

**Media Type** یا **MIME Type** نشان دهنده فرمت یک مجموعه داده است. در HTTP، مدیا تایپ بیان کننده فرمت message body یک درخواست / پاسخ است و به دریافت کننده اعلام می‌کند که چطور باید پیام را بخواند. محل استاندارد تعیین Mime Type در هدر Content-Type است. درخواست کننده می‌تواند با استفاده از هدر Accept لیستی از MimeTypes های قابل قبول را به عنوان پاسخ، به سرور اعلام کند.

#### Response Headers

[view source](#)

```
Cache-Control public, max-age=2542200
Connection keep-alive
Content-Disposition attachment; filename=d79319c858e147f281eb2d0eebba7fc6.jpg
Content-Length 7387
Content-Type image/jpeg
Date Sat, 03 May 2014 16:52:31 GMT
Expires Mon, 02 Jun 2014 03:02:31 GMT
Last-Modified Sat, 03 May 2014 03:02:31 GMT
Server Microsoft-IIS/6.0
Vary *
X-Powered-By ASP.NET
```

#### Request Headers

[view source](#)

```
Accept image/png,image/*;q=0.8,*/*;q=0.5
Accept-Encoding gzip, deflate
Accept-Language en-US,en;q=0.5
Connection keep-alive
Cookie __utma=95334921.1626186387.1386794008.1399109809.1399127295.237; __utms=95334921
```

از [Asp.net Web API](#) MimeType برای تعیین نحوه serialize یا deserialize کردن محتوای دریافتی / ارسالی استفاده می‌کند.

Web API **MediaTypeFormatter** برای خواندن/درج پیام در بدنه درخواست/پاسخ از [MediaTypeFormmater](#) ها استفاده می‌کند. اینها کلاس‌هایی هستند که نحوه‌ی Serialize کردن و deserialize کردن اطلاعات به فرمت‌های خاص را تعیین می‌کنند. Web API به صورت توکار دارای formatter هایی برای نوع‌های XML ، JSON ، BSON و Form-UrlEncoded می‌باشد. همه این‌ها کلاس پایه MediaTypeFormatter را پیاده سازی می‌کنند.

#### مسئله

یک پروژه Web API بسازید و view model زیر را در آن تعریف کنید:

```
public class NewProduct
{
    [Required]
    public string Name { get; set; }

    public double Price { get; set; }

    public byte[] Pic { get; set; }
}
```

همانطور که می بینید یک فیلد از نوع byte[] برای تصویر محصول در نظر گرفته شده است. حالا یک کنترلر API ساخته و اکشنی برای دریافت اطلاعات محصول جدید از کاربر می نویسیم :

```
public class ProductsController : ApiController
{
    [HttpPost]
    public HttpResponseMessage PostProduct(NewProduct model)
    {
        if (ModelState.IsValid)
        {
            // ثبت محصول

            return new HttpResponseMessage(HttpStatusCode.Created);
        }

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }
}
```

و یک صفحه html به نام index.html که حاوی یک فرم برای ارسال اطلاعات باشد :

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1>ساخت MediaTypeFormatter برای Multipart/form-data</h1>
    <h2>محصول جدید</h2>

    <form id="newProduct" method="post" action="/api/products" enctype="multipart/form-data">
        <div>
            <label for="name">نام محصول</label>
            <input type="text" id="name" name="name" />
        </div>
        <div>
            <label for="price">قیمت</label>
            <input type="number" id="price" name="price" />
        </div>
        <div>
            <label for="pic">تصویر</label>
            <input type="file" id="pic" name="pic" />
        </div>
        <div>
            <button type="submit">ثبت</button>
        </div>
    </form>
</body>
</html>
```

زمانی که فرم حاوی فایلی برای آپلود باشد مشخصه enctype باید برابر با [Multipart/form-data](#) مقداردهی شود تا اطلاعات فایل به درستی کد شوند. در زمان ارسال فرم Content-type درخواست برابر با Multipart/form-data و فرمت اطلاعات درخواست ارسالی به شکل زیر خواهد بود :



برای ساختن یک MediaTypeFormatter یکی از 2 کلاس [MediaTypeFormatter](#) یا [BufferedMediaTypeFormatter](#) را باید پیاده سازی کنیم. تفاوت این دو در این است که BufferedMediaTypeFormatter برخلاف MediaTypeFormatter از متدهای synchronous استفاده می‌کند.

### پیاده سازی :

یک کلاس به نام MultiPartMediaTypeFormatter می‌سازیم و کلاس MediaTypeFormatter را به عنوان کلاس پایه آن قرار می‌دهیم.

```
public class MultiPartMediaTypeFormatter : MediaTypeFormatter
{
    ...
}
```

ابتدا در تابع سازنده کلاس فرمت‌هایی که می‌خواهیم توسط این کلاس خوانده شوند را تعریف می‌کنیم :

```
public MultiPartMediaTypeFormatter()
{
    SupportedMediaTypes.Add(new MediaTypeHeaderValue("multipart/form-data"));
}
```

در اینجا multipart/form-data را به عنوان تنها نوع مجاز تعریف کرده ایم.

سپس با پیاده سازی توابع CanWriteType و CanReadType مربوط به کلاس MediaTypeFormatter مشخص می‌کنیم که چه مدل‌هایی را می‌توان توسط این کلاس serialize / deserialize کرد. در اینجا چون می‌خواهیم این کلاس محدود به یک مدل خاص نباشد، از یک اینترفیس برای شناسایی کلاس‌های مجاز استفاده می‌کنیم.

```
public interface INeedMultiPartMediaTypeFormatter
{
}
```

و آنرا به کلاس newProduct اضافه می‌کنیم :

```
public class NewProduct : INeedMultiPartMediaTypeFormatter
{
    ...
}
```

از آنجا که تنها نیاز به خواندن اطلاعات داریم و قصد نوشتن نداریم، در متد CanWriteType مقدار false را برمی‌گردانیم.

```
public override bool CanReadType(Type type)
{
    return typeof(INeedMultiPartMediaTypeFormatter).IsAssignableFrom(type);
}

public override bool CanWriteType(Type type)
{
    return false;
}
```

و اما تابع ReadFromStreamAsync که کار خواندن محتوای ارسال شده و باید کردن آنها به پارامترها را برعهده دارد

```
public async override Task<object> ReadFromStreamAsync(Type type, Stream stream, HttpContent content,
    IFormatterLogger formatterLogger)
```

که در آن پارامتر type مربوط به مدل مشخص شده به عنوان پارامتر اکشن (NewProduct) است و پارامتر content محتوای

درخواست را در خود دارد.

ابتدا محتوای ارسال شده را خوانده و اطلاعات فرم را استخراج می‌کنیم و از طرف دیگر با استفاده از کلاس Activator یک نمونه از مدل جاری را ساخته و لیست property های آنرا استخراج می‌کنیم.

```
MultipartMemoryStreamProvider provider = await content.ReadAsMultipartAsync();
IEnumerable<HttpContent> formData = provider.Contents.AsEnumerable();

var modelInstance = Activator.CreateInstance(type);
IEnumerable<PropertyInfo> properties = type.GetProperties();
```

سپس در یک حلقه به ترتیب برای هر property متعلق به مدل، در میان اطلاعات فرم جستجو می‌کنیم. برای پیدا کردن اطلاعات متناظر با هر property در هدر Content-Disposition که در بالا توضیح داده شد، به دنبال فیلد همنام با property می‌گردیم.

```
foreach (PropertyInfo prop in properties)
{
    var propName = prop.Name.ToLower();
    var propType = prop.PropertyType;

    var data = formData.FirstOrDefault(d =>
        d.Headers.ContentDisposition.Name.ToLower().Contains(propName));
```

در صورتی که فیلدی وجود داشته باشد کار را ادامه می‌دهیم.

گفتیم که هر فیلد یک هدر، Content-Type هم می‌تواند داشته باشد. این هدر به صورت پیش فرض معادل text/plain است و برای فیلدهای عادی قرار داده نمی‌شود. در این مثال چون فقط یک فیلد غیر رشته ای داریم فرض را بر این گرفته ایم که در صورت وجود Content-Type، فیلد مربوط به تصویر است. در صورتیکه ContentType وجود داشته باشد، محتوای فیلد را به شکل Stream خوانده به [byte\[\]](#) تبدیل و با استفاده از متد SetValue در property مربوطه قرار می‌دهیم.

```
if (data != null)
{
    if (data.Headers.ContentType != null)
    {
        using (var fileStream = await data.ReadAsStreamAsync())
        {
            using (MemoryStream ms = new MemoryStream())
            {
                fileStream.CopyTo(ms);
                prop.SetValue(modelInstance, ms.ToArray());
            }
        }
    }
}
```

در صورتی که Content-Type غایب باشد بدین معنی است که محتوای فیلد از نوع رشته است ( عدد ، تاریخ ، guid ، رشته ) و باید به نوع مناسب تبدیل شود. ابتدا آن را به صورت یک رشته می‌خوانیم و با استفاده از Convert.ChangeType آنرا به نوع مناسب تبدیل می‌کنیم و در property متناظر قرار می‌دهیم.

```
if (data != null)
{
    if (data.Headers.ContentType != null)
    {
        //...
    }
    else
    {
        string rawVal = await data.ReadAsStringAsync();
        object val = Convert.ChangeType(rawVal, propType);

        prop.SetValue(modelInstance, val);
    }
}
```

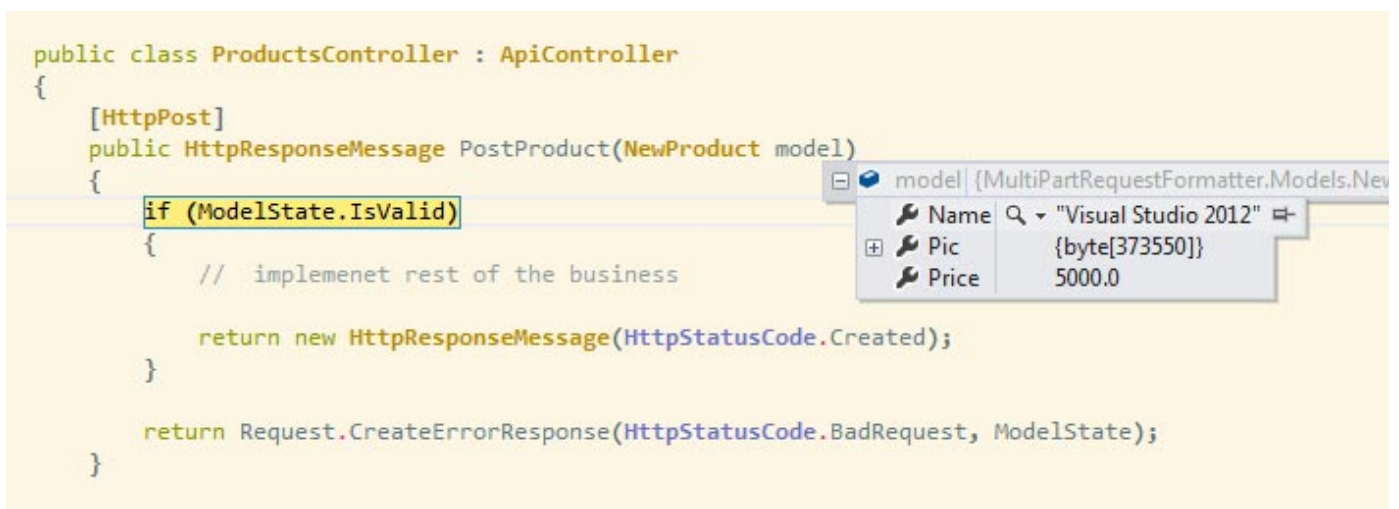
و در نهایت نمونه ساخته شده از مدل را برگشت می‌دهیم.

```
return modelInstance;
```

برای فعال کردن این Formatter باید آنرا به لیست formatters های web api اضافه بکنیم. فایل WebApiConfig در App\_Start را باز کرده و خط زیر را به آن اضافه می‌کنیم:

```
config.Formatters.Add(new MultiPartMediaTypeFormatter());
```

حال اگر مجدداً فرم را به سرور ارسال کنیم، با پیام خطایی، مواجه نشده و عمل binding با موفقیت انجام می‌گیرد.



The screenshot shows a C# code snippet for a `ProductsController` inheriting from `ApiController`. The `PostProduct` method is decorated with `[HttpPost]` and takes a `NewProduct` model. The code checks `ModelState.IsValid` and either returns a `HttpStatusCode.Created` response or a `BadRequest` response. A tooltip for the `model` parameter shows the following data contract:

Property	Value
Name	"Visual Studio 2012"
Pic	{byte[373550]}
Price	5000.0