Kapitel 2

ASP.NET Web API

In diesem Kapitel:

Einen einfachen REST-Service erstellen	76
Mehr Kontrolle über HTTP-Nachrichten	80
REST-Dienste über HttpClient konsumieren	83
Weiterführende Schritte mit der Web-API	86
Serialisierung beeinflussen	98
Web-API und HTML-Formulare	102
Fortschritt ermitteln	105
Feingranulare Konfiguration	106

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Egal ob auf mobilen Geräten, in Webbrowsern oder bei Cloud-Services, egal ob auf der Java-Plattform, unter PHP oder .NET: HTTP wird überall unterstützt. Dies ist auch der Grund dafür, warum die REST-Bewegung, die den puren Einsatz von HTTP zur Bereitstellung von Services fordert, in den letzten Jahren zunehmend an Bedeutung gewinnt.

Die ASP.NET Web API, welche im Rahmen von ASP.NET MVC 4 genutzt werden kann, wird in Zukunft die Entwicklung REST-basierter Dienste erheblich vereinfachen. Anders als WCF wird dabei das darunter liegende Protokoll nicht verborgen und abstrahiert, sondern ein direkter Zugriff auf die Möglichkeiten von HTTP gewährt.

Ursprünglich sollte die Web-API ein Teil von WCF werden. Allerdings entschied man sich bei Microsoft, dass sich ASP.NET MVC als übergeordnetes Framework besser eignet. Deswegen wurde sie von WCF Web API in ASP.NET Web API umbenannt.

Einen einfachen REST-Service erstellen

Möchte man REST-Dienste über einen Webserver zur Verfügung stellen, sind diese in einem ASP.NET MVC 4-Projekt anzulegen. Alternativ dazu kann sich der Entwickler auch, wie weiter unten beschrieben, selbst um das Hosting kümmern. In diesen Fällen wird ein eigener Webserver, der im Lieferumfang der Web-API enthalten ist, in einer benutzerdefinierten Anwendung gestartet.

REST-Dienste leiten von `ApiController` ab (siehe Listing 2.1). Die vom Entwickler bereitgestellten Methoden dieser Klassen stellt Web-API unter Beachtung bestimmter Konventionen via HTTP bereit. Beginnt der Name einer Methode zum Beispiel mit `Get`, kann sie über das HTTP-Verb `GET` erreicht werden. Dieses kommt per Definition immer dann zum Einsatz, wenn Daten abgerufen werden sollen, und es wird zum Beispiel immer dann von einem Webbrowser herangezogen, wenn der Benutzer angibt, zu einer bestimmten Adresse navigieren zu wollen. Dasselbe gilt analog für `POST`, `PUT` und `DELETE`. Hierbei steht `POST` für das Erzeugen von Ressourcen am Server, `PUT` für das Aktualisieren und `DELETE` für das Löschen.

Weicht der Name einer Methode von dieser Konvention ab, kann diese über die Attribute `HttpGet`, `HttpPost`, `HttpPut` und `HttpDelete` mit den entsprechenden Verben assoziiert werden (siehe `FindHotelsBySterne` in Listing 2.1). Weitere Verben können mit dem Attribut `AcceptVerbs` zugewiesen werden. Beispiele dafür finden sich in Listing 2.2, wo unter anderem das benutzerdefinierte Verb `X-ECHO`[1] der Methode `Echo` zugewiesen wird. Der Einsatz von `AcceptVerbs` verhindert jedoch nicht, dass die Web-API zusätzlich auch die erwähnten Konventionen anwendet. Aus diesem Grund könnte der Aufrufer die Methode `GetImplementationInfo` in Listing 2.2 sowohl über `X-INFO` als auch über `GET` erreichen. Damit `GET` nicht automatisch zugewiesen wird, wird hier jedoch unter Verwendung des Attributs `ActionName` angegeben, dass intern der Name `ImplementationInfo` anstatt von `GetImplementationInfo` zu verwenden ist.

[1] Benutzerdefinierte Verben sollten sparsam eingesetzt werden und beginnen per Definition mit `X-`

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Parameter und Rückgabewerte

Die Übergabeparameter dieser Methoden entnimmt die Web-API aus den übersandten URL-Parametern. Wird sie dort nicht fündig, versucht sie, die mittels HTTP in der Nutzlast (engl. payload) übertragenen Daten heranzuziehen. Im Zuge dessen werden auch Überladungen von Methoden unterstützt. Der von einer Methode zurückgelieferte Wert wird immer im Rahmen der Nutzlast zurückgeliefert.

Daten, die über die Nutzlast via HTTP zum Dienst oder zum Client übertragen werden, werden standardmäßig unter Verwendung von JSON dargestellt. Auf Wunsch des Clients kann die Darstellung auch via XML erfolgen. Daneben können binäre Daten ohne weitere Formatierung übersendet werden. Dazu ist der Typ `Stream` als Übergabeparameter und/oder Rückgabewert heranzuziehen. Weitere Formate unterstützt die Web-API, wie weiter unten beschrieben, unter Verwendung von benutzerdefinierten Formatierern. Um die Darstellungsart für die übermittelten Daten anzugeben, kann der Aufrufer den HTTP-Header `Content-Type` verwenden. Analog dazu kann er mittels `Accept` angeben, in welchem Format er den Rückgabewert erwartet.

```
public class HotelsController : ApiController
{
     public Hotel GetHotel(int id)
    {
        var rep = new HotelRepository();
        var hotel = rep.FindById(id);
        if (hotel == null) throw new HttpResponseException(HttpStatusCode.NotFound);

        return hotel;
    }

    public Hotel PostHotel(Hotel hotel)
    {
        var rep = new HotelRepository();
        rep.Create(hotel);
        return hotel;
    }

    [HttpGet]
    public List<Hotel> FindHotelsBySterne(int minSterne)
    {
        var rep = new HotelRepository();
        var hotels = rep.FindBySterne(minSterne);
        return hotels;
    }

    […]
}
```

Listing 2.1 Einfacher, auf einer Web-API basierender Dienst

```
public class HotelsController : ApiController
{
   […]

   [AcceptVerbs("X-ECHO")]
    public List<Hotel> EchoHotels(List<Hotel> hotels)
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
    {
        return hotels;
    }

    [AcceptVerbs("X-INFO")]
    [ActionName("ImplementationInfo")]
    public string GetImplementationInfo()
    {
        […]
    }
}
```

Listing 2.2 Konventionen überschreiben

REST-Dienste konfigurieren

Für das Konfigurieren von auf einer Web-API basierenden REST-Diensten bietet sich die Datei *global.asax* an. Diese beinhaltet eine Klasse mit Methoden, welche ASP.NET beim Eintreten bestimmter Ereignisse aufruft. Beispielsweise ist hier eine Methode `Application_Start` vorgesehen, welche beim Hochfahren der Anwendung im Webserver zur Ausführung kommt. Bei Verwendung der Visual Studio-Vorlage für Web-API-Projekte wird hier die statische Methode `WebApiConfig.Register` aufgerufen, welche für sämtliche Einstellungen gedacht ist, die ASP.NET Web API betreffen. Diese Methode definiert Routen, welche URLs auf `ApiController` abbilden.

Standardmäßig wird, wie in Listing 2.3 gezeigt, eine Route mit dem Namen `DefaultAPI` eingerichtet. Diese legt fest, dass `ApiController` über den URL `api/{controller}/{id}` erreichbar sind, wobei `{controller}` ein Platzhalter für den Namen des `ApiController`-Objekts ist; mit `{id}` wird ein beliebiger Wert bezeichnet, der einem eventuell vorhandenen Methodenprameter `id` zugewiesen wird. Wird zum Beispiel

http://servername:port/api/Hotel/17

unter Verwendung von `GET` angefordert, führt dies bei Verwendung des Controllers aus Listing 2.1 dazu, dass die Web-API die Methode `GetHotel` aufruft und für den Parameter `id` den Wert 17 übergibt. Dabei fällt auf, dass in dem URL lediglich der Name `Hotel` und nicht `HotelController` vorkommt – die Endung `Controller` wird also weggelassen. Der URL

http://servername:port/api/Hotel?id=17

führt zur selben Action-Methode, wobei in diesem Fall die ID explizit als URL-Parameter angeführt wird. Da die Web-API auch Methodenüberladungen unterstützt, könnte der Aufrufer hingegen den URL

http://servername:port/api/Hotel?minSterne=3

heranziehen, um die Methode `FindHotelsBySterne` aus Listing 2.1 zur Ausführung zu bringen.

Die Methode `WebApiConfig.Register` bietet sich auch für die Konfiguration weiterer Web-API-bezogener Aspekte an. Über die statische Eigenschaft `GlobalConfiguration.Configuration` kann hierzu ein Objekt, welches die aktuelle Konfiguration repräsentiert, abgerufen werden.

Listing 2.4 nutzt dieses Objekt, um mit der Eigenschaft `IncludeErrorDetailPolicy` das Verhalten der Web-API beim Auftreten von Ausnahmen im Programmcode zu konfigurieren. Der festgelegte Wert `Always` definiert, dass Fehlermeldungen in solchen Fällen immer übertragen werden sollen. Dies kann zwar für die Diagnose von Fehlern nützlich sein, bietet jedoch potenziellen Angreifern auch eine Menge Informationen. Aus

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

diesem Grund ist es eine gute Idee, im Produktivbetrieb die Einstellungen `LocalOnly` oder `Never` heranzuziehen. Erstere legt fest, dass Fehlermeldungen nur dann übertragen werden, wenn sich der Aufrufer auf demselben Rechner wie der Service befindet, was im Zuge der Entwicklung in der Regel der Fall ist. `Never` gibt hingegen an, dass Fehlermeldungen nie übertragen werden.

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

Listing 2.3 Standard-Route

```
GlobalConfiguration
            .Configuration
            .IncludeErrorDetailPolicy = IncludeErrorDetailPolicy.Always;
```

Listing 2.4 Web-API-Konfiguration

REST-Dienste mit Fiddler testen

Zum Testen von REST-Diensten bieten sich Anwendungen an, die ein direktes Versenden und Empfangen von HTTP-Nachrichten erlauben. Eine hierfür sehr beliebte Implementierung stellt Fiddler dar (*http://fiddler2.com*). Neben dem direkten Versenden von Nachrichten kann damit auch die HTTP-basierte Kommunikation anderer Anwendungen, wie zum Beispiel des Internet Explorers, überwacht werden.

Abbildung 2.1 zeigt zum Beispiel eine in Fiddler formulierte HTTP-Anfrage, welche beim in Listing 2.1 dargestellten Dienst Hotels abruft. Dabei kommt das Verb `GET` zum Einsatz. Mit dem Kopfzeileneintrag `Accept` wird angegeben, dass JSON als Antwortformat erwartet wird. Die dazugehörige Antwort findet sich in Listing 2.5.

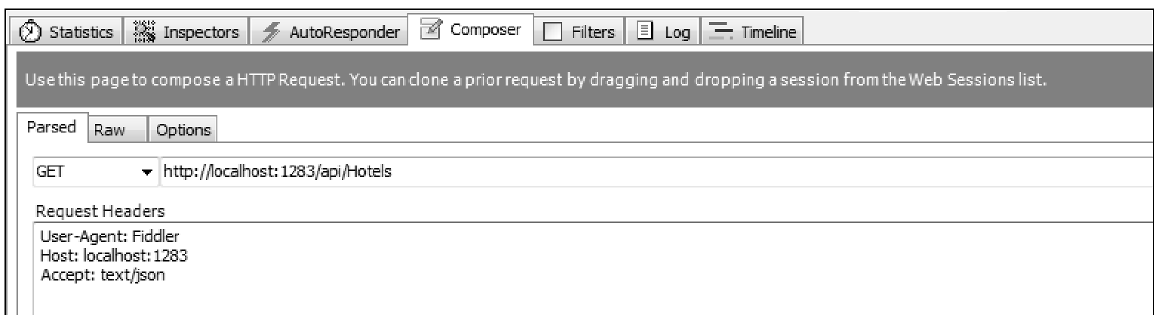

Abbildung 2.1 Daten mit Fiddler von einem REST-Dienst abrufen

```
HTTP/1.1 200 OK
[…]
Content-Type: text/json; charset=utf-8
Connection: Close
Content-Length: 57

[{"Bezeichnung":"Hotel zur Post","HotelId":1,"Sterne":3}]
```

Listing 2.5 HTTP-basierte Antwort mit JSON

Alternativ dazu findet sich in Abbildung 2.2 eine auf `POST` basierende Anfrage, welche im Gegensatz zum zuvor betrachteten Beispiel auch Daten an den Service sendet. Der Kopfzeileneintrag `Content-Type` zeigt hier an, dass die übersendeten Daten, welche im Feld `Request Body` zu finden sind, in Form von JSON vorliegen. Mit `Accept` wird wieder angezeigt, dass die Antwort ebenfalls JSON-formatiert sein soll.

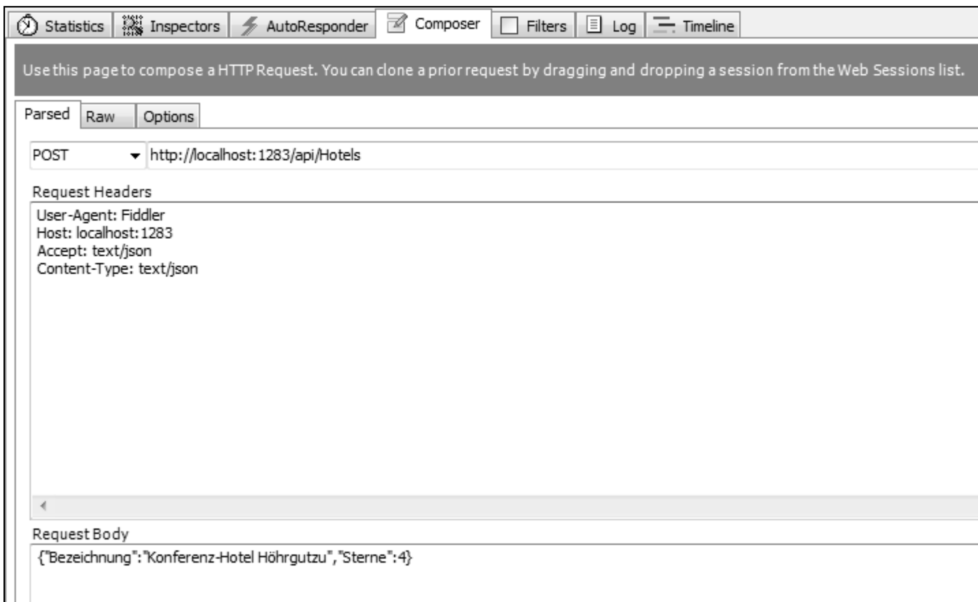

Abbildung 2.2 Daten mit Fiddler an REST-Dienst senden

Mehr Kontrolle über HTTP-Nachrichten

Um mehr Kontrolle über die von der Web-API via HTTP versendeten Nachrichten zu erhalten, liefert der Entwickler anstatt des eigentlichen Rückgabewerts eine Instanz von `HttpResponseMessage` zurück. Listing 2.6 demonstriert dies. Die darin enthaltene Methode `Post` nimmt eine `HotelBuchung` entgegen und speichert diese. Anschließend erzeugt es eine `HttpResponseMessage` und legt den HTTP-Statuscode auf den für diese Fälle vorgesehenen Wert `Created` (201) fest. Anschließend setzt sie den Kopfzeileneintrag `Location` auf jenen URL, unter dem die soeben angelegte Ressource ab sofort zu finden ist. Am Ende liefert sie die `HttpResponseMessage` zurück. In diesem Fall liefert die Web-API abgesehen von den definierten Kopfzeilen keine Daten zurück.

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
class BuchungenController: ApiController {
    public HttpResponseMessage Post(HotelBuchung buchung)
    {
        var rep = new HotelBuchungRepository();
        rep.Create(buchung);

        var response = new HttpResponseMessage(HttpStatusCode.Created);
        string uri = Url.Route(null, new { id = buchung.HotelBuchungId });
        response.Headers.Location = new Uri(Request.RequestUri, uri);

        return response;
    }
    […]
}
```

Listing 2.6 Antwort mit `HttpResponseMessage` beeinflussen

Während ASP.NET Web API beim Aufruf der soeben betrachteten Methode lediglich Kopfzeilen retour liefert, finden sich im Ergebnis der Methode in Listing 2.7 auch Nutzdaten wieder. Sie erzeugt unter Verwendung der Eigenschaft `Request`, die die aktuelle Anfrage repräsentiert, ein `HttpResponse`-Objekt als Antwort. Im Zuge dessen übergibt sie den gewünschten Status-Code sowie die im Rahmen der Nutzdaten zu übertragenden Informationen. Bei diesen handelt es sich um eine Liste mit Buchungen. Anschließend legt sie ausgewählte Kopfzeileneinträge für die Antwort fest und liefert diese zurück.

```
public HttpResponseMessage GetByHotel(int hotelId)
{
    var rep = new HotelBuchungRepository();
    var buchungen = rep.FindByHotel(hotelId);


    var response =
          Request.CreateResponse<List<HotelBuchung>>(
                              HttpStatusCode.OK, buchungen);

    response.Headers.CacheControl = new CacheControlHeaderValue();
    response.Headers.CacheControl.NoCache = true;

    return response;

}
```

Listing 2.7 Beeinflussung der zurückgelieferten Kopfzeileneinträge

Objekte der Klasse `HttpResponseMessage` verweisen über ihre `Content`-Eigenschaft auf die zurückzugebenden Nutzdaten. Diese Eigenschaft ist vom Typ `HttpContent`. Im zuletzt betrachteten Beispiel erzeugt `Request.CreateResponse` eine Instanz von `ObjectContent`, die die Liste mit den abgerufenen Buchungen ummantelt, und weist diese `Content` zu.

Bei `ObjectContent` handelt es sich um eine von mehreren Subklassen von `HttpContent`. Wie der Name vermuten lässt, besteht die Aufgabe dieser Klasse darin, die an den Aufrufer zu sendenden Objekte zu repräsentieren. Bei der direkten Instanziierung dieser Klasse muss der Entwickler einen so genannten `Formatter` angeben, der das Übertragungsformat bestimmt.

Bei der Erzeugung über Request.CreateResponse muss er hingegen keinen Formatter angeben. In diesem Fall wählt ASP.NET Web API einen passenden Formatter unter Berücksichtigung jenes Formats, welches der Aufrufer in Form eines eventuellen Accept-Headers angefordert hat. Beispielsweise kann über die Header der Anfrage durch Angabe von Accept: text/json JSON als Antwortformat erbeten werden. Standardmäßig unterstützt ASP.NET Web API XML und JSON.

Neben ObjectContent bietet ASP.NET MVC noch weitere HttpContent-Derivate: ByteArrayContent, StreamContent, StringContent, FormUrlEncodedContent, MultipartContent. Die ersten drei veranlassen den Download des Inhalts eines Byte-Arrays, eines Streams oder eines Strings; FormUrlEncodedContent liefert URL-kodierte Schlüssel/Wert-Paare zurück (z.B. Parameter1=Wert1&Parameter2=Wert2) und MultipartContent liefert mehrere Ergebnisse zurück, die allesamt durch jeweils ein weiteres HttpContent-Derivat repräsentiert werden. Eine Subklasse von MultipartContent ist MultipartFormDataContent, welche bei Dateiuploads über HTML-Formulare zum Einsatz kommt.

Während die Thematik rund um Dateiuploads und MultipartContent weiter unten besprochen wird, demonstriert das nachfolgende Codefragment den Einsatz der Eigenschaft Content unter Verwendung einer StringContent-Instanz.

```
var r = new HttpResponseMessage();
r.Content = new StringContent("Hallo Welt");
return r;
```

Analog zum Zugriff auf die Kopfzeilen der Antwortnachricht kann der Entwickler auch auf die Kopfzeilen der Anfragenachricht zugreifen. Diese ist vom Typ HttpRequestMessage und kann innerhalb eines ApiController-Objekts über dessen Eigenschaft Request erreicht werden. Alternativ dazu kann der Entwickler einen Parameter dieses Typs definieren. In diesem Fall injiziert ASP.NET Web API beim Aufruf der Action-Methode das Request-Objekt in diesen Parameter.

Das auf diese Weise erhaltene HttpRequestMessage-Objekt wird in Listing 2.8 verwendet, um den Kopfzeileneintrag If-None-Match in Erfahrung zu bringen. Dieser wird zur Implementierung eines bedingten GET (engl. conditional get) herangezogen. *Bedingtes GET* bedeutet, dass der Aufrufer eine Ressource nur dann haben möchte, wenn eine bestimmte Bedingung erfüllt ist, beispielsweise wenn die Ressource seit dem letzten Abruf geändert wurde. Im Fall von If-None-Match gibt der Aufrufer hierzu ein so genanntes Entity-Tag (ETag) an.

Ein Entity-Tag steht für den aktuellen Zustand einer Ressource und muss sich per Definition ändern, wenn jemand die Ressource ändert. Hierbei kann es sich zum Beispiel um einen Hashwert, den Zeitstempel der letzten Änderung oder um eine Versionsnummer handeln. Weist die angeforderte Ressource nach wie vor das angegebene Entity-Tag auf, muss sie nicht erneut über das Netzwerk übertragen werden. Diesen Umstand zeigt der Dienst durch den HTTP-Status Not Modified (304) an. Hat sich jedoch das Entity-Tag und somit auch die gewünschte Ressource in der Zwischenzeit geändert, liefert der Dienst diese zurück, wobei das aktuelle Entity-Tag innerhalb der Kopfzeilen positioniert wird.

```
private string Quote(string str)
{
    return "\"" + str + "\"";
}

public HttpResponseMessage Get(int id, HttpRequestMessage request)
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
{
    var rep = new HotelBuchungRepository();
    var buchung = rep.FindById(id);

    if (buchung == null)
    {
        return this.Request.CreateResponse(HttpStatusCode.NotFound);
    }

    var etag = Quote(buchung.Version.ToString());
    var ifNoneMatchHeader = request.Headers.IfNoneMatch.FirstOrDefault();
    if (ifNoneMatchHeader != null && ifNoneMatchHeader.Tag == etag)
    {
        return this.Request.CreateResponse(HttpStatusCode.NotModified);
    }

    var response = this.Request.CreateResponse<HotelBuchung>(
                                          HttpStatusCode.OK, buchung);

    response.StatusCode = HttpStatusCode.OK;
    response.Headers.CacheControl = new CacheControlHeaderValue();
    response.Headers.CacheControl.NoCache = true;
    response.Headers.ETag = new EntityTagHeaderValue(etag);
    return response;
}
```

Listing 2.8 Bedingtes GET

Dass die übermittelten Nutzdaten nicht zwangsläufig in typisierter Form gelesen werden müssen, veranschaulicht Listing 2.8. Die hier gezeigte Methode `Put` liest die Nutzdaten über einen Stream und gibt sie zu Demonstrationszwecken im Debugfenster aus. Da es sich bei der Methode `ReadAsStreamAsync` um eine Erweiterungsmethode handelt, muss der Namespace `System.Net.Http` eingebunden werden, damit sie zur Verfügung steht.

```
public void Put(int id)
{
    var stream = Request.Content.ReadAsStreamAsync().Result;

    using (var r = new StreamReader(stream)) {
        Debug.WriteLine(r.ReadToEnd());
    }
}
```

Listing 2.9 Übersendete Daten als Stream lesen

REST-Dienste über HttpClient konsumieren

Web-API-basierte Services können mit jedem Client, der HTTP unterstützt, konsumiert werden. Unter JavaScript bietet sich dazu der Einsatz von Bibliotheken, wie *jQuery*, an (siehe Kapitel 3). Für .NET-basierte Clients stellt die Web-API mit der Klasse `HttpClient` ein neues, vereinfachtes Programmiermodell für solche Szenarien zur Verfügung. Wie viele neue Programmiermodelle ist auch dieses asynchron und hilft somit, Anwendungen zu schreiben, die jederzeit auf Benutzereingaben reagieren. Aus diesem Grund kommen dabei die mit .NET 4.0 eingeführten Tasks immer wieder zum Einsatz.

Um sicherzustellen, dass sämtliche Bibliotheken, die für den Einsatz von `HttpClient` benötigt werden, referenziert sind, empfiehlt sich das Einbinden des NuGet-Packages *Microsoft.AspNet.WebApi.Client*. Im Zuge dessen wird auch die von ASP.NET Web API zur JSON-Serialisierung herangezogene freie Bibliothek *JSON.Net* eingebunden. Da sich `HttpClient` teilweise auf Erweiterungsmethoden stützt, sollte darüber hinaus der Namespace `System.Net.Http` mit einer entsprechenden `using`-Anweisung eingebunden werden.

Listing 2.10 zeigt einen einfachen Client, der unter Verwendung von `HttpClient` auf den in diesem Kapitel gezeigten Hoteldienst zugreift. Dazu fordert dieser mit `GetAsync` den gegebenen URL an. Der Rückgabewert dieser Methode ist ein Task, welcher eine asynchrone Operation repräsentiert. Mit `ContinueWith` gibt der Client einen Lambda-Ausdruck an, der auszuführen ist, wenn dieser Task beendet wurde. Als Parameter nimmt dieser Ausdruck den fertiggestellten Task entgegen und gibt mittels `Result.Content.ReadAsAsync` an, dass das Ergebnis geparst in ein `IEnumerable<Hotel>` umgewandelt werden soll. Dies resultiert in einem weiteren asynchronen Task, welcher wiederum mit `ContinueWith` erweitert wird. Der dazu verwendete Lambda-Ausdruck nimmt das `Result`, bei dem es sich nun um das gewünschte `IEnumerable<Hotel>` handelt, entgegen und gibt die darin enthaltenen Hotels aus.

Dabei ist zu beachten, dass der Entwickler die Klasse `Hotel` am Client selber erstellen muss. Kommt sowohl am Client als auch am Server .NET zum Einsatz, könnte er die Klasse auch kopieren oder diese in eine Assembly auslagern, die sowohl beim Client als auch beim Server eingebunden ist. Dieser Umstand ist der Tatsache geschuldet, dass es keinen akzeptierten Standard gibt, um REST-basierte Dienste formal zu beschreiben. Somit besteht, im Gegensatz zur SOAP-Welt, wo Webdienste durch WSDL-Dokumente beschrieben werden, leider auch nicht die Möglichkeit, die benötigten Klassen oder gar Proxys am Client zu generieren.

```
var client = new HttpClient();

var url = "http://localhost:1283/api/Hotels";

// Simple Get
client.GetAsync(url).ContinueWith(getTask =>
{
    getTask.Result
               .Content.ReadAsAsync<IEnumerable<Hotel>>()
               .ContinueWith(readTask =>
    {
        var hotels = readTask.Result;
        foreach (var hotel in hotels)
        {
            Console.WriteLine(hotel.Bezeichnung);
        }

    });
});
Console.WriteLine("Aktion wird im Hintergrund ausgeführt...");
Console.ReadLine();
```

Listing 2.10 Ressource mit `HttpClient` abrufen

Der Einsatz von solchen asynchronen APIs mag an dieser Stelle ein wenig kompliziert erscheinen. Die aufwändige Handhabung von Tasks gestaltet sich ab .NET 4.5 bei Verwendung asynchroner Methoden, die die neuen Schlüsselwörter `async` und `await` nutzen, um einiges einfacher. Listing 2.11 demonstriert dies, indem es die Logik von Listing 2.10 unter Verwendung dieser neuen Schlüsselwörter implementiert. Die gezeigte Methode wurde mit dem Schlüsselwort `async` definiert. Das bedeutet, dass sie beim ersten Auftreten von `await` im Hintergrund ausgeführt wird, um den aktuellen Thread nicht zu blockieren.

```
static async void SimpleGet()
{
    var client = new HttpClient();

    var url = "http://localhost:1307/api/Hotels";

    var response = await client.GetAsync(url);
    var hotels = await response.Content.ReadAsAsync<IEnumerable<Hotel>>();

    foreach (var hotel in hotels)
    {
        Console.WriteLine(hotel.Bezeichnung);
    }
}
```

Listing 2.11 `HttpClient` unter Verwendung von `async` und `await` nutzen

Um Kopfzeileneinträge an den Server zu senden, müsste der Entwickler Gebrauch von der Auflistung `DefaultRequestHeaders` des `HttpClients` machen. Die hier hinterlegten Parameter werden bei jedem nachfolgenden Aufruf in die Nachricht eingebunden. Der folgende Schnipsel demonstriert deren Einsatz.

```
client.DefaultRequestHeaders.AcceptLanguage.Add(
               new StringWithQualityHeaderValue("de-DE"));
```

Analog zu `GetAsync` stehen auch weitere Methoden für den Einsatz der Verben `POST`, `PUT` und `DELETE` zur Verfügung. Diese nennen sich `PostAsync`, `PutAsync` und `DeleteAsync`. Weitere Verben können mit `SendAsync` verwendet werden. Alternativ zur Methode `ReadAsAsync`, welche die Nutzlast in ein Objekt umwandelt, kann auch die Methode `ReadAsStringAsync`, welche die Nutzlast als Zeichenfolge zurückliefert, oder die Methode `ReadAsStreamAsync`, welche über einen Stream Zugriff auf die Nutzlast gewährt, herangezogen werden. Daneben kann mit `ReadAsByteArrayAsync` ein Byte-Array angefordert werden und `ReadAsMultipartAsync` erlaubt den Zugriff auf eine mehrteilige MIME-basierte Nachricht. Neben der Eigenschaft `Content`, die die Nutzlast repräsentiert, erhält der Entwickler über die Eigenschaft `Header` Zugriff auf Kopfzeileneinträge oder über `StatusCode` auf den vom Server gemeldeten HTTP-Status.

Ein Beispiel für den Einsatz von `SendAsync` findet sich in Listing 2.12. Dieses Beispiel macht deutlich, dass `SendAsync` mehr Kontrolle über die Anfrage und Antwort erlaubt, indem diese als `HttpRequestMessage` und `HttpResponseMessage` explizit dargestellt werden. Im betrachteten Fall wird ein `HotelBuchung`-Objekt an den Dienst gesendet. Im Zuge dessen werden die für die Antwort bevorzugten Sprachen über den Kopfzeileneintrag `Accept-Language` angegeben.

```
private async static void SendBuchung()
{
    var url = "http://localhost:1307/api/Buchungen";

    var buchung = new HotelBuchung
    {
        HotelId = 1,
        Vorname = "Max",
        Nachname = "Muster"
    };
```

```
    var client = new HttpClient();

    var request = new HttpRequestMessage();
    request.Content = new ObjectContent<HotelBuchung>(buchung,
                                       new JsonMediaTypeFormatter());
    request.Content.Headers.ContentType = new MediaTypeHeaderValue("text/json");


    request.Method = HttpMethod.Post;
    request.RequestUri = new Uri(url);

    request.Headers.AcceptLanguage.Add(
                new StringWithQualityHeaderValue("de-DE"));
    request.Headers.AcceptLanguage.Add(
                new StringWithQualityHeaderValue("de-AT"));

    var response = await client.SendAsync(request);

    Console.WriteLine("Status: " + response.StatusCode);
    Console.WriteLine("Location: " + response.Headers.Location);
}
```

Listing 2.12 Details einer Anfrage beeinflussen

Weiterführende Schritte mit der Web-API

Nachdem die letzten Abschnitte gezeigt haben, wie man einen einfachen REST-Dienst mit der ASP.NET Web API entwickelt, beschäftigt sich dieser Abschnitt mit weiterführenden Aspekten, wie dem Definieren benutzerdefinierter Routen, dem Hinterlegen von Querschnittsfunktionen, die von allen oder vielen Diensten benötigt werden oder der Implementierung benutzerdefinierter Formate jenseits von XML und JSON.

Benutzerdefinierte Routen

Web-API gestattet die Definition von benutzerdefinierten Routen. Diese sind unter Verwendung der Methode `MapHttpRoutes` in der Datei *App_Start/WebApiConfig.cs* innerhalb von `Register` zu definieren. In Listing 2.13 wird zum Beispiel festgelegt, dass die Methoden des in den vorangegangenen Abschnitten besprochenen `BuchungenControllers` über `api/Hotels/{hotelId}/Buchungen/{id}` erreichbar sind. Dies schließt die Verwendung der Standardroute, die in derselben Datei definiert wird, jedoch nicht aus.

Bei `{hotelId}` handelt es sich hierbei um einen Platzhalter für einen Wert, der einem gleichnamigen Übergabeparameter der jeweiligen Methode übergeben wird. `{id}` stellt ebenfalls einen solchen Platzhalter dar. Im Gegensatz zu `{hotelId}` ist dieser, wie im Parameter `defaults` angegeben, optional.

```
[…]
    routes.MapHttpRoute(
        name: "BuchungenByHotelRoute",
        routeTemplate: "api/Hotels/{hotelId}/Buchungen/{id}",
        defaults: new { controller = "Buchungen", id = RouteParameter.Optional }
    );
[…]
```

Listing 2.13 Benutzerdefinierte Route

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Dynamische Parameter

In Fällen, in denen der Entwickler keine eigene Klasse für das Objekt erstellen möchte, das eine Action-Methode zurückliefert, kann er auch ein anonymes Objekt erzeugen und dieses als `object` zurückgeben (Listing 2.14).

```
public object Get()
{
    return new
    {
        Version = 0.9
    };
}
```

Listing 2.14 Anonymes Objekt als Rückgabewert

Wird JSON als Übergabeformat verwendet, kann das übersendete JSON-Objekt auch generisch als `JObject` dargestellt werden (siehe Listing 2.15). Dieses bietet über Indexer Zugriff auf die einzelnen Eigenschaften. Das betrachtete Beispiel prüft, ob die Eigenschaft `All` die Zeichenkette `true` aufweist sowie in weiterer Folge, ob es eine Eigenschaft `Settings.IncludeEMail` mit dem Wert `true` gibt.

```
public object Post(JObject value)
{

    if (value["All"].ToString().ToLower() == "true")
    {

        string email = "";
        JToken includeEMail = null;
        JToken settings = value["Settings"];
        if (settings != null) includeEMail = settings["IncludeEMail"];

        if (includeEMail != null && includeEMail.ToString().ToLower() == "true")
        {
            email = "vorname.nachname@domain.siehe.oben";
        }
        else
        {
            email = "";
        }


        var response = new
        {
            Version = 0.9,
            Autor = "Manfred Steyer",
            URL = "www.softwarearchitekt.at",
            EMail = email
        };


        return response;

    }
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
    return new {
        Version = 0.9
    };


}
```

Listing 2.15 Generische Darstellung von JSON-Objekten mit JToken

Tracing

ASP.NET Web API verwendet einen anpassbaren Tracing-Mechanismus, um Nachrichten über durchgeführte Aktionen zu protokollieren. Um die protokollierten Informationen zu erhalten, implementiert der Entwickler das Interface `ITraceWriter` (Listing 2.16) sowie deren Methode `Trace`, an welche ASP.NET Web API die einzelnen Informationen übergibt. Im betrachteten Beispiel werden die erhaltenen Informationen lediglich mit `Debug.WriteLine` im Debugfenster ausgegeben. Alternativ dazu könnte der Entwickler an dieser Stelle die Daten in einer Protokolldatei oder in einer Datenbank ablegen bzw. an ein Protokollierungsframework der Wahl delegieren.

Interessant ist auch die Tatsache, dass zunächst nur die wichtigsten Daten der zu protokollierenden Nachricht zur Verfügung stehen. Dabei handelt es sich um die aktuelle `HttpRequestMessage`, eine Nachrichtenkategorie sowie ein Level, das den Schweregrad der Nachricht (`Debug`, `Info`, `Warn`, `Error`, `Fatal`) anzeigt. Aufgrund dieser Informationen kann die Implementierung entscheiden, ob weitere Details der Nachricht ermittelt werden sollen. Falls dem so ist, erzeugt sie – wie in der betrachteten Implementierung gezeigt – mit den genannten Daten einen `TraceRecord` und übergibt diesen an die übergebene `Action`. Diese hat die Aufgabe, den `TraceRecord` mit Details zu befüllen. Dadurch, dass diese Aufgabe von einer Action wahrgenommen wird, muss das Ermitteln von Details nur dann durchgeführt werden, wenn die Nachricht auch wirklich protokolliert werden soll. In allen anderen Fällen wird auf diese Aufgabe zugunsten der Performance verzichtet.

```
public class CustomTraceWriter : ITraceWriter
{
    public void Trace(HttpRequestMessage request, string category,
                      System.Web.Http.Tracing.TraceLevel level, Action<TraceRecord> traceAction)
    {
        TraceRecord record = new TraceRecord(request, category, level);
        traceAction(record);
        Debug.WriteLine(category + ": " + level + " " + record.Message);
    }
}
```

Listing 2.16 Benutzerdefinierter `TraceWriter`

Um den benutzerdefinierten `TraceWriter` zu registrieren, verwendet der Entwickler den nachfolgenden Schnipsel, welchen er zum Beispiel in der Methode `WebApiConfig.Register` platziert:

```
config.Services.Replace(typeof(ITraceWriter), new SimpleTracer());
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Der auf diese Weise registrierte `TraceWriter` kann auch von den einzelnen Action-Methoden verwendet werden. Der folgende Schnipsel zeigt, wie dies bewerkstelligt wird:

```
Configuration.Services.GetTraceWriter().Info(Request, "Kategorie123", "Hallo Welt!");
```

Querschnittsfunktionen mit Message-Handlern implementieren

Um zu verhindern, dass der Entwickler allgemeine Logiken, wie Sicherheits-Prüfungen oder Protokollierungen, in jeder Methode wiederholen muss, kann er diese in Subklassen von `DelegatingHandler` auslagern. Die gewünschte Logik ist dabei innerhalb der zu überschreibenden Methode `SendAsync` zu hinterlegen.

Ein Beispiel dafür findet sich in Listing 2.17. Der hier gezeigte `LoggingHandler` gibt Informationen über den aktuellen Methodenaufruf im Debugfenster aus. Anschließend wird `base.SendAsync` aufgerufen. Diese Methode veranlasst die Web-API, den nächsten konfigurierten `DelegatingHandler` zur Ausführung zu bringen. Existiert kein weiterer `DelegatingHandler`, stößt `base.SendAsync` die eigentliche Dienstmethode an. Somit sind Aktionen, die vor dem Ausführen der Dienstmethode stattfinden sollen, vor diesem Aufruf zu platzieren und jene, die die Web-API erst danach zur Ausführung bringen soll, danach.

```
public class LoggingHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,

System.Threading.CancellationToken cancellationToken)
    {
        Debug.WriteLine("Begin Request: {0} {1}", request.Method, request.RequestUri);
        return base.SendAsync(request, cancellationToken);
    }
}
```

Listing 2.17 Benutzerdefinierter Handler

Ein weiteres Beispiel für einen `DelegatingHandler` weist der `LimitResultMessageHandler` in Listing 2.18 auf. Er prüft, ob die angestoßene Action-Methode ein `IEnumerable` zurückgeliefert hat. Ist dem so, limitiert er dessen Inhalt auf die ersten drei Einträge.

```
public class LimitResultMessageHandler : DelegatingHandler
{

    protected override async System.Threading.Tasks.Task<HttpResponseMessage> SendAsync(
                                                                HttpRequestMessage request,
                                       System.Threading.CancellationToken cancellationToken)
    {
        var response = await base.SendAsync(request, cancellationToken);

        var objectContent = response.Content as ObjectContent;
        if (objectContent == null) return response;

        var collection = objectContent.Value as IEnumerable<object>;
        if (collection == null) return response;
```

```
        if (collection.Count() > 3)
        {
            return request.CreateResponse(
                            response.StatusCode,
                            collection.Take(3),
                            objectContent.Formatter);
        }

        return response;
    }
}
```

Listing 2.18 `MessageHandler` zum Limitieren der abgefragten Datenmenge

Auch benutzerdefinierte `DelegatingHandler` müssen der Web-API bekannt gemacht werden. Hierfür bietet sich abermals die Methode `WebApiConfig.Register` an. Den dazu heranzuziehenden Methodenaufruf zeigt Listing 2.19.

```
GlobalConfiguration.Configuration
                       .MessageHandlers.Add(new MethodOverrideHandler());
GlobalConfiguration.Configuration.MessageHandlers.Add(new LoggingHandler());
```

Listing 2.19 Handler registrieren

Handler mit HttpClient verwenden

Auch clientseitig können Handler zum Ausführen allgemeiner Querschnittsfunktionen herangezogen werden. In diesem Fall ist jedoch nicht von `DelegatingHandler` sondern von `MessageProcessingHandler` abzuleiten (Listing 2.20). Zusätzlich müssen die Methoden `ProcessRequest` und `ProcessResponse` aufgerufen werden. Wie die Namen schon vermuten lassen, ruft der `HttpClient` die Methode `ProcessRequest` auf, bevor er eine Anfrage sendet, und `ProcessResponse`, nachdem er eine Antwort empfangen hat.

```
class LoggingMessageHandler : MessageProcessingHandler
{
    public LoggingMessageHandler() : base() { }

    public LoggingMessageHandler(MessageProcessingHandler h) : base(h) { }

    protected override HttpRequestMessage ProcessRequest(HttpRequestMessage request,
                                          System.Threading.CancellationToken cancellationToken)
    {
        Debug.WriteLine("Request: " + request.RequestUri.ToString());
        return request;

    }

    protected override HttpResponseMessage ProcessResponse(HttpResponseMessage response,
                                          System.Threading.CancellationToken cancellationToken)
    {
        Debug.WriteLine("Response: " + response.StatusCode);
        return response;
    }
}
```

Listing 2.20 Protokollierungshandler

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Ein weiterer Unterschied zu DelegatingHandler besteht darin, dass beim Einsatz von MessageHandler manuell eine Aufrufkette zu erzeugen ist – serverseitig kümmert sich die API darum. Am Ende dieser Kette muss sich eine Instanz von HttpClientHandler befinden. Dieser Handler kümmert sich um die Verarbeitung der empfangenen Nachricht. Listing 2.21 demonstriert dies, indem es eine zweiteilige Kette erzeugt, die aus dem zuvor gezeigten LoggingMessageHandler und dem obligatorischen HttpClientHandler besteht. Der erste Knoten dieser Kette wird anschließend an den Konstruktor von HttpClient übergeben.

```
// Handler instanziieren und verketten
var handler = new LoggingMessageHandler
{
    InnerHandler = new HttpClientHandler()
};

// Client erzeugen
var client = new HttpClient(handler);

// Liste mit Formatter bereitstellen
var formatters = new List<MediaTypeFormatter>();
formatters.Add(new FlatFileFormatter());

// Accept-Header auf text/csv setzen
client.DefaultRequestHeaders.Accept.Add(
                  new MediaTypeWithQualityHeaderValue("text/csv"));

var url = "http://localhost:1307/api/Hotels";

// Daten anfordern
var response = await client.GetAsync(url);
var hotels = await response.Content.ReadAsAsync<IEnumerable<Hotel>>(formatters);

foreach (var hotel in hotels)
{
    Console.WriteLine(hotel.Bezeichnung);
}
```

Listing 2.21 HttpClient mit Handler nutzen

Als Alternative zum manuellen Erzeugen von Handlerketten kann der Entwickler auch die Methode Create der statischen HttpClientFactory verwenden. Diese verkettet die übergebenen Handler und platziert am Ende den obligatorischen HttpClientHandler. Mit dieser Kette erzeugt sie anschließend einen HttpClient und gibt ihn zurück:

```
var client = HttpClientFactory.Create(new LoggingMessageHandler());
```

An Create können dabei beliebig viele Handler übergeben werden, zumal diese Methode unter anderem ein Parameter-Array mit Handlern erwartet.

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Querschnittsfunktionen mit Filter realisieren

Neben Handler kann der Entwickler auch Filter zur Realisierung von Querschnittsfunktionen einsetzen. Dieses Konzept, welches dem gleichnamigen Konzept aus ASP.NET MVC gleicht, kann jedoch nur serverseitig verwendet werden. Im Gegensatz zu Handlern kann der Entwickler Filter gezielt für ausgewählte Operationen aktivieren.

Ein weiterer Unterschied zu Handler ist, dass es drei verschiedene Filterarten gibt: *ActionFilter*, *AuthorizationFilter* und *ExceptionFilter*. Um Verwechslungen mit der MVC-Welt zu vermeiden, muss der Entwickler darauf achten, die Komponenten aus dem Namensraum `System.Web.Http` und nicht ihre Gegenstücke aus dem Namensraum `System.Web.Mvc` zu verwenden.

Autorisierungsfilter (`AuthorizationFilter`) kommen zum Einsatz, wenn geprüft werden soll, ob der aktuelle Benutzer die adressierte Operation ausführen darf. Liefert der Filter zum Beispiel eine `SecurityException`, wird die Operation nicht ausgeführt. Die Web-API aktiviert Ausnahmefilter (`ExceptionFilter`), wenn Ausnahmen auftreten. Sie können eingesetzt werden, um Fehler zu protokollieren. Action-Filter werden hingegen eingesetzt, um Logiken vor und/oder nach der eigentlichen Dienstoperation zur Ausführung zu bringen.

Zur Implementierung von Filtern leitet der Entwickler von einer Basisklasse ab, die ASP.NET Web API für die gewünschte Filterart vorsieht. Tabelle 2.1 gibt eine Übersicht über diese Klassen und die im Zuge des Ableitens zu überschreibenden Methoden.

Schnittstelle	Methode	Beschreibung
`AuthorizationFilterAttribute`	`OnAuthorization`	Wird ausgeführt, bevor die Anfrage abgearbeitet wird
`ActionFilterAttribute`	`OnActionExecuting`	Wird vor der Action-Methode ausgeführt
	`OnActionExecuted`	Wird nach der Action-Methode ausgeführt
`ExceptionFilterAttribute`	`OnException`	Wird ausgeführt, wenn eine Ausnahme ausgelöst wurde

Tabelle 2.1 Übersicht über die Filterarten

Ein Beispiel für eine Implementierung der drei Filterarten findet sich in Listing 2.22. Die implementierten Methoden nehmen Informationen zum aktuellen Operationsaufruf über den übergebenen Kontext entgegen und geben diese aus. Stattdessen könnten Filter auch die einzelnen Kontexteigenschaften abändern bzw. im Fehlerfall eine Ausnahme auslösen.

```
public class SampleExceptionFilterAttribute : ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext actionExecutedContext)
    {
        Debug.WriteLine("SampleExceptionFilterAttribute.OnException");
        Debug.WriteLine("Exception: " + actionExecutedContext.Exception.Message);
    }
}

public class SampleAuthorizationFilterAttribute : AuthorizationFilterAttribute
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
{
    public override void OnAuthorization(System.Web.Http.Controllers.HttpActionContext actionContext)
    {
        Debug.WriteLine("SampleAuthorizationFilterAttribute.OnAuthorization");
        Debug.WriteLine("Authorization-Header" + actionContext.Request.Headers.Authorization);
    }
}

public class SampleActionFilterAttribute : ActionFilterAttribute
{

    public override void OnActionExecuting(System.Web.Http.Controllers.HttpActionContext actionContext)
    {
        var request = actionContext.Request;
        var response = actionContext.Response;
        var action = actionContext.ActionDescriptor;
        var actionArguments = actionContext.ActionArguments;
        var modelState = actionContext.ModelState;

        Debug.WriteLine("SampleActionFilterAttribute.OnActionExecuting");
        Debug.WriteLine("HTTP Method: " + request.Method);
        Debug.WriteLine("Uri: " + request.RequestUri);
        Debug.WriteLine("ActionName: " + action.ActionName);
        Debug.WriteLine("ReturnType: " + action.ReturnType);
        Debug.WriteLine("Arguments: " + actionArguments.Count);
        Debug.WriteLine("IsValid: " + modelState.IsValid);

    }

    public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)
    {
        var request = actionExecutedContext.Request;
        var response = actionExecutedContext.Response;
        var actionContext = actionExecutedContext.ActionContext;
        var exception = actionExecutedContext.Exception;

        Debug.WriteLine("SampleActionFilterAttribute.OnActionExecuting");
        Debug.WriteLine("HTTP Method: " + request.Method);
        Debug.WriteLine("Uri: " + request.RequestUri);
        Debug.WriteLine("ActionName: " + actionContext.ActionDescriptor.ActionName);

        if (exception != null)
        {
            Debug.WriteLine("Exception: " + exception.Message);
        }

        if (response != null && response.Content != null)
        {
            Debug.WriteLine("Content: " + response.Content.ReadAsStringAsync().Result);
        }

    }
}
```

Listing 2.22 Beispielhafte Filter

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Listing 2.23 zeigt, dass Filter angewendet werden können, indem `ApiController` oder Dienstoperationen damit annotiert werden. Wird ein `ApiController` annotiert, kommt der Filter für sämtliche Operationen des Controllers zum Einsatz; ansonsten lediglich für die jeweilige Operation.

```
[SampleAuthorizationFilter]
public class FilterSampleController : ApiController
{
    private static string value = "42";
    [SampleActionFilter]
    public string Get()
    {
        return value;
    }
    public void Post([FromUri] string newValue) {

        if (string.IsNullOrEmpty(newValue)) throw new ArgumentException("Darf nicht null oder leer sein!");
        value = newValue;
    }
}
```

Listing 2.23 Filter anwenden, indem Controller bzw. Operationen damit annotiert werden

Eine weitere Möglichkeit zum Anwenden von Filtern stellt die Implementierung von `FilterProvider` dar. Bei einem `FilterProvider` handelt es sich um eine Klasse, welche die Schnittstelle `IFilterProvider` realisiert. Dieses Interface gibt die Methode `GetFilter` vor. Immer, wenn eine Dienstoperation angestoßen werden soll, ruft ASP.NET Web API diese Methode auf und übergibt Informationen über den gewünschten Aufruf. Die Aufgabe von `GetFilter` besteht darin, sämtliche Filter zu ermitteln, die im Zuge des jeweiligen Aufrufs zu verwenden sind, und diese zurückzuliefern.

Ein Beispiel dafür findet sich in Listing 2.24. Es beinhaltet einen `FilterProvider`, welcher prüft, ob der Aufrufer gerade eine POST-Anfrage an `FilterSampleController` sendet. Ist dem so, wird eine Instanz von `FilterInfo` erzeugt, die auf ein neues `SampleExceptionFilterAttribute` verweist. Anschließend wird dieses `FilterInfo`-Objekt innerhalb einer Liste zurückgegeben.

```
public class CustomFilterProvider: IFilterProvider
{
    public IEnumerable<FilterInfo> GetFilters(System.Web.Http.HttpConfiguration configuration,↙
 System.Web.Http.Controllers.HttpActionDescriptor actionDescriptor)
    {
        var result = new List<FilterInfo>();

        if (actionDescriptor.ActionName == "Post" &&
actionDescriptor.ControllerDescriptor.ControllerType == typeof(FilterSampleController))
        {
            var filterInfo = new FilterInfo(new SampleExceptionFilterAttribute(), FilterScope.Action);
            result.Add(filterInfo);
        }

        return result;
    }
}
```

Listing 2.24 Filterprovider

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Damit ASP.NET Web API einen FilterProvider einsetzt, muss ihn der Entwickler bei der Konfiguration registrieren, zum Beispiel innerhalb der Methode WebApiConfig.Register. Das nachfolgende Schnipsel zeigt, wie dies für den zuvor betrachteten CustomFilterProvider bewerkstelligt werden kann:

```
config.Services.Add(typeof(IFilterProvider), new CustomFilterProvider());
```

Wie dieser Schnipsel vermuten lässt, können beliebig viele FilterProvider auf diese Weise registriert werden.

Soll ein Filter bei sämtlichen Dienstoperationen zum Einsatz kommen, besteht auch die Möglichkeit, ihn in der Konfiguration unter Verwendung der Eigenschaft Filter zu registrieren:

```
config.Filters.Add(new SampleActionFilterAttribute());
```

Als Alternative zum Ableiten von den zuvor besprochenen Basisklassen kann der Entwickler auch die Schnittstellen IExceptionFilter, IAuthorizationFilter und IActionFilter implementieren. Auf diesem Weg können Filter bereitgestellt werden, die gleichzeitig mehrere Filterarten darstellen bzw. auch von anderen Klassen abgeleitet werden können. Sollen die auf diesem Weg entwickelten Filter auch als Attribute eingesetzt werden, muss der Entwickler jedoch zusätzlich von der Klasse FilterAttribute ableiten.

Benutzerdefinierte Formate unterstützen

Neben *JSON* und *XML* können weitere Formate unterstützt werden, indem eine benutzerdefinierte Subklasse von MediaTypeFormatter bzw. BufferedMediaTypeFormatter bereitgestellt wird. Erstere sieht den Einsatz asynchroner Methoden zum Lesen und Schreiben von Objekten vor. Deren Subklasse BufferedMediaTypeFormatter macht diese asynchrone API über synchrone Methoden zugänglich.

Ein Beispiel dafür stellt der FlatFileFormatter in Listing 2.25 dar. Dieser bietet die Möglichkeit, eine List<Hotel> als CSV-Datei zu serialisieren sowie Daten, die in diesem Format vorliegen, wieder als List<Hotel> zu deserialisieren.

Im Konstruktor wird der zu verwendende Mime-Typ auf text/csv festgelegt. Dies veranlasst die Web-API dazu, den FlatFileFormatter immer dann in Erwägung zu ziehen, wenn mit Daten dieses Mime-Typs gearbeitet werden soll. Durch weitere analoge Aufrufe könnten mit dem vorliegenden FlatFileFormatter auch weitere Mime-Typen assoziiert werden.

Die überschriebenen Methoden CanReadType und CanWriteType zeigen an, dass der FlatFileFormatter lediglich Objekte des Typs List<Hotel> (de)serialisieren kann; die Methoden OnWriteStream und OnReadStream und legen die zum Serialisieren bzw. Deserialisieren zu verwendende Logik fest.

```
public class FlatFileFormatter : BufferedMediaTypeFormatter
{
    public FlatFileFormatter()
    {
        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
    }

    public override bool CanReadType(Type type)
    {
        return typeof(IEnumerable<Hotel>).IsAssignableFrom(type);
    }
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
    public override bool CanWriteType(Type type)
    {
        return typeof(IEnumerable<Hotel>).IsAssignableFrom(type);
    }

    public override object ReadFromStream(Type type, Stream readStream,
                                          System.Net.Http.HttpContent content,
                                          IFormatterLogger formatterLogger)
    {
        var hotels = new List<Hotel>();
        StreamReader r = new StreamReader(readStream);

        string line;
        while ((line = r.ReadLine()) != null)
        {
            if (line.Trim() == "") continue;
            var cols = line.Split(',');
            var hotel = new Hotel
            {
                HotelId = Convert.ToInt32(cols[0]),
                Bezeichnung = cols[1],
                Sterne = Convert.ToInt32(cols[2])
            };
            hotels.Add(hotel);
        }

        return hotels;
    }

    public override void WriteToStream(Type type, object value,
                      Stream writeStream, System.Net.Http.HttpContent content)
    {
        var hotels = (IEnumerable<Hotel>)value;
        if (hotels == null) return;

        StreamWriter w = new StreamWriter(writeStream);
        foreach (Hotel h in hotels)
        {
            w.WriteLine(h.HotelId + "," + h.Bezeichnung + "," + h.Sterne);
        }
        w.Flush();

    }
}
```

Listing 2.25 Formatter für CSV-Dateien

Damit die Web-API einen benutzerdefinierten `MediaTypeFormatter` verwenden kann, ist der Entwickler angehalten, diesen zu registrieren. Dazu bietet sich abermals die Methode `WebApiConfig.Register` an. Listing 2.26 zeigt den dafür benötigten Aufruf.

```
[…]
GlobalConfiguration.Configuration.Formatters.Add(new FlatFileFormatter());
[…]
```

Listing 2.26 Benutzerdefinierten Formatter registrieren

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Um einen Formatter in Aktion zu erleben, kann Fiddler herangezogen werden, zumal der Entwickler damit via HTTP Anfragen generieren kann. Abbildung 2.3 zeigt einen solchen Aufruf sowie eine dazu passende Antwort in Fiddler. Dabei ist zu beachten, dass das gewünschte Format bei der Anfrage im Kopfzeileneintrag `Accept` angegeben wurde.

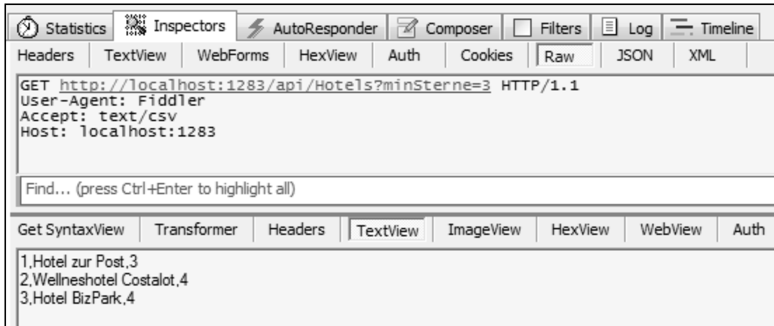

Abbildung 2.3 Mittels Fiddler Informationen im CSV-Format anfordern

Formatter mit HttpClient verwenden

Um auch clientseitig Formate jenseits von JSON und XML beim Senden von Anfragen heranzuziehen, hinterlegt der Entwickler beim Erzeugen einer `ObjectContent`-Instanz den gewünschten Formatter. Das nachfolgende Schnipsel führt den von der Web-API bereitgestellten `JsonMediaTypeFormatter` an.

```
var request = new HttpRequestMessage();
request.Content = new ObjectContent<HotelBuchung>(buchung,
                                        new JsonMediaTypeFormatter());
```

Eine Liste der Formatter, die beim Deserialisieren der empfangenen Daten Verwendung finden sollen, kann darüber hinaus beim Aufruf von `ReadAsync` angegeben werden (Listing 2.27). Den Mime-Typ der zu sendenden Nachricht gibt der Entwickler über den Kopfzeileneintrag `Content-Type` an; den als Antwort Erwarteten mittels `Accept`.

```
// Client erzeugen
var client = new HttpClient();

// Liste mit Formattern bereitstellen
var formatters = new List<MediaTypeFormatter>();
formatters.Add(new FlatFileFormatter());

// Accept-Header auf text/csv setzen
client.DefaultRequestHeaders.Accept.Add(
             new MediaTypeWithQualityHeaderValue("text/csv"));

var url = "http://localhost:1307/api/Hotels";

// Daten anfordern
var response = await client.GetAsync(url);
var hotels = await response
                         .Content
                         .ReadAsAsync<IEnumerable<Hotel>>(formatters);
```

```
foreach (var hotel in hotels)
{
    Console.WriteLine(hotel.Bezeichnung);
}
```

Listing 2.27 `HttpClient` mit Formattern nutzen

Validieren

Die an eine Operation übersendeten Daten können auf dieselbe Weise wie Modelle bei ASP.NET MVC validiert werden. Informationen darüber finden sich im ersten Kapitel.

Serialisierung beeinflussen

ASP.NET Web API bietet standardmäßig einen JSON- sowie einen XML-Serializer. Diese können, zum Beispiel zur Vermeidung von Interoperabilitätsproblemen, angepasst werden.

JSON-Serializer konfigurieren

Details der Serialisierung sowie den zu verwendenden JSON- bzw. XML-Serializer kann der Entwickler über die globale Konfiguration anpassen – als Ort für diese Anpassungen bietet sich, wie so häufig, die Methode `WebApiConfig.Register` an. Listing 2.28 demonstriert einige Einstellungsmöglichkeiten für jenen Serializer, der standardmäßig vom `JsonFormatter` herangezogen wird.

`UseDataContractJsonSerializer` definiert, ob der aus WCF bekannte `DataContractJsonSerializer` zum Einsatz kommen soll. Obwohl diese Einstellung aus Gründen der Kompatibilität zu WCF-basierten Systemen sinnvoll erscheint, sollte sie gerade bei Neuentwicklungen auf ihrem Standardwert `false` belassen werden. Dies hat zur Folge, dass ASP.NET Web API auf den mächtigeren Serializer aus dem freien Projekt *JSON.Net*, welches mit der Web-API ausgeliefert wird, zurückgreift.

`DateTimeZoneHandling` legt fest, welche Zeitzone bei der Übertragung von Datumswerten zu verwenden ist. Die im betrachteten Fall verwendete Option `DateTimeZoneHandling.Local` bewirkt, dass ASP.NET Web API Uhrzeiten als lokale Uhrzeiten (unter Berücksichtigung der Einstellungen am Server) inkl. Zeitzonenoffset überträgt. Alternativen sind `DateTimeZoneHandling.Utc` sowie `DateTimeZoneHandling.RoundtripKind`. Erstere konvertiert Uhrzeiten immer nach UTC; Letztere erhält die Zeitzone, die der Client übersendet hat.

Eine der wohl wichtigsten Eigenschaften, die den Umgang mit Datumswerten beeinflusst, ist `DateFormatHandling`. Sie legt fest, wie Datumswerte in JSON dargestellt werden sollen und adressiert somit die Problemstellung, dass die JSON-Spezifikation keine Aussage darüber macht. Die Option `IsoDateFormat`, welche auch den Standardwert darstellt, bewirkt, dass ASP.NET Web API Datumswerte wie in XML üblich nach ISO 8601 darstellt. Ein Beispiel dafür ist *2013-01-20T21:00:00+01:00*. Das *T* trennt dabei den Datumsteil von der Uhrzeit und am Ende wird ein eventueller Zeitzonenoffset angehängt. Bei +01:00 handelt es sich um das Offset für die mitteleuropäische Zeit (MEZ), welche von der Standardzeit (UTC) um eine Stunde abweicht.

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Eine Alternative hierzu stellt die Option `MicrosoftDateFormat` dar. Sie legt fest, dass jene Darstellung für Datumswerte heranzuziehen ist, die Microsoft-Frameworks im Zuge der JSON-Serialisierung in der Vergangenheit verwendet haben. Zugunsten der allgemein akzeptierten ISO-Repräsentation sollte der Entwickler diese Einstellung nur nutzen, wenn das entwickelte System zu solchen Systemen kompatibel sein muss.

Eine weitere Eigenschaft, die in Listing 2.28 verwendet wird, ist `Formatting`. Wird sie auf `Formatting.Indented` gesetzt, kommen – zur besseren Lesbarkeit – Zeilenschaltungen und Einrückungen zum Einsatz. Ansonsten verzichtet der JSON-Serializer darauf.

```
var jf = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
jf.UseDataContractJsonSerializer = false;
jf.SerializerSettings.DateTimeZoneHandling =
                            Newtonsoft.Json.DateTimeZoneHandling.Local;
jf.SerializerSettings.DateFormatHandling =
                            Newtonsoft.Json.DateFormatHandling.IsoDateFormat;
// jf.SerializerSettings.DateFormatHandling =
                            Newtonsoft.Json.DateFormatHandling.MicrosoftDateFormat;
jf.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
```

Listing 2.28 JSON-Serialisierung anpassen

XML-Serializer konfigurieren

Ähnlich wie der Serializer, der standardmäßig vom `JsonFormatter` verwendet wird, kann auch jener, der standardmäßig vom XML-Formatter herangezogen wird, konfiguriert werden. Listing 2.29 demonstriert dies. Die Option `UseXmlSerializer` legt fest, ob der `XmlSerializer`, der seit .NET 1.0 mit von der Partie ist, zum Serialisieren heranzuziehen ist. Wird diese Option auf `false` gesetzt, kommt der `DataContractSerializer`, der auch von der WCF genutzt wird, zum Einsatz. `Ident` gibt darüber hinaus Auskunft, ob zur besseren Lesbarkeit Einrückungen verwendet werden sollen.

```
var xf = GlobalConfiguration.Configuration.Formatters.XmlFormatter;

xf.UseXmlSerializer = true;
xf.Indent = true;
```

Listing 2.29 XML-Serialisierung anpassen

Eigenschaften von der Serialisierung ausschließen

Um festzulegen, dass der JSON-Serializer bestimmte Eigenschaften nicht serialisieren soll, annotiert sie der Entwickler mit dem Attribut `JsonIgnore`. Damit der `XmlSerializer` das Attribut ebenfalls nicht serialisiert, ist es mit `XmlIgnore` zu annotieren. Beim Einsatz des aus der WCF bekannten `DataContractSerializer`- bzw. `DataContractJsonSerializer`-Objekts wird die jeweilige Eigenschaft hingegen nicht mit `DataMember` annotiert. Listing 2.30 demonstriert dies mit der Klasse `Abteilung`, deren Eigenschaft `Budget` in keinem der soeben beschriebenen Fälle serialisiert wird.

```
[DataContract]
public class Abteilung
{
    [DataMember(Name="AbteilungsName", IsRequired=true)]
    public String Bezeichnung { get; set; }
```

```
    [DataMember]
    public Mitarbeiter Manager { get; set; }

    [JsonIgnore]
    [XmlIgnore]
    public string Budget{ get; set; }
}
```

Listing 2.30 Serialisierung von Eigenschaften durch Annotationen unterdrücken

Zirkuläre Referenzen serialisieren

Da sowohl XML als auch JSON prinzipiell Daten in hierarchischer Form repräsentieren, sind Fälle, in denen zwei Elemente gegenseitig aufeinander verweisen, problematisch. Diese Problemstellung wird durch die beiden Klassen in Listing 2.31 veranschaulicht: Die Klasse `Mitarbeiter` verweist auf die Klasse `Abteilung` und die Klasse `Abteilung` verweist über die Eigenschaft `Manager` zurück auf die Klasse `Mitarbeiter`. Bei einer strikten hierarchischen Serialisierung würde sich eine Endlosschleife ergeben.

```
public class Mitarbeiter
{
    public String Name { get; set; }
    public Abteilung Abteilung { get; set; }
}
public class Abteilung
{
    public String Bezeichnung { get; set; }
    public Mitarbeiter Manager { get; set; }
    public string Budget{ get; set; }
}
```

Listing 2.31 Zirkuläre Verweise

Um dieses Problem zu umgehen, müssen die verwendeten Serializer angewiesen werden, Referenzen zu verwenden, anstatt referenzierte Elemente an Ort und Stelle zu platzieren und somit ggf. zu wiederholen. Der Serializer des `JsonFormatters` kann über seine Eigenschaft `PreserveReferencesHandling` zu diesem Verhalten bewegt werden (Listing 2.32).

```
var jf = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
[…]
jf.SerializerSettings.PreserveReferencesHandling = PreserveReferencesHandling.All;
```

Listing 2.32 Referenzen aktivieren

Setzt der Entwickler diese Eigenschaft auf die Option `PreserveReferencesHandling.All`, vergibt der Serializer jedem serialisierten Objekt eine ID und Verweise auf diese Objekte werden unter Angabe dieser IDs modelliert. Listing 2.33 demonstriert dies, indem es das Ergebnis der JSON-Serialisierung der Mitarbeiterin Susi Sorglos zeigt, welche Managerin der eigenen Abteilung ist. Dabei fällt auf, dass der Serialisierer zum Darstellen von IDs die einzelnen Objekte um eine Eigenschaft `$id` erweitert hat. Zum Darstellen von Verweisen hat er darüber hinaus die Eigenschaft `$ref` eingeführt.

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Beim Einsatz dieser Lösung ist zu beachten, dass es sich hierbei um eine Erweiterung von JSON.Net handelt, welche somit nicht von allen Kommunikationspartnern verstanden wird.

```
{
    "$id": "1",
    "Name": "Susi Sorglos",
    "Abteilung": {
       "$id": "2",
       "Bezeichnung": "Abteilung Einkauf",
       "Manager": {
           "$ref": "1"
       }
    }
}
```

Listing 2.33 JSON-Objekt mit Referenzen

Der `DataContractSerializer` bietet für XML-Dokumente mit zirkulären Verweisen zwei ähnliche Lösungsansätze. Zum einen kann der Entwickler bei der Definition eines Datenvertrags angeben, dass Verweise auf diesen Datenvertrag durch Angabe von IDs aufzulösen sind. Dazu legt er die Eigenschaft `IsReference` des Attributs `DataContract` auf `true` fest (siehe Listing 2.34).

```
[DataContract(IsReference = true)]
public class Abteilung
{
    [DataMember(Name="AbteilungsName", IsRequired=true)]
    public String Bezeichnung { get; set; }

    [DataMember]
    public Mitarbeiter Manager { get; set; }

    [JsonIgnore]
    [XmlIgnore]
    public string Budget{ get; set; }
}

[DataContract(IsReference = true)]
public class Abteilung
{
    [DataMember]
    public String Bezeichnung { get; set; }

    [DataMember]
    public Mitarbeiter Manager { get; set; }
}
```

Listing 2.34 Referenzen für Datenverträge aktivieren

Alternativ dazu kann der Entwickler pro Datenvertrag in der globalen Konfiguration einen vorkonfigurierten `DataContractSerializer` hinterlegen. Damit diese Serializer zum Auflösen von Verweisen ID-Referenzen einsetzen, übergibt er den Wert `true` an den Konstruktor-Parameter `preserveObjectReferences` (Listing 2.35).

```
var dcsAbteilung = new DataContractSerializer(typeof(Abteilung), null,
                            int.MaxValue, false,
                            /* preserveObjectReferences: */ true, null);
xf.SetSerializer<Abteilung>(dcsAbteilung);

var dcsMitarbeiter = new DataContractSerializer(typeof(Mitarbeiter), null,
                                int.MaxValue, false,
                                /* preserveObjectReferences: */ true, null);
xf.SetSerializer<Mitarbeiter>(dcsMitarbeiter);
```

Listing 2.35 Vorkonfigurierte Serializer registrieren

Web-API und HTML-Formulare

Neben der Tatsache, dass ASP.NET Web API XML- und JSON-basierte Daten empfangen und senden kann, besteht auch die Möglichkeit, Daten in jenen Formaten entgegenzunehmen, in denen Browser sie senden. Somit können Action-Methoden auch als Ziel von HTML-basierten Formularen dienen.

Einfache Formularfelder übermitteln

Die standardmäßig eingerichteten Formatter binden die übersendeten Felder eines HTML-Formulars an die Übergabeparameter der angestoßenen Action-Methode. Daneben kann auch mit `Request.Content.ReadAsFormDataAsync` eine Auflistung mit sämtlichen Formularparametern abgerufen werden (siehe Listing 2.36).

```
public async Task<string> Post()
{
    if (Request.Content.IsFormData())
    {
        var formData = await Request.Content.ReadAsFormDataAsync();
        return formData["message"];
    }
    return "No Message!";
}
```

Listing 2.36 Daten eines HTML-Formulars lesen

Dateiupload via HTML-Formular

Um Dateiuploads, die von einem HTML-Formular aus erfolgen, verarbeiten zu können, muss ASP.NET Web API dazu gebracht werden, mit Daten umzugehen, die sich am Mime-Typ `multipart/form-data` orientieren. Dieser Mime-Type sieht vor, dass die übersendeten Informationen in mehrere Sektionen geteilt werden, wobei jede Sektion einen eigenen Mime-Typ aufweisen kann. Diese Sektionen beinhalten zum Beispiel die hochzuladenden Dateien oder auch die Inhalte von Formularfeldern.

Zum Lesen einer solchen Nachricht verwendet der Entwickler die Methode `Content.ReadAsMultipartAsync` des aktuellen `Request`-Objekts (Listing 2.37). An diese kann er eine Instanz einer Subklasse von `MultipartStreamProvider` übergeben, wobei diese die Art der Verarbeitung der übersendeten Daten festlegt.

Im betrachteten Fall kommt ein `MultipartFormDataStreamProvider` zum Einsatz. Dieser speichert die hochgeladenen Daten in jenem Ordner, den der Entwickler über dessen Konstruktor festgelegt hat. Darüber hinaus

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

hält er die Daten von Formularfeldern im Hauptspeicher vor und macht sie über die Eigenschaft `FormData` zugänglich. Die Auflistung `FileData` beinhaltet daneben Informationen über die hochgeladenen Dateien, welche im spezifizierten Verzeichnis abgelegt wurden.

Die einzelnen Einträge sind vom Typ `MultipartFileData`, welcher zwei Eigenschaften aufweist: `LocalFileName` repräsentiert den vollständigen Namen der hochgeladenen Datei im festgelegten Uploadordner des Servers; `Headers` liefert Kopfzeileneinträge, welche der Browser für die jeweilige Datei übersendet hat. Über den Kopfzeileneintrag `Content-Disposition` kann zum Beispiel jener Name, den die Datei auf der Clientseite hat, ermittelt werden; der Kopfzeileneintrag `Content-Type` gibt hingegen Auskunft über das Dateiformat (über `Content-Type`) der Datei. Im betrachteten Beispiel werden diese Informationen zur Demonstration im Debugfenster ausgegeben. Listing 2.38 zeigt zur Demonstration jene Ausgaben, die beim Hochladen einer ZIP-Datei entstanden sind. Eine HTML-Seite zum Testen dieser Action-Methode findet sich in Listing 2.39.

```
public async Task<string> Post()
{
    if (!Request.Content.IsMimeMultipartContent("form-data"))
    {
        throw new HttpResponseException(HttpStatusCode.UnsupportedMediaType);
    }

    var formDataProvider = new
               MultipartFormDataStreamProvider(@"c:\temp\bilder");
    var bodyParts = await
              Request.Content.ReadAsMultipartAsync(formDataProvider);

    var hotelId = bodyParts.FormData["hotelId"];
    Debug.WriteLine("HotelId: " + hotelId);

    foreach (var file in bodyParts.FileData)
    {
        Debug.WriteLine("File: " + file.LocalFileName);
        foreach (var h in file.Headers)
        {
            var array = h.Value as IEnumerable<string>;
            if (array != null) {
                Debug.WriteLine("   Header: " + h.Key + ": " + string.Join("|", array));
            }
            else {
                Debug.WriteLine("   Header: " + h.Key + ": " + h.Value);
            }

        }
    }

    return bodyParts.FileData.Count + " Dateien für Hotel #" + hotelId + " hochgeladen!";

}
```

Listing 2.37 File-Upload bearbeiten

```
HotelId: 333
File: c:\temp\bilder\BodyPart_2dc6a787-06f1-4dff-91cc-d1f48cf7d6e6
   Header: Content-Disposition: form-data; name="data";
                      filename="C:\Users\steyer\Desktop\ExtendingMVC.zip"
   Header: Content-Type: application/x-zip-compressed
```

Listing 2.38 Ausgabe der übersendeten Daten

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Hotel-Bilder</title>

    </head>
    <body>

        <form
            method="POST"
            action="http://localhost:1307/api/Bilder"
            enctype="multipart/form-data">

            <div>
                HotelId
            </div>
            <div>
                <input name="hotelId" >
            </div>
            <div>
                Bilder:
            </div>
            <div>
                <input name="data" type="file" multiple>
            </div>
            <input type="submit" />

        </form>

    </body>

</html>
```

Listing 2.39 HTML-Formular für Dateiupload

Neben dem `MultipartFormDataStreamProvider` existieren noch ein paar weitere Implementierungen, die in solchen Szenarien verwendet werden können. Tabelle 2.2 gibt Aufschluss darüber.

Implementierung	Beschreibung
MultipartMemoryStreamProvider	Speichert sämtliche Teile der Nachricht im Hautspeicher und stellt diese in Form von MemoryStream-Objekten bereit
MultipartFileStreamProvider	Speichert sämtliche Teile der Nachricht im Dateisystem
MultipartFormDataStreamProvider	Speichert nur hochgeladene Dateien im Dateisystem; Formularfelder werden im Hauptspeicher vorgehalten
MultipartRelatedStreamProvider	Unterstützt den Mime-Typ Multipart/Related (RFC 2387), bei dem einzelne Nachrichtenteile aufeinander verweisen können (zum Beispiel eine XML-Nachricht im ersten Teil auf ein Bild im zweiten Teil)

Tabelle 2.2 `MultipartStreamProvider`-Derivate

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Fortschritt ermitteln

Zum Ermitteln des Fortschritts eines Upload- bzw. Downloadvorgangs bietet ASP.NET Web API einen `ProgressMessageHandler` (Listing 2.40). Wie alle anderen `MessageHandler` auch, ist dieser mit allen anderen zu verwendenden Handlern zu verketten, wobei sich am Ende dieser Kette ein `HttpClientHandler` befinden muss. Nachdem er sie erzeugt hat, übergibt der Entwickler diese `MessageHandler`-Kette an den Konstruktor von `HttpClient`.

Der Eigenschaft `HttpReceiveProgress` von `ProgressMessageHandler` weist er zusätzlich eine Methode zu, die laufend über den Fortschritt eines Downloads informiert werden soll (siehe Listing 2.41). Diese Methode erhält bei jedem Aufruf ein Argument vom Typ `HttpProgressEventArgs`, das Zugriff auf Informationen, welche den aktuellen Fortschritt repräsentieren, gewährt. Die Eigenschaft `ProgressPercentage` enthält zum Beispiel den Grad der Fertigstellung in Prozent. Im betrachteten Fall wird dieser Wert ausgegeben, sofern er sich seit dem letzten Aufruf der Benachrichtigungs-Methode geändert hat.

HINWEIS Die Verwendung von `ProgressMessageHandler` schließt den Einsatz von Streaming aus.

Um sich über den Fortschritt eines Uploadvorgangs informieren zu lassen, geht der Entwickler analog vor. Allerdings verwendet er hierzu die Eigenschaft `HttpSendProgress` anstatt von `HttpReceiveProgress`.

```
static async void DownloadDemo()
{
     var progressMessageHandler = new ProgressMessageHandler()
    {
        InnerHandler = new HttpClientHandler()
    };

    progressMessageHandler.HttpReceiveProgress +=
                      progressMessageHandler_HttpReceiveProgress;

    HttpClient client = new HttpClient(progressMessageHandler);

    var response = await client.GetAsync("http://localhost:1307/api/Bilder");
    var stream = await response.Content.ReadAsStreamAsync();

    Console.WriteLine("Habe Stream erhalten!");
    long bytes = 0;

    using (FileStream fs =
                   new FileStream(@"c:\temp\download.dat", FileMode.Create))
    {
        int b;
        while ((b = stream.ReadByte()) != -1)
        {
            fs.WriteByte((byte)b);
            bytes++;
            if (bytes % (200 * 1024) == 0)
            {
                Console.WriteLine(bytes / 1024 + " KB gelesen ...");
            }
        }
    }
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
    Console.WriteLine("Fertig gelesen");

}
```

Listing 2.40 Einsatz von `ProgressMessageHandler`

```
static int progress = -1;
[…]
static void progressMessageHandler_HttpReceiveProgress(
                                 object sender, HttpProgressEventArgs e)
{
    if (progress == e.ProgressPercentage) return;
    Console.WriteLine(e.ProgressPercentage + "% heruntergeladen");
    progress = e.ProgressPercentage;
}
```

Listing 2.41 Ereignisbehandlungsroutine für `ProgressMessageHandler`

Feingranulare Konfiguration

Bis dato waren sämtliche beschriebenen Konfigurationseinstellungen global und wirkten sich somit auf sämtliche Controller aus. Dieser Abschnitt zeigt, wie Konfigurationseinstellungen für bestimmte Controller hinterlegt werden können.

Controllerbasierte Konfiguration

Obwohl die globale Konfiguration von ASP.NET Web API in vielen Fällen ausreicht, gibt es Situationen, in denen Einstellungen lediglich für einen einzelnen oder wenige Controller vorgenommen werden sollen. In diesen Situationen besteht die Möglichkeit, die Schnittstelle `IControllerConfiguration` zu implementieren (Listing 2.42). Dieses gibt die Methode `Initialize` vor, welche die Konfiguration eines bestimmten Controllers übernimmt. Dazu übergibt ASP.NET Web API einen Parameter vom Typ `HttpControllerSettings` sowie einen `HttpControllerDescriptor`. An Ersteren sind die gewünschten Konfigurationseinstellungen zu übergeben; Letzterer bietet Informationen über den zu konfigurierenden Controller.

Im betrachteten Fall entfernt `Initialize` den ersten Formatter – es handelt sich dabei um den JSON-Formatter – und ersetzt die zu verwendende `ITraceWriter`-Implementierung durch eine benutzerdefinierte, die – um das Beispiel kurz zu halten – von der eigenen Klasse bereitgestellt wird. Dazu wird der benutzerdefinierte `ITraceWriter` implementiert und somit auch eine Methode `Trace` bereitgestellt.

```
public class CustomConfig : Attribute, IControllerConfiguration, ITraceWriter
{
    public void Initialize(HttpControllerSettings controllerSettings,
                                    HttpControllerDescriptor controllerDescriptor)
    {
        controllerSettings.Formatters.RemoveAt(0);
        controllerSettings.Services.Replace(typeof(ITraceWriter), this);
    }

    public void Trace(HttpRequestMessage request, string category,
                          System.Web.Http.Tracing.TraceLevel level, Action<TraceRecord> traceAction)
```

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

```
    {
        TraceRecord record = new TraceRecord(request, category, level);
        traceAction(record);
        Debug.WriteLine(category + ": " + level + " " + record.Message);
    }
}
```

Listing 2.42 Controllerbasierte Konfiguration

Da die betrachtete Implementierung darüber hinaus auch von der Basisklasse `Attribute` erbt, kann sie zum Annotieren der zu konfigurierenden Controller herangezogen werden (Listing 2.43).

```
[CustomConfig]
public class MiniBarController : ApiController
{

    public double GetPrice(int hotelId, int roomNumber)
    {
        return 42;
    }

}
```

Listing 2.43 Einsatz einer controllerbasierten Konfiguration

Routenbasierte Konfiguration

Auch auf der Ebene von Routen kann der Entwickler die Standardkonfiguration variieren. Dazu hat er die Möglichkeit, einer Route eine Kette von `MessageHandler`-Objekten zuzuordnen. Listing 2.44 weist zum Beispiel der Route `BuchungenByHotelRoute` eine Kette bestehend aus einem `LimitResultMessageHandler` und einem `HttpControllerDispatcher` zu. Dabei ist zu beachten, dass Ketten dieser Art immer mit einem `HttpControllerDispatcher` abzuschließen sind, da dieser die entsprechende Action-Methode anstößt.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");


    var handler = new LimitResultMessageHandler()
    {
        InnerHandler = new HttpControllerDispatcher(GlobalConfiguration.Configuration)
    };

    routes.MapHttpRoute(
        name: "BuchungenByHotelRoute",
        routeTemplate: "api/Hotels/{hotelId}/Buchungen",
        defaults: new { controller = "Buchungen" },
        constraints: null, // Notwendig, damit richtige Überladung gewählt wird!
        handler: handler
    );

    […]
}
```

Listing 2.44 Konfiguration auf der Ebene einer Route

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)

Manfred Steyer, Dr. Holger Schwichtenberg: Moderne Webanwendungen mit ASP.NET MVC 4,
Microsoft Press 2013 (ISBN 978-3-86645-564-1)