üstüne gelip. Add Controller seçeneklerini seçerek. Aşağıda açılan penceren Web API 2 Controller Boş template seçimi yaparak açılan pencereden TestController ismini vererek projemize api controller açmış oluruz.



Add Controller

Controller name:

Add
Cancel

Örneğimizde gördüğümüz gibi TestController ile uygulamamız için web isteklerimizi yakalayacağımız bir yapı oluşturduk. TestController içerisindeki kodları incelediğimizde ApiController sınıfından miras alarak oluşturulduğunu görebiliriz. ApiController sınıfı bir apinin göstermesi gereken özelliklere sahiptir.

ApiController'dan miras alan TestController ile belirli dönüş tipleri kullanarak web sunucusundan istemciye belirli medya tiplerinde sonuç gönderilir. Şimdi bu parta gelen istekleri işleyeceğimiz Action Result tiplerinden ve bu istekleri yapmamızı sağlayan methodların kullanımından bahsedeceğiz.

Web API Action Result Kullanımı


Web api uygulamamızda web-client, web-server daki kaynak kodlara Http protokolü üzerinden Http methodlarını kullanarak ulaşır. Bu bölümde Web API'nin HttpRequest'leri nasıl işleyeceği ve HttpResponse oluşturken ne gibi işlemlerden geçileceği, sunucu cevap olarak hangi medya formatları destekleyeceğinden bahsedilecektir. HttpRequest isteklerini HttpVerbs dediğimiz http fiilleri kullanarak yaparken **POSTMAN** denilen Chrome üzerine Extension olarak kurabileceğimiz bir eklenti olup, http trafiği üzerinde işlem yapabilmemizi sağlayan program kullanılacaktır. ASP.Net web api 4 farklı ActionType desteklemektedir.

VOID Method Kullanımı

Void methodlar gelen isteğin fiiline göreHttpGet, HttpPost, HttpPut, HttpDelete HttpVerbs attributeleri ile kullanılabilir. **HttpResponseMessage** olarak **HttpStatus Code** : 204, No-Content gönderilir. Genelde Api de bir işlemi run etmek için veya belli bir konfigürasyonu tetiklemek için tercih edilir. Veri alış-verişinde kullanamamızın sebebi sunucudan istemciye yapılan işlem ile ilgili herhangi bir sonuç döndürmemesidir. Örnek kullanım.

TestController.cs

```
// HttpGet ile URL üzerinden HttpRequest yapacağımızı söyledik.
// localhost:8xxx:api/Test URL postman ile istekte bulunun dönen sonucu görelim
[HttpGet]
public void Data()
{
}
```


Postman
sunan: www.getpostman.com
★★★★★ (7835) | [Geliştirici Araçları](#) | 3.246.862 kullanıcı

↑ UYGULAM. BAŞLAT

GENEL BAKIŞ
YORUMLAR
DESTEK
ALAKALI

G+1 2.555

Chrome üzerine kurabileceğimiz bir eklenti olup, http trafiği üzerinde işlem yapabilmemizi sağlayan postmani kuruyoruz Web sunucusuna İsteklerimizi postman ile göndereceğiz.

GET

localhost:14924/api/Test

Params

Send

Save

AuthorizationHeadersBodyPre-request ScriptTestsCode

TypeNo Auth

BodyCookiesHeaders (8)TestsStatus: 204 No ContentTime: 27 ms

PrettyRawPreviewText

1

Yukarıda ki gibi HttpGet fiili seçilerek url'den istekde bulunduk. Status 204 No Content döndüğüne dikkat edelim. İsteğimize 27 ms de cevap verilmiştir. **HttpResponseMessage** yoktur. Headers kısmında sunucudan aldığımız bilgiler ise aşağıdaki gibidir. Headers Web-Server ile Web-Client arasında her bir haberleşmede HttpProtokolü üzerinden taşınan bilgilerin gönderildiği HttpVerb'tür.

```
Cache-Control →no-cache
Date →Sun, 26 Feb 2017 05:49:29 GMT
Expires →-1
Pragma →no-cache
Server →Microsoft-IIS/10.0
X-AspNet-Version →4.0.30319
X-Powered-By →ASP.NET
```

ReturnType Kullanımı (Custom Object)

ReturnType döndüren methodlar gelen isteğin fiiline göre HttpGet, HttpPost, HttpPut, HttpDelete HttpVerbs attributeleri ile kullanılabilir. Kaynağa ulaşıldığı takdirde **HttpStatus Code** ve **HttpResponseMessage** olarak XML ve JSON medya formatında veri istemciye gönderilir. Kendi sınıflarımızdan oluşturduğumuz modelleri dışarı açtığımız senaryolarda tercih edilebilir. Varsayılan olarak WebApi aldığı isteklerin cevaplarını XML medya formatında verecektir. Fakat bu ayar yapılan isteğe göre değiştirilebileceği gibi aynı zamanda Global olarak proje bazlıda değiştirilebilir.

TestController.cs

```
// HttpGet Verb yazmamamıza rağmen bu istek HTTPGET isteği olarak sunucu tarafından kabul edilecektir. Sebebi ise Method ismi olarak HTTPVERB olan GET, POST, PUT, DELETE kullanmamız yeterlidir. Kaynağın üzerine HTTPVERBS attributelerini yazmamıza gerek yoktur.

// localhost:8xxx:api/Test URL postman ile istekte bulunun dönen sonucu görelim

public List<PersonModel> GetPersons()
{
    return PersonList;
}
```

Uyarı : Eğer bir controller içerisinde aynı method imzasına sahip iki farklı aynı fiilde bir kaynak varsa apicontroller hangi kaynağın istemci tarafından istendiğini bilemez. Bu sebeple Multiple action were Found hatası verir. Bu durumda WebapiConfig.cs dosyasından route düzenlemesi yapılmalıdır.

GET localhost:14924/api/Test/GetPersons Params Send

Authorization Headers Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (10) Tests Status: 500 Internal Server Error

Pretty Raw Preview JSON

```

1 {
2   "Message": "An error has occurred.",
3   "ExceptionMessage": "Multiple actions were found that match the request: \r\nData on type ApiOgreniyorum.Controllers
4   .TestController\r\nGetPersons on type ApiOgreniyorum.Controllers.TestController",
5   "ExceptionType": "System.InvalidOperationException",

```

Yukarıda görüldüğü gibi route dosyamızda herhangi bir ayara yapmadığımızda sunucuya yaptığımız istek bize Internal Server Error Status Code ile geri dönüş yapmıştır.

```

public static class WebApiConfig
{
    1 reference
    public static void Register(HttpConfiguration config)
    {
        // Web API configuration and services

        // Web API routes
        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            //routeTemplate kısmında {controller} dan sonra {action}
            //eklenip kaynakların aynı imzaya sahip olduğunda birden fazla
            //sonuç döndürmesinin önüne geçilmiş oldu.
            routeTemplate: "api/{controller}/{action}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}

```

WebApiConfig dosyasında yukarıda yapılan konfigürasyon işleminden sonra web sunucuna istekte bulunuyoruz.

GET localhost:14924/api/Test/GetPersons Params Send

Authorization Headers (1) Body Pre-request Script Tests

Accept application/xml Bulk Edit

key value

Body Cookies Headers (10) Tests Status: 200 OK

Pretty Raw Preview XML

```

1 <ArrayOfPersonModel xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Api
2   <PersonModel>
3     <Name>Ali</Name>
4     <SurName>Tan</SurName>
5     <Title>Müdür</Title>
6   </PersonModel>
7   <PersonModel>
8     <Name>Mehmet</Name>
9     <SurName>Karabaş</SurName>
10    <Title>Uzman</Title>
11  </PersonModel>
12 </ArrayOfPersonModel>

```

Yukarıdaki gibi HttpResponseMessage Body'sinde XML medya tipinde verilerimizin döndüğünü gördük.

Uyarı : Eğer herhangi bir MediaType Formatter ayarı yapmadıysanız, WebApi varsayılan da veriyi XML medya tipinde serileze edecektir.

HttpResponseMessage Kullanımı

HttpResponseMessage döndüren methodlar gelen isteğin fiiline göre HttpGet, HttpPost, HttpPut, HttpDelete HttpVerbs attributeleri ile kullanılabilir. Kaynağa ulaşıldığı takdirde **HttpStatusCode** ve **HttpResponseMessage** olarak XML ve JSON, PlainText, Html vs medya formatlarından biri halinde veri istemciye gönderilir. ReturnType methodlardan farkı erişilen kaynaktaki logic'e göre HttpStatusCode döndürebilir veya Response olarak taşınacak verinin medya tipini belirleyebiliriz. **HttpRequest.CreateResponse()** methodu ile kedimiz gelen talebe bir cevap üretebiliriz. Yani HttpResponseMessage dönüş tipi sayesinde gelen talebe http protokolü üzerinden direk bir cevap mesajı oluşturmamızı sağlar ve bu sayede gönderilecek olan cevap üzerinden bir çok kontrol yapabilme imkanımız olur.

TestController.cs

```
// localhost:8xxx:api/Test/getcontent URL postman ile istekte bulunun dönen sonucu görelim
public HttpResponseMessage GetContent()
{
    HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, "value");
    response.Content = new StringContent("hello", Encoding.Unicode);
    response.Headers.CacheControl = new CacheControlHeaderValue()
    {
        MaxAge = TimeSpan.FromMinutes(20)
    };
    return response;
}
```

Response.Content ile isteğe dönecek olan cevabı string tipinde döndürüyoruz. Kaynağa yapılan isteği de 20 dakikalığına cache'e alıyoruz. Tabi burada string Content yerine medya tipi belirtilerek kendi döndürmek istediğimiz modelleri de cevap olarak döndürebiliriz.

Response

```
HTTP/1.1 200 OK
Cache-Control: max-age=1200
Content-Length: 10
Content-Type: text/plain; charset=utf-16
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 08:53:35 GMT
hello
```

Yukarı da görüldüğü gibi cevap text/plain dönüş tipi olarak gelmiştir. Kaynağın cache'ten okunacağı maksimum süre 20 dk olarak sunucudan cevap olarak gönderilmiştir. Bu bilgiler sunucudan gelen cevabın Headers ile gönderilken. Hello cevabı ResponseBody dediğimiz cevabın gövdesinde gönderilmiştir.

IHttpRequestResult Kullanımı

IHttpRequestResult olarak döndüren methodlar gelen isteğin fiiline göre HttpGet, HttpPost, HttpPut, HttpDelete HttpVerbs attributeleri ile kullanılabilir. Kaynağa ulaşıldığı takdirde **HttpStatusCode** ve **HttpResponseMessage** olarak XML ve JSON, PlainText, Html vs medya formatlarından biri halinde veri istemciye gönderilir. HttpResponseMessage veri tipi ile döndürebileceğimiz cevapları bu tip ile döndürmemiz mümkündür. Web API 2.0 ile birlikte sıkça kullanılan async çalışan bu dönüş tipi genelde tercih edilmektedir. Tercih edilmesinin sebebine gelince;

- HttpResponseMessage Fabrikası olarak kullanılan bir arayüzdür. Bu arayüzden kendi medya formatlarımızı döndürebiliriz.

- Bu arayüzden implemente olmuş farklı sınıflar sayesinde unit testi daha kolay yapılan ve birbirlerinden tümüyle soyutlanmış olan sınıflar ile çalışabilme imkanı sunar.
- Medya tipini döndüren sınıfın logiclerinden controller'ın action'ı içerisindeki yapıyı tümü ile birbirinden ayırarak temiz bir kullanım sağlar. Ve alt seviye detaylardan controller içindeki actionları kurtarmış olur.

Tek bir method içeririr.

C#

```
public interface IHttpActionResult
{
    Task<HttpResponseBody> ExecuteAsync(CancellationToken cancellationToken);
}
```

C#

```
public class TextResult : IHttpActionResult
{
    string _value;
    HttpRequestMessage _request;

    public TextResult(string value, HttpRequestMessage request)
    {
        _value = value;
        _request = request;
    }

    public Task<HttpResponseBody> ExecuteAsync(CancellationToken cancellationToken)
    {
        var response = new HttpResponseMessage()
        {
            Content = new StringContent(_value),
            RequestMessage = _request
        };
        return Task.FromResult(response);
    }
}
```

Not: Yukarıdaki örnek ile IHttpActionResult arayüzünü kendi sınıflarımıza implement ederek kendi dönüş tiplerimizi tanımlayabildiğimizi görmüş olduk.

C#

```
public IHttpActionResult GetCustomType()
{
    return new TextResult("hello", Request);
}
```

Postman programı ile api/test/getcustomtype URI istek yaparak dönen sonucun aşağıdaki gibi oluştuğunu gözlemleyelim.

TestController.cs

HTTP/1.1 200 OK

```
Content-Length: 5
Content-Type: text/plain; charset=utf-8
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 08:53:35 GMT
hello
```

Bunun dışında `IHttpActionResult` tipinde yapılan isteklerde aşağıdaki dönüş tipleri daha önceden tanımlı olduğu için action içerisinde daha temiz kod yazabilme imkanı sağlamaktadır.

```
NotFound(model); // Kaynak bulunamadı
Ok(model) // Kaynağa erişildi
BadRequest(modelState); // Validasyon Hatası Mevcut
InternalServerError(error); // Sunucuda run-time hata oluştuğunda
Unauthorized(message); // Kaynağa erişim yetkisi yok
Json(model); // modelimizi JSON medya tipine serilize eder
```

Şimdi bir örnek üzerinden `IHttpActionResult` arayüzünün dönüş tiplerini kullanarak nasıl daha temiz kodlar yazacağımızı görebileceğimiz bir örnek yapalım. Günlük haber girişi yaptığımız bir API mizin olduğunu ve buradan bir çok haber sağlayıcına son dakika haberleri verdiğimizizi düşünelim. NewsController adında bir ApiController üzerinden JSON ve XML medya tiplerinde veri çıkış noktası (endpoint) açalım. Haberlerin dışarı çıkartacağımız model

NewsModel.cs

```
namespace NewsAPI.Models.Model
{
    public class NewsModel
    {
        public int NewsID { get; set; }

        [Required(ErrorMessage = "Haber Başlığı Giriniz")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Haber Kısa içeriği giriniz"), MaxLength(100, ErrorMessage = "100 karakter girebilirsiniz")]
        public string Content { get; set; }

        [Required(ErrorMessage = "Haber içeriği giriniz.")]
        public string FullContent { get; set; }

        [IgnoreDataMember] // Eğer modelden dışarı serilize olmasını istemediğimiz üye var ise bu attribute kullanılır.
        public DateTime PostDate { get; set; }

        public int CID { get; set; }
    }
}
```

Aşağıdaki kod ise dışarı çıkartacağımız modelin Entity'sidir. Entityleri dışarıya direk çıkarmamızın sebebi tablolarımızdaki her bir alanın dışarı çıkmasını istememek veya dışarı çıkan verilerimizin güvenlik açısından tablodaki sütun alanlarının isimleri ile aynı olmamasını sağlamaktır. Bu sayede veritabanına yapılan herhangi bir

atağa karşı yazılımsal olarak bir açık oluşturmamış olacağız. Bunun dışında validasyon işlemlerimizi modeller üzerinden yapıp Entityleri kirlenmemektir. Kurumsal Projelerde bu tarz yapılar için Modellerimizi Entitylere çeviren yada tam tersi işlem yapan Mapper'lar kullanılabilir.

News.cs

```
namespace NewsAPI.Models.Data
{
    public class News
    {
        [Key]
        public int ID { get; set; }
        public string Title { get; set; }
        public string Short Content { get; set; }
        public string Body { get; set; }
        public int Category ID { get; set; }
        public DateTime? Create Date { get; set; }

        [ForeignKey("Category ID")]
        public virtual News Category Category { get; set; }
    }
}
```

Son olarak CodeFirst ile oluşturduğumuz yapımızın veritabanı bağlantıları için aşağıdaki kodu da ProjectContext.cs içerisinde yazdık.

ProjectContext.cs

```
public class ProjectContext:DbContext
{
    public ProjectContext()
    {
        Database.Connection.ConnectionString = "Server=.;Database=NewsDb;uid=sa;pwd=123";
    }

    public DbSet<News> News { get; set; }
    public DbSet<News Category> Categories { get; set; }
}
```

Aşağıdaki action da ise haberlerimizi liste olarak veritabanımızdan çeken GetNews adında bir method ile endpoint oluşturmuş olduk.

Uyarı : Aşağıdaki kodlar db , database instance alındığı varsıyalarak yazılmıştır.

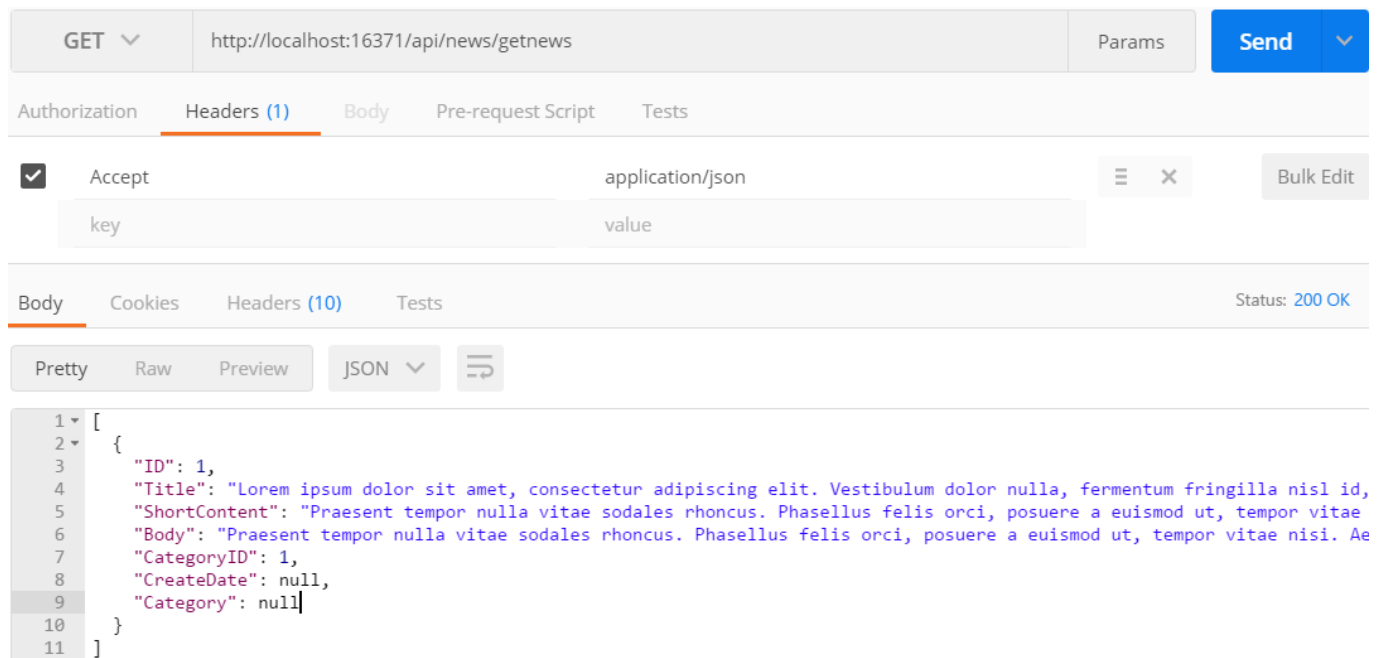
NewsApiController.cs

```
public IActionResult GetNews()
{
    ProjectContext db = new ProjectContext();
    db.Configuration.ProxyCreationEnabled = false;
    // database deki ilişkileri koparmamızı sağlar. Bu sayede serilization işleminde
```

```
exception oluşmaz
    var model = db.News.Select(a => new NewsModel
    {
        Name = a.Title,
        Content = a.ShortContent,
        CID = a.CategoryID,
        FullContent = a.Body

    }).ToList();
    return Ok(model);
}
```

Şimdi ise Postman kullanarak ilgili kaynağa Get isteğinde bulunarak verilerimizin geldiğini gözlemleyelim.



The screenshot shows the Postman interface. The request method is GET, and the URL is http://localhost:16371/api/news/getnews. The response status is 200 OK. The response body is displayed in JSON format, showing a list of news items. The first item has an ID of 1, a title, short content, full content, category ID, and creation date.

```
1 [
2   {
3     "ID": 1,
4     "Title": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum dolor nulla, fermentum fringilla nisl id,
5     "ShortContent": "Praesent tempor nulla vitae sodales rhoncus. Phasellus felis orci, posuere a euismod ut, tempor vitae
6     "Body": "Praesent tempor nulla vitae sodales rhoncus. Phasellus felis orci, posuere a euismod ut, tempor vitae nisi. Ae
7     "CategoryID": 1,
8     "CreateDate": null,
9     "Category": null]
10 }
11 ]
```

Gördüğümüz gibi Web Server 'a istek yaparken application/json medya formatında veri isteğinde bulunduk ve HttpStatusCode : 200 Ok döndü ve Response Body'sinde ise Json olarak verinin döndüğünü görüyoruz.

Şimdi ise NewsApiController altına PostNews adında bir action oluşturup, server-side validasyon yaparak veri kaynağımıza nasıl veri göndereceğimizin örneğini yapalım.

NewsApiController.cs

```
public IActionResult PostNews(NewsModel model)
{
    ProjectContext db = new ProjectContext();

    if (ModelState.IsValid)
    {
        var entity = new News();
        entity.CategoryID = model.CID;
        entity.Body = model.FullContent;
        entity.ShortContent = model.Content;
        entity.Title = model.Name;
```

```
        db.News.Add(entity);
        db.SaveChanges();

        return Json(model);
    }

    return BadRequest(ModelState);
}
```

POST

http://localhost:16371/api/news/postnews

Params

Send

Authorization

Headers (1)

Body

Pre-request Script

Tests

☐ form-data

☒ x-www-form-urlencoded

☐ raw

☐ binary

<input checked="" type="checkbox"/>	Name	Başlık-1	
<input checked="" type="checkbox"/>	Content	Test İçerik	
<input checked="" type="checkbox"/>	FullContent	Test Tüm İçerik	
<input checked="" type="checkbox"/>	CID	1	
<input checked="" type="checkbox"/>	key	value	
	key	value	

Body

Cookies

Headers (10)

Tests

Status: 200 OK

Pretty

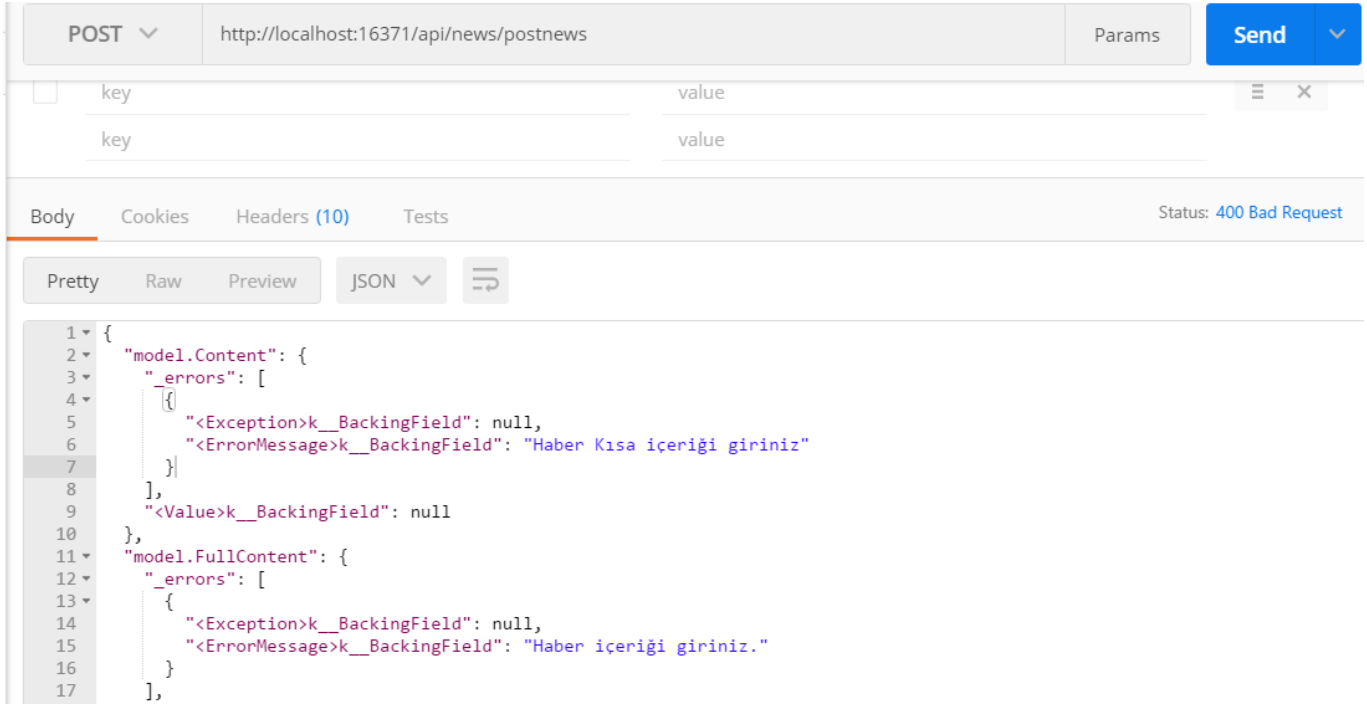
Raw

Preview

JSON

```
1 {
2   "NewsID": 0,
3   "Name": "Başlık-1",
4   "Content": "Test İçerik",
5   "FullContent": "Test Tüm İçerik",
6   "CID": 1
7 }
```

Evet yukarıda ki sonuçtan anlaşıldığı gibi form üzerinden gönderdiğimiz model başarılı bir şekilde veri kaynağımıza ulaştı. Şimdi de modelimizdeki zorunlu olanları istediğimiz formatta sunucuya göndermediğimiz durumda web sunucunun nasıl bir HttpResponseMessage döndüreceğini gözlemleyelim.



POST http://localhost:16371/api/news/postnews

Params

Send

key value

key value

Body Cookies Headers (10) Tests Status: 400 Bad Request

Pretty Raw Preview JSON

```
1 {
2   "model.Content": {
3     "_errors": [
4       {
5         "<Exception>k__BackingField": null,
6         "<ErrorMessage>k__BackingField": "Haber Kısa içeriği giriniz"
7       }
8     ],
9     "<Value>k__BackingField": null
10  },
11  "model.FullContent": {
12    "_errors": [
13      {
14        "<Exception>k__BackingField": null,
15        "<ErrorMessage>k__BackingField": "Haber içeriği giriniz."
16      }
17    ],
18  }
```

WEB API Model Validation

ModelState.IsValid ile server-side modellerimizin validasyon kontrolünü api katmanında da yapabiliriz. Ama model validasyonlarını tek bir yerden yönetmek istiyorsak bu durumda ActionFilterAttribute kullanarak kendi ModelValidasyon attribute'umuzu oluşturabiliriz. Bu durumda sadece istekde bulunduğumuz method üstünde tanımlayacağımız attribute sayesinde model validasyonları merkezi bir yerden yönetme imkanı sağlayacağız. Aşağıdaki örnek bu işlemi nasıl yapacağımız ile ilgili kodları içermektedir.

ModelValidAttribute.cs

```
public class ModelValidAttribute:ActionFilterAttribute
{
    public override void OnActionExecuting(HttpContext actionContext)
    {
        if (!actionContext.ModelState.IsValid)
        {
            actionContext.Response =
            actionContext.Request.CreateResponse(System.Net.HttpStatusCode.BadRequest,
            actionContext.ModelState);
        }
    }
}
```

ActionFilterAttribute : Her hangi bir action'a girmeden önce actionfilter attribute ile model kontrolü yapıp buna göre api'den farklı HttpResponseMessage'lar döndürmemize olanak tanır. OnActionExecuting ve OnActionExecuted tiye virtual tanımlanmış iki methodu vardır. Kendi sınıflarımızda ActionFilterAttribute den miras alarak methdolarımızın nasıl çalışacağını belirtebiliriz.

HttpContext sınıfı ise o an api de istek de bulunduğumuz action kapsamına girip, bu kapsama parametre olarak geçilen modelin validasyonunu kontrol edebilir. Yukarıda validasyon kontrolünden geçemediğimiz durumda ModelState HttpStatusCode 400 olarak gönderen bir kullanım örneği görmekteyiz. yönetme imkanı sağlayacağız. Aşağıdaki örnek bu işlemi nasıl yapacağımız ile ilgili kodları içermektedir.

Son olarak Post methodmuzu aşağıdaki gibi güncelleyebiliriz.

NewsApiController.cs

```
[ModelValid]
public IHttpActionResult PostNews(NewsModel model)
{
    var entity = new News();
    entity.CategoryID = model.CID;
    entity.Body = model.FullContent;
    entity.ShortContent = model.Content;
    entity.Title = model.Name;

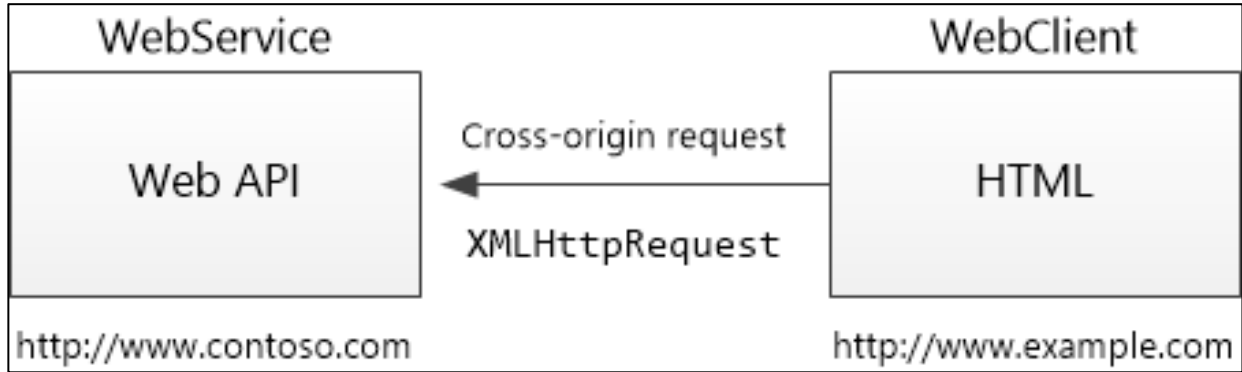
    context.News.Add(entity);
    context.SaveChanges();

    return Json(model);
}
```

CORS (Cross Origin Request Sharing)

Web uygulamaların birbirleri konuşmaları için kaynak paylaşımı izinlerinin yapılması gerekir. Kaynak paylaşımı yaparken;

- Hangi domainlerin bu kayanağa erişebileceğini
- Hangi methodların hizmet vereceğini
- Hangi medya tipinde veri çıkışına izin vereceği gibi senaryolarımız vardır.



Same Origin

Uygulama aynı domainde ise aynı origindedir. Aynı domainde olan uygulamalar arasında kaynak paylaşımı için özel bir konfigürasyon yapma ihtiyacı yoktur. Aşağıda aynı origin de olan kaynaklara yer verilmiştir.

1. **http://www.example.net - domain**
2. **http://example.net:9000/foo.html - port**
3. **http://www.turkey.example.net/foo.html sub-domain**

Aşağıdaki örnek ile farklı domainler üzerinden kaynak paylaşımı yapmak için gerekli olan konfigürasyon adımlarından bahsedilecektir. NewsApi uygulaması üzerinden örneklere yer verilecektir.

CORS WebConfig Ayarı

Aşağıdaki konfigürasyon ayarını web sunucunuzun Web.config xml dosyası içerisinde WebServer node içine yazmanız gerekmektedir.

WebConfig.cs

```
<system.webServer>
```

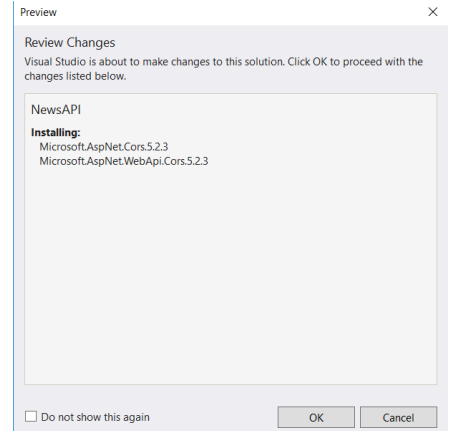
```
<httpProtocol>
  <customHeaders>
    <add name="Access-Control-Allow-Origin" value="*" />
    <add name="Access-Control-Allow-Methods" value="OPTIONS,GET,POST,PUT,DELETE,HEAD" />
    <add name="Access-Control-Allow-Headers" value="Content-Type,Accept" />
  </customHeaders>
</httpProtocol>
</system.webServer>
```

Yukarıdaki gibi WebConfig dosyasında yapacağımız bu ayar ile uygulama bazlı kaynak paylaşımı yapmış oluruz. **Access-Control-Allow-Origin** anahtar değerine karşılık gelen * tüm domainler'e kaynak paylaşımı izni verdiğimizi gösteriyor. **Access-Control-Allow-Methods** ile api üzerinden izin verilen tüm HttpVerbler global olarak belirtilmiştir. **Access-Control-Allow-Headers** anahtar değeri ile de web sunucusuna yapılan isteklerde header da izin verilen değerler aralarına virgül (,) konularak yazılır. Web Api bir kullanıcının hangi niyetle kaynak talep ettiğini anlamak için talebin(HttpRequest) Header bilgilerini kontrol eder. Aşağıda Headers olarak tanımlayabileceğimiz değerler ve bunların ne anlam ifade ettiklerinden bahsedilmiştir.

- **Content-Type:** API Burada belirtilen formatta veri sunar.
- **Accept:** Kullanıcı burada kabul ettiği veri türlerini belirtir. (application/json gibi)
- **Accept-Charset:** Kabul edilen karakter kodlaması. (UTF-8 gibi)
- **Accept-Language:** Kabul edilen dil seçeneği. (en-us, tr-tr gibi)
- **Accept-Encoding:** Kabul edilen içerik kodlaması. (gzip gibi)

Tabi yukarıdaki gibi Webconfig dosyası üzerinde global bir ayar yapmak istemeyebilirsiniz. Bazı methodlarınızı bazı domainlere açmak için özelleştirmek isteyebilirsiniz. Bu durumda Asp.Net Web Api Cors Kütüphanesi ile çalışabilir. Ve daha özelleştirilmiş konfigürasyonları kod bazlı yapabilirsiniz. Bu sayede Global.asax da yazacağınız kodlar ile uygulama genelinde bir CORS yapılandırması yaparken aynı zamanda actionlarınız için özel CORS istekleri de yapılandırabilirsiniz.

Install-Package Microsoft.AspNet.WebApi.Cors Nuget Package üzerinden ilgili paketi projemize indirip yapılandırma ayarlarını yapabiliriz.



Global.asax

```
protected void Application_Start()
{
    GlobalConfiguration.Configure(WebApiConfig.Register);

    //ilk parameter izin verilen domainler
    //ikinci parameter izin verilen methodlar
    //son parameter izin verilen headers bilgisi

    var cors = new EnableCorsAttribute("*", "*", "*");
    GlobalConfiguration.Configuration.EnableCors(cors);
}
```

Eğer yukarıdaki gibi Asp.Net Web Api Cors kütüphanesini yüklerseniz WebConfig dosyasında herhangi bir yapılandırma yapmadan, uygulama genelinde CORS ayarlarını Global.asax içerisine yukarıdaki kodlar ile yapabilirsiniz. Eğer sadece ilgili method için özel bir yapılandırma yapmak istiyorsanız bu durumda aşağıdaki gibi kullanmalısınız.

Global.asax

```
[ModelValid][EnableCors("*", "*", "*")]
public IHttpActionResult PostNews(NewsModel model)
{
    //Kodlar

    return Json(model);
}
```

Attribute Based Routing

Asp.Net Web Api 2.0 ile gelenek yeniliklerden bir diğeri de actionlarımız üzerinde tanımlayacağımızı [Route] attribute ile WebApiConfig dosyasının haricinde de kaynak yönlendirme kuralları tanımlayabilmemizdir. Kullanım sebebine gelince default route üzerinen verilerimizi dışarı açmak düzensiz ve dökümente edilmemiş gelişi güzel çıkış noktalarına sahip olduğumuzun göstergesidir. Bu sebeple default route dışında kaynaklarımıza erişmek için daha anlamlı seo bazlı yönlendirmeler belirlenebilir. Aşağıda birkaç örnek kullanım mevcuttur.

NewsapiController.cs

```
[Route("haberler")]
public IHttpActionResult GetNews()
{
    context.Configuration.ProxyCreationEnabled = false;
    var model = context.News.ToList().Select(a => new NewsModel
    {
        Name = a.Title,
        Content = a.ShortContent,
        CID = a.CategoryID,
        FullContent = a.Body
    }).ToList();
    return Ok(model);
}
```

NewsapiController.cs

```
namespace NewsAPI.Controllers
{
    [RoutePrefix("haberapi")]
    public class NewsApiController : ApiController
    {
    }
}
```

[RoutePrefix("haberapi")] = NewsApiController 'a gelen isteklerin başına haberapi ön takısı eklenmesini isteyecektir.Yani yönlendirmelerimizin ön takısı olarak kullanabiliriz.

[Route("haberler")] = Attribute kullanımı ile haberler veri kaynağının yönlendirmesi haberapi/haberler şeklinde olacağını söylemiş olduk. Şimdi postman kullanarak haberler veri kaynağımıza yeni route yapılandırmamız ile istek de bulunalım.

GET

http://localhost:16371/haberapi/haberler

Params

Send

AuthorizationHeaders (2)BodyPre-request ScriptTests

TypeNo Auth

BodyCookiesHeaders (13)TestsStatus: 200 OK

PrettyRawPreviewJSON

```
1 [
2   {
3     "NewsID": 0,
4     "Name": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum dolor nulla, fermentum fringilla nisl id,
5     "Content": "Praesent tempor nulla vitae sodales rhoncus. Phasellus felis orci, posuere a euismod ut, tempor vitae nisi.
6     "FullContent": "Praesent tempor nulla vitae sodales rhoncus. Phasellus felis orci, posuere a euismod ut, tempor vitae n
7     "CID": 1
8   },
9   {
10    "NewsID": 0,
11    "Name": "Başlık",
12    "Content": "Test",
13    "FullContent": "Test2",
14    "CID": 1
```

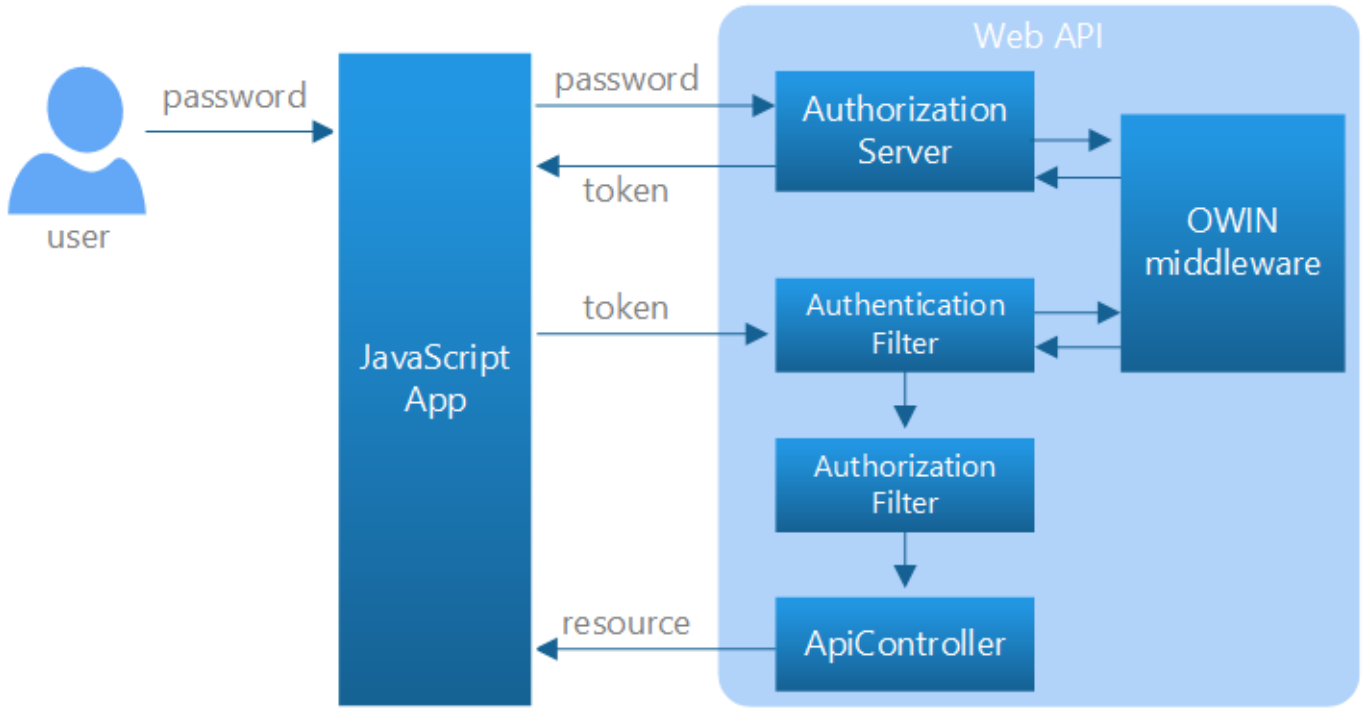
NewsApiController 'a gelen istek sonucunda aynı cevabın döndüğünü gözlemlemiş olduk.

Webapi Sık Kullanılan Attributelar

[RoutePrefix]	[Route]
[HttpVerbs]	[HttpGet]
[ActionName]	[HttpPost]
[Route]	[HttpDelete]
[Authorize]	[Authorize(Roles="")]
[FromUri]	[FromBody]
[EnableCors('*','*','*')]	[Queryable]
[HttpHead]	[HttpOptions]

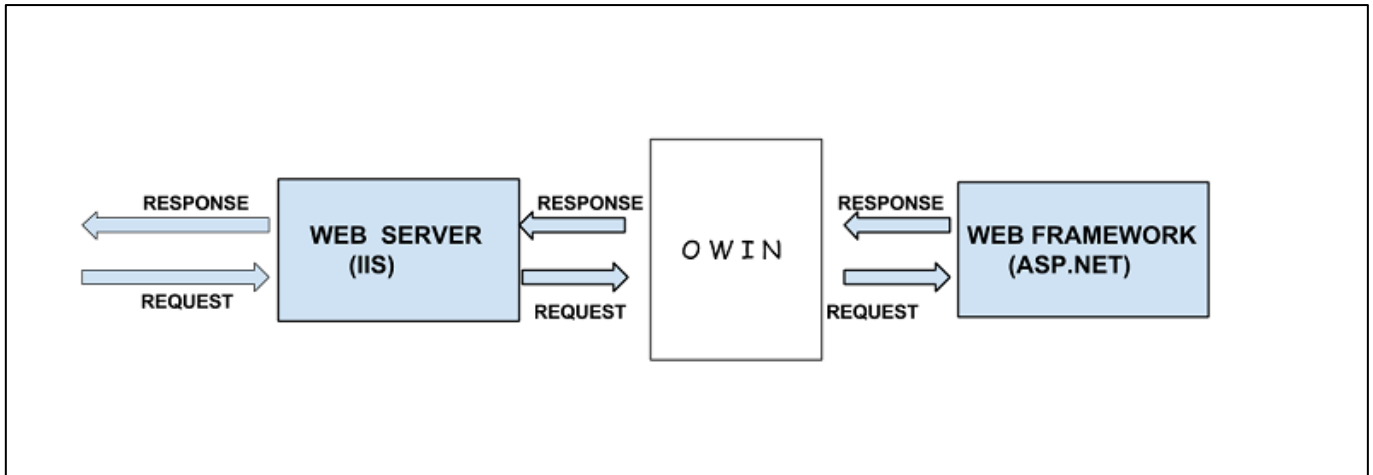
Web Api Authentication

Bu konumuzda api gibi dağıtık sistemlerde kimlik doğrulama ve kaynağa erişim gibi kavramlar üzerinde durulacaktır. Kimlik doğrulama ve yetkilendirme süreçlerinin yönetimi için ise OWIN adı verilen microsoft tarafından geliştirilmiş bir middleware kullanılacaktır. Dilerseniz öncelikle OWIN ne olduğu ile ilgili kısa bir bilgiden sonra kaynak erişiminin kısıtlanması ve kimlik doğrulama ile kaynak erişimin sağlanması gibi konulara değinilecektir.



OWIN Nedir

Öncelikle OWIN kelimesinin Open Web Interface for .Net (.Net için açık web arayüzü) olduğuyla söze başlayalım. Bir teknoloji olmaktan ziyade bir standart olan OWIN, .Net web sunucularıyla, yine .Net ile geliştirilmiş web uygulamalarının birbirleriyle nasıl iletişim kuracağını anlatmakta kullanılan bir arayüzü tanımlamaktadır. Bu tanımlamadaki amaç ise sunucu ve uygulamaları birbirlerinden net çizgilerle ayırmak ve uygulamaları sunuculardan bağımsız hale getirmektir.



OAuth Nedir

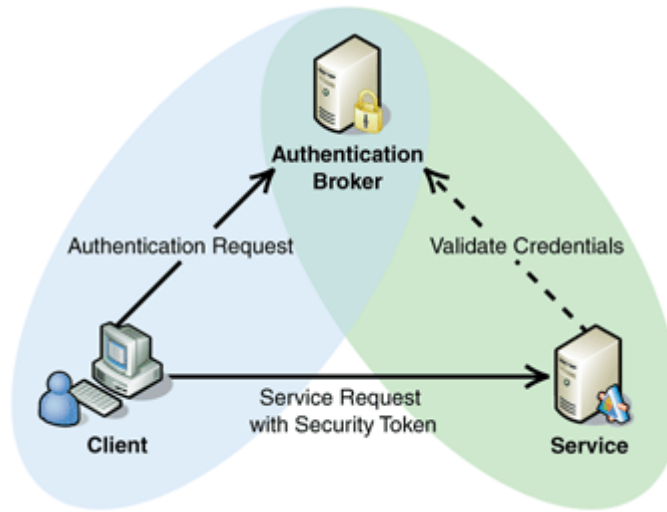
En kısa tanımla OAuth, kullanıcıların üyesi oldukları bir site yada platformun şifresini üye oldukları başka bir web sitesi yada platformla paylaşmadan, izin verdiği bilgilere diğer site tarafından ulaşılmasını sağlayan bir kimlik doğrulama protokolüdür.



Claim Based Authentication

Claim bir kullanıcıya ait bilgileri key value tipinde tutmamıza olanak sağlayan bir yapıdır. Claim Based Authentication yapısı sayesinde 3rd parti programlarda kimlik doğrulaması yaparak, hesabı doğrulanan kullanıcının bilgilerini talep edebiliriz.

Örneğin windows tarafında kimlik doğrulama yapan bir kullanıcının claim based authentication ile oturum açınca talep edilmesi gereken bilgileri varsayılanda groupid, sid, username olabilirken, Instagram üzerinden kimlik doğrulama yapılan kullanıcıların bilgileri, Doğum Tarihi, KullanıcıAdı, Sosyal medya hesapları gibi değişken bilgiler olabilir. Bu gibi durumlar için bize bu esnekliği sağlayan bir yapısı mevcuttur. Bu sayede kullanıcının sisteme giriş yaptığı takdir de hangi bilgilerinin kullanılacağını belileryerek bu claimlere daha sonradan ulaşabiliriz. Aşağıda çalışma prensibini anlatan bir görsel bulunmaktadır.



Token Based Authentication

Token; bilgisayar kavramı içerisinde yer alan, daha çok veri haberleşme dalında güvenlik açısından kullanılan bir terimdir. **Access token** ise kaynağa erişim kontrol işlemleri konusunu temsil eden bir sistem nesnesidir.

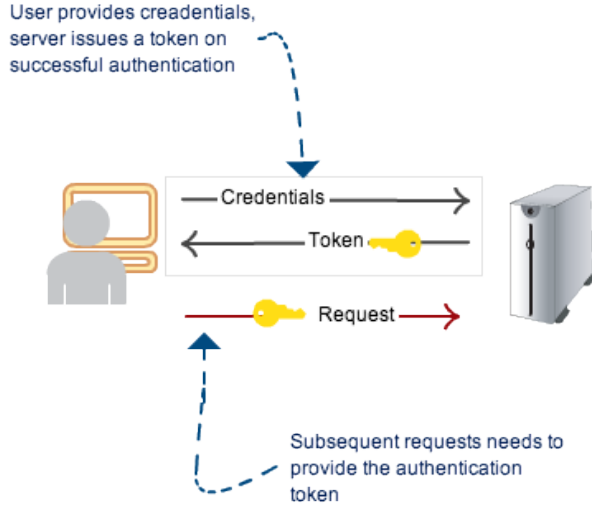
Token based Authentication kavramı ise sunucuda bulunan bir kaynağa yetkisiz erişimi engellemek amacı ile sunucu tarafından belli bir süreliğine oluşturulmuş, hashlenmiş bir kod olarak authentication sürecinden sonra istemciye gönderilen ve kullanıcının sunucudaki tüm kaynak erişimlerine bu erişim bileti (access-token) ile erişebilmesini sağlayan dağıtık mimarilerde tercih edilen bir güvenlik yöntemidir.

Access_Token sayesinde kullanıcının access_token üzerinden hangi claimlerine ulaşılması gerektiği sunucuya tanıtılarak, her bir access_token üzerinden kaynak erişimlerinde, kimlik doğrulayan kullanıcının bilgileri üzerinden kaynak yönetimi yapmamızı sağlar. Kullanıcı sunucudan yeni bir erişim bileti talep edene kadar, kaynak erişimleri bu access_token ile yürütülecektir.

Token Based Authentication Çalışma Prensibi

1. Kullanıcı; kullanıcı adı ve şifre bilgilerini sunucuda tanımlanmış bir token endpoint'e gönderir. Eğer kullanıcı hesabı sistemde tanımlanan bir kullanıcı ise WebApi kullanıcı bilgilerine göre kullanıcının talep edilecek olan claimslerini oluşturur. Tüm yetkilendirme süreçleri bu claimsler üzerinden kontrol edilecektir. Daha sonra claimsler ClaimsIdentity olarak sunucuya tanıtılarak oturum açmış olan kullanıcı için api tarafından kaynak erişim bileti access_token gönderilir.
2. Api ile haberleşen web-client tarafında access_token, cookie, local veya session storage gibi client traflı veri saklama yöntemleri kullanılarak sunucudan gönderilen bilgiler doğrultusunda yaşam süresi kadar oluşturulur. Kullanıcı sunucu ile olan tüm bu iletişimi access_token ile gerçekleştirir.
3. Eğer istemci oturumunu kapatmak istiyor ise access_token client tarafında silinir ve kullanıcı tekrar Api den access_token almak için login sayfasına yönlendirilir.
4. Eğer http protokolü üzerinden web sunucusuna gelen istek de access_token yok ise sunucu istemciye Unauthorized 401 HttpStatusCode ile dönüş yapar. Tabi burada dikkat edilmesi gereken nokta kaynağa erişilmesi gereken endpointlerin üzerinde [Authorize] attribute kullanılmalıdır.

5. Bunun dışında Web Api tarafında Token Based Authentication yapısını AuthenticationMode.Bearer ile yapmaktayız.



Token Based Authentication Uygulama

Senaryo için ihtiyacımız olan gereksinimlerimiz;

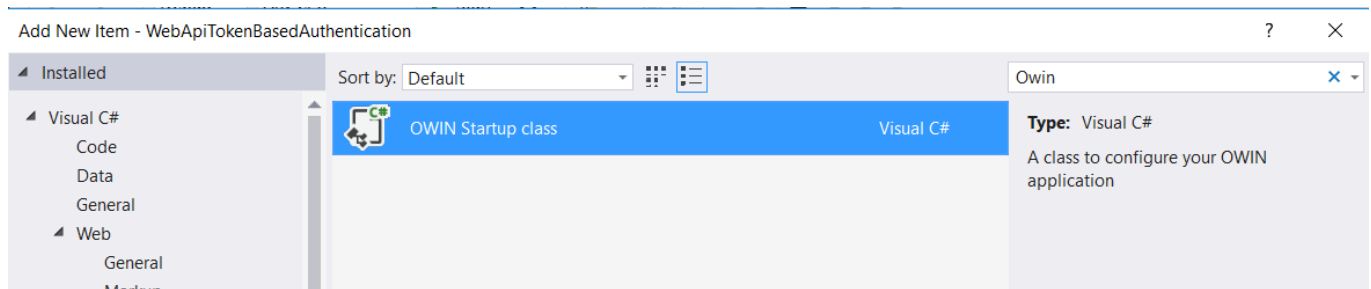
- Web Server (Asp.Net Web Api)
- Web Client (MVC veya SPA)

Web Api Konfigürasyonu için gerekli olan kütüphanelerimiz

- Microsoft.Owin
- Microsoft.Owin.System.Web
- Microsoft.Owin.Security
- Microsoft.Owin.Security.OAuth
- Microsoft.AspNet.WebApi.Cors

Owin Entegrasyonu

Bu kısımda Owin middleware kullanarak StartUp dosyası oluşturma ve Global olarak uygulama bazlı konfigüre etmek ile ilgili ayarlarımızı yapacağız. Öncelikle OwinStartup sınıfı oluşturmak için aşağıdaki adımları takip edelim. Projemiz Sağ Click -> Add New Item -> Arama kutusuna OWIN yazarak OWINStartUp class projemize ekleriz.



Daha sonra StartUp sınıfımız içerisinde aşağıdaki kodları yazarak Owin middleware ile uygulamamızın başlatılmasını sağlıyoruz.

Startup.cs

```
public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        //IApplicationBuilder uygulama oluşturuvcu arayüz. OWIN Assembly'sinden gelir.
        HttpConfiguration = new HttpConfiguration();
        //ConfigureOAuth(); method ile uygulamamızda Open Authentication protokolü ile kimlik
        doğrulama yapacağımızı söyledik. Microsoft.Owin.Security.OAuth Assembly ile projemize refereans
        vermiştik.
        ConfigureOAuth(app);
        //UseWebApi ile global olarak yapılan konfigürasyonların tanımlanmasını belirtmiş
        olduk.
        WebApiConfig.Register(httpConfiguration);
        app.UseWebApi(httpConfiguration);
    }
}
```

Startup.cs

```
public class Startup
{
    private void ConfigureOAuth(IApplicationBuilder appBuilder)
    {
        OAuthAuthorizationServerOptions = new OAuthAuthorizationServerOptions()
        {
            TokenEndpointPath = new Microsoft.Owin.PathString("/token"), // token
            alacağımız path'i belirtiyoruz
            AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
            AllowInsecureHttp = true,
            Provider = new CustomAuthorizationServerProvider()
        };

        //AppBuilder'a token üretimini gerçekleştirebilmek için ilgili authorization
        ayarlarımızı veriyoruz.
        appBuilder.UseOAuthAuthorizationServer(oAuthAuthorizationServerOptions);

        //Authentication type olarak ise Bearer Authentication'ı kullanacağımızı
        belirtiyoruz.
        //Bearer token OAuth 2.0 ile gelen standartlaşmış token türüdür.
        //Herhangi kriptolu bir veriye ihtiyaç duymadan client tarafından token isteğinde
        bulunulur ve server belirli bir expire date'e sahip bir access_token üretir.
        //Bearer token üzerinde güvenlik SSL'e dayanır.
        //Bir diğer tip ise MAC token'dır. OAuth 1.0 versiyonunda kullanılıyor, hem
        client'a, hemde server tarafına implementasyonlardan dolayı ek maliyet çıkartmaktadır. Bu
        maliyetin yanı sıra ise Bearer token'a göre kaynak alış verişinin biraz daha güvenli olduğu
        söyleniyor çünkü client her request'inde veriyi hmac ile imzalayıp verileri kriptolu bir şekilde
        göndermeleri gerektiği için.
        appBuilder.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());
    }
}
```

Cors Konfigürasyonu

Bu Web Client ile Web Server farklı domainler üzerinden olacağından, ve birbirleri arasında kaynak paylaşımı yapacaklarından, CORS konfigürasyonu yapmamız gerekmektedir. Aşağıda Global.asax da uygulama bazlı Cors ayarlarına yer verilmiştir.

Global.asax

```
protected void Application_Start()
{
    GlobalConfiguration.Configure(WebApiConfig.Register);

    //ilk parameter izin verilen domainler
    //ikinci parameter izin verilen methodlar
    //son parameter izin verilen headers bilgisi

    var cors = new EnableCorsAttribute("*", "*", "*");
    GlobalConfiguration.Configuration.EnableCors(cors);
}
```

Authorization Server Konfigürasyonu

Bu kısımda StartUp dosyamızda tanımladığımız ayarları uygulayacak olan ve kaynak dağıtımını yapabilmek için erişim bileti Access_Token üreteceğimiz ve aynı zamanda kaynağa erişmek isteyen kullanıcımızın claimlerini oluşturduğumuz Authorization Server konfigürasyonu yapacağız.

Startup.cs

```
public class CustomAuthorizationServerProvider: OAuthAuthorizationServerProvider
{
    public override async Task
    ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
    {
        context.Validated();
    }

    public override async Task
    GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
    {
        context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin", new[] { "*"
    });

    // Kullanıcının access_token alabilmesi için gerekli validation işlemlerini
    yapıyoruz.
    if (context.UserName == "User" && context.Password == "123")
    {
        // Oturum Açan kullanıcının talep edildiğinde erişilmesi istenen claimlerini
        oluşturuyoruz. Kullanıcıya name ve role özellikleri verdik. Authenticated olan kullanıcıdan bu
        bilgileri talep edebiliriz.

        var identity = new ClaimsIdentity(context.Options.AuthenticationType);

        identity.AddClaim(new Claim("name", context.UserName));
        identity.AddClaim(new Claim("role", "user"));

        context.Validated(identity);
    }
    else
    {
        // Eğer kullanıcı validasyondan geçemezse bu durumda erişim yetkisi olmadığına dair
        invalid_grant ile 401 Unauthorized HttpStatusCode Response Body olarak gönderilir.
        context.SetError("invalid_grant", "Kullanıcı adı veya şifre yanlış.");
    }
}
}
```

Web Api Resource Konfigürasyonu

Aşağıda kimlik doğrulaması yapıldıktan sonra erişeceğimiz olan kaynak ile ilgili ayarları yapılandırdığımız apicontroller ait kodlar bulunmaktadır. Dikkat ederseniz kaynağı kimlik doğrulaması yapmış bir kullanıcıya açmak için `[Authorize]` attribute kullanıyoruz.

Startup.cs

```
public class TestController : ApiController
{
    [Authorize]
    public IHttpActionResult Get()
    {
        return Json("OK");
    }
}
```

Access_Token LocalStorage Barındırma

Bu kısımda SPA uygulaması ile Api deki kaynaklara ulaşmak için gerekli olan endpoint den Access_Token isteğinde bulunuruz.

login.script.js

```
$("#btnlogin").click(function () {
    var User = new Object();
    User.grant_type = "password"
    User.UserName = "User";
    User.Password = "123";
    $.ajax({
        url: "http://localhost:61376/token",
        type: "POST",
        dataType: "json",
        data: User,
        success: function (response) {
            localStorage.setItem(response.access_token);
        }
    })
})
```

HttpHeaders da Access_Token Konfigürasyonu

Http isteği gönderilmeden önce isteğin headers bilgisinde Authorization: 'Bearer' ile access_token sunucuya taşınarak kullanıcının kaynak üzerinde erişim yetkisi olup olmadığı tespit edilir. Eğer yetki var ise HttpResponseMessage dönecektir. Eğer yetki yok ise 401 hatası ile kullanıcı yetkisi yok sayfasına yönlendirilebilir.

getInfo.script.js

```
$("#getInfo").click(function () {
    var access_token = localStorage.getItem("accessToken")
    $.ajax({
        url: "http://localhost:61376/api/test/get",
        beforeSend: function (xhr) { xhr.setRequestHeader('Authorization', 'Bearer ' + access_token); },
        type: "GET",
        dataType: "json",
        success: function (response) {
            console.log(response);
        },
        error: function (error) {
            console.log(error);
        }
    })
})
```

```
    },
  })
})
```

WEB API ODATA

OData ile Http istekleri üzerinden veri kaynaklarını sorgulayabilir, filtreleyebilir, select işlemleri yapabilir ve birden fazla model ile entegre çalışıp UI tarafta kullanabileceğimiz modelleri kendimize response olarak döndürmek için veri genişletme (expand) işlemleri yapabiliriz. Daha kısa anlatmak gerekirse OData açık olan bir veri kaynağı üzerinde belirli parametreler kullanılarak veriyi düzenleme işlemlerinde kullanılan bir tekniktir.

Istemci bu parametreleri querystring olarak kaynak Uri ye gönderir, server tarafında bu gönderilen parametreler sayesinde belli sıralama, filtreleme, sayfalama, limitleme gibi logicler'e göre sonuç client tarafına response olarak gönderilir. Yani anlayacağınız Sql ile ilgili sorgulama işlemlerini url üzerinden handle edebilmemizi sağlayan ve bize hızlı sonuçlar döndüren bir açık veri protokolüdür. OData ile çalışmak için Nuget paketlerinden Microsoft.AspNet.OData isimli Assembly'den yararlanmaktayız. Aşağıdaki tabloda ODATA Query Seçenekleri ve açıklamalarına yer verilmiştir.

Seçenek	Açıklama
\$expand	İlişkili entitileri tek bir json response olarak döndürmek için kullanılan bir yöntemdir. Eğer bir entity'nin ilişkili olduğu başka bir entity var ise ana entity'e expand edilerek veri çıkışı sağlanabilir.
\$filter	Veri kaynağı üzerinde belli bir logic üzerinden filtreleme işlemi yapar
\$orderby	Veri kaynağı üzerinde cevabı büyükten küçüğe veya küçüktten büyüğe sıralamamızı sağlar
\$select	Cevap ile ilişkili olan özelliklerin seçimi izin tercih edilir.
\$skip	Belirtilen sayıda kaydı atlamak için tercih edilir.
\$top	Belirli limitteki veriyi seçmek için kullanılır

OData Global olarak Register Etme

Uygulama genelinde ODATA kullanmak için aşağıdaki konfigürasyon ayarını Global.asax dosyası içerisinde gösteririz.

Global.asax

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(WebApiConfig.Register);

        //uygulama genelinde bu ayarı kullanabiliriz
        GlobalConfiguration.Configuration.EnableQuerySupport();
    }
}
```

Daha Sonra OData Kullanmak istediğiniz actionların Dönüş tipini **IQueryable** yapmalıyız. İlgili actionların üzerine ise **[Queryable]** attribute tanımlayarak artık querystring ile parametre bazlı sorgulama yapabiliriz. Aşağıda örnek bir kullanım mevcuttur.

ProductApiController.cs

```
public class ProductApiController : ApiController
{
    [Queryable]
    IQueryable<Product> Get() {

        ProjectContext db = new ProjectContext();

        var model = db.Products.AsQueryable();

        return model;
    }
}
```

Aşağıda products tablosuna querystring olarak atılacak olan sorgular ve sonuç olarak döndürecekleri veri tipleri ile ilgili örnek query isteklerine yer verilmiştir.

Seenek	Aıklama
Kategorisi Chai olan tüm ürünleri filtrelemek için kullanabiliriz	http://localhost/api/products/get?\$filter=CategoryName eq 'Chai'
Fiyatı 10 dan küçük olan tüm ürünleri sorgulamak için	http://localhost/api/products/get?\$filter=UnitPrice lt 10
Fiyatı 5 ile 10 arasında olan ürünleri sorgulamak için	http://localhost/api/products/get?\$filter=UnitPrice ge 5 and Price le 10
Ürün isminde aa geçen ürünler	http://localhost/api/products/get?\$filter=Substringof('aa',ProductName)
Ürünleri yüksek fiyattan düşük fiyata göre sıralamak için kullanılan sorgu	http://localhost/api/products/get?\$filter=\$orderby=UnitPrice desc
Ürünlerin fiyatlarını küçükten büyüğe doğru sıralamak için kullanılan sorgu	http://localhost/api/products/get?\$filter=\$orderby=UnitPrice
Ürünlerden sadece Fiyatı ve Ürün adı bilgisini seçmek için kullanılan sorgu	http://localhost/api/products/get?\$select=UnitPrice,ProductName