Supporting Diverse Clients

At this point we've demonstrated ASP.NET Web API as an excellent platform on which to implement REST-based services. However, what if you found yourself in a situation where you had to support existing "legacy" (asmx-, or SOAP-based) clients? Or clients that required messages to be in XML format? Would that automatically eliminate ASP.NET Web API as a technology choice? Fortunately, the answer is no. And that leads us to the subject of the current chapter; namely, how to use ASP.NET Web API to simulteneously support legacy clients and extend reach to clients requiring various message formats. Along the way we'll throw in some ASP.NET Web API goodies, of course, just to keep it interesting for those who don't need to support SOAP-based clients or multiple message formats.

# The Situation

Recently, we were given the job of developing a REST-based Web API for an existing system. There were a couple of interesting requirements for this project:

1. New clients must be able to consume the service using their choice of either an XML or JSON message format.
2. The new REST-based API and the legacy SOAP-based API must be implemented and deployed as a single application.
3. Existing clients must be able to benefit from the new features of the new application without affecting any existing integration points; in other words, backwards compatibility is required.

An architectural overview of this is depicted in the following figures. Note that though the actual system didn't have anything to do with task-management, we will use our task-management service example for illustration and explanation of key concepts throughout this chapter:



Figure 8-1. Architectural Overview - Current State

In the current state (Figure 8-1), we see the legacy service integrating with clients via the SOAP messaging protocol. The target architecture (Figure 8-2), however, shows the task-management service concurrently supporting existing clients and new clients by offering a choice of messaging protocols and formats.



Figure 8-2. Architectural Overview - Target State

We'll begin this chapter by demonstrating how to support various message formats (i.e., XML and JSON) in our REST-based API; this will satisfy requirement 1. Then we will demonstrate how to support the existing SOAP-based clients with the same application, satisfying requirements 2 and 3.

# Content Negotiation

We introduced the concept of content negotiation, defined by the HTTP specification as “the process of selecting the best representation for a given response when there are multiple representations available”, way back in Chapter 1. We also made mention of it in Chapter 5. In this brief section we will demonstrate how to leverage it to support clients that require specific message formats.

The following HTTP request headers are the primary mechanism for content negotiation with services built using ASP.NET Web API:

* Accept: Used to specify which media types are acceptable for the response, such as application/json, application/xml, or a custom media type.
* Accept-Charset: Used to specify which character sets are acceptable, such as UTF-8 or ISO 8859-1.

Services built using ASP.NET Web API will respond to request messages received without these headers with a default content type and character set (application/json and utf-8, respectively). For example, here is the request-response message pair that gets all tasks using bhogg's highly-privileged credentials:

Get Tasks Request - Without Accept Header (abbreviated)

GET http://localhost:61589/api/v1/tasks HTTP/1.1

Authorization: Basic YmhvZ2c6aWdub3JlZA==

Get Tasks Response - Without Accept Header (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

{"Items":[{"TaskId":1,"Subject":"Triage the important tasks","StartDate":"2014-04-24T15:20:58","DueDate":null,"CreatedDate":"2014-04-24T11:19:41","CompletedDate":null,"Status":{"StatusId":2…

Note that the service responded with the default content type and character set (above), and notice the difference when the caller specifies that it would like to receive the response in XML format (below):

Get Tasks Request - With Accept Header (abbreviated)

GET http://localhost:61589/api/v1/tasks HTTP/1.1

Authorization: Basic YmhvZ2c6aWdub3JlZA==

Accept: application/xml

Get Tasks Response - With Accept Header (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/xml; charset=utf-8

<PagedDataInquiryResponseOfTaskQgo2FNlC xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/WebApi2Book.Web.Api.Models"><Items><Task><Assignees><User><Firstname>Boss</Firstname><Lastname>Hogg</Lastname><Links><Link><Href>http://localhost:61589/api/v1/users/1</Href><Method>GET</Method><Rel>self</Rel></Link></Links><UserId>1</UserId><Username>bhogg</Username>…

The best part is that this content negotiation is built into ASP.NET Web API. Simply by specifying the Accept header, the caller is able to control the format that the server uses to create the response message. Pretty cool - we have satisfied requirement 1!

There are limitations to this, of course. For example, the service can not produce a response message in a format it knows nothing of (i.e., fooBar):

Get Tasks Request - Unknown Content Type (abbreviated)

GET http://localhost:61589/api/v1/tasks HTTP/1.1

Authorization: Basic YmhvZ2c6aWdub3JlZA==

Accept: application/fooBar

Get Tasks Response - Unknown Content Type (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

{"Items":[{"TaskId":1,"Subject":"Triage the important tasks","StartDate":"2014-04-24T15:20:58","DueDate":null,"CreatedDate":"2014-04-24T11:19:41","CompletedDate":null,"Status":{"StatusId":2…

When presented with a request for an unknown content type like this, ASP.NET Web API tries to match on the media type of the request body, if any. In this case, the message was a GET, with no request body; therefore, the service simply returned the response message using the default format.

For request messages with an XML request body (i.e., PUT and POST messages), one very important thing to note is that you will need to specify the XML serializer so that the body can be parsed for model binding. This can easily be accomplished by adding the following line to the WebApiConfig.Register method:

config.Formatters.XmlFormatter.UseXmlSerializer = true;

This concludes our brief introduction to content negotiation, and this is probably all you'll need to know about it for most projects. However, content negotiation, and related concerns such as serialization and formatting, can be highly customized on the server side. If you'd like to learn more about this, we encourage you to visit the official Microsoft ASP.NET Web API site and make use of the excellent resources they have available. This one is particularly apropos: <http://www.asp.net/web-api/overview/formats-and-model-binding/content-negotiation>.

# Supporting SOAP-based Clients

A couple of things made it relatively easy for us to satisfy the remaining requirements, 2 and 3. First, the legacy-task management service was well-architected. We were able to use virtually all of the code implementing the business logic because it was decoupled from the asmx-based host plumbing. This example underscores why throughout this book we have emphasized the importance of "thin" controllers. After all, someday some exciting new platform will replace ASP.NET Web API, and it will be a whole lot easier to reuse your core business logic if it is not all entangled with ASP.NET Web API-specific types.

Second - and this was key - was the realization that SOAP requests are merely HTTP POST messages with an XML body. Once we knew this, we were sure that we could support existing clients with ASP.NET Web API... and some custom parsing and formatting to replace the functionality provided by the former asmx-based host.

Now we're ready to explore the implementation. However, because this particular topic may tend to appeal to a smaller audience, our explanation will be abridged. Instead of our usual exhaustive code walkthrough, including explanations of every dependency, we will highlight only the major components. We will also demonstrate some cool ASP.NET Web API features, including:

* A message handler that takes the place of a controller
* Route-specific message handlers
* Mixing convention-based routing with attribute-based routing
* A custom formatter

Let's begin!

## Where is The Controller?

First off, we decided to use a message handler, rather than a controller, to orchestrate the processing of request messages. The main reasons for this were

* We needed full control of the message parsing and formatting, both inbound and outbound
* We had no need for controller selection, action selection, or model binding, and we therefore didn't want to incur the overhead of these functions.

Keeping with our "thin" controller approach, we implemented the "thin" handler, and the supporting processor, as follows:

Message Handler

using System.Net;

using System.Net.Http;

using System.Threading;

using System.Threading.Tasks;

using System.Xml.Linq;

using WebApi2Book.Web.Common;

namespace WebApi2Book.Web.Api.LegacyProcessing

{

public class LegacyMessageHandler : DelegatingHandler

{

public virtual ILegacyMessageProcessor LegacyMessageProcessor

{

get { return WebContainerManager.Get<ILegacyMessageProcessor>(); }

}

protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,

CancellationToken cancellationToken)

{

var requestContentAsString = request.Content.ReadAsStringAsync().Result;

var requestContentAsDocument = XDocument.Parse(requestContentAsString);

var legacyResponse = LegacyMessageProcessor.ProcessLegacyMessage(requestContentAsDocument);

var responseMsg = request.CreateResponse(HttpStatusCode.OK, legacyResponse);

return Task.FromResult(responseMsg);

}

}

}

Message Processor

using System;

using System.Collections.Generic;

using System.Xml.Linq;

using WebApi2Book.Web.Api.LegacyProcessing.ProcessingStrategies;

namespace WebApi2Book.Web.Api.LegacyProcessing

{

public class LegacyMessageProcessor : ILegacyMessageProcessor

{

private readonly ILegacyMessageParser \_legacyMessageParser;

private readonly IEnumerable<ILegacyMessageProcessingStrategy> \_legacyMessageProcessingStrategies;

public LegacyMessageProcessor(ILegacyMessageParser legacyMessageParser,

IEnumerable<ILegacyMessageProcessingStrategy> legacyMessageProcessingStrategies)

{

\_legacyMessageParser = legacyMessageParser;

\_legacyMessageProcessingStrategies = legacyMessageProcessingStrategies;

}

public virtual LegacyResponse ProcessLegacyMessage(XDocument request)

{

var operationElement = \_legacyMessageParser.GetOperationElement(request);

var opName = operationElement.Name.LocalName;

foreach (var legacyMessageProcessingStrategy in \_legacyMessageProcessingStrategies)

{

if (legacyMessageProcessingStrategy.CanProcess(opName))

{

var legacyResponse = new LegacyResponse

{

Request = request,

ProcessingResult = legacyMessageProcessingStrategy.Execute(operationElement)

};

return legacyResponse;

}

}

throw new NotSupportedException(opName);

}

}

}

This implementation provides a nice overview of the processing. The message contents are read by the message handler and then parsed into an object that can be processed by the message processor. The result from the message processor is then packaged up in the response and returned as a generic Task as required by the SendAsync method signature.

The message processor is straightforward as well. It simply parses the operation element from the SOAP request message body, and, based on the operation name, finds the appropriate dependency to which it can delegate the rest of the work.

As an example, for the following request…

Get Tasks Request - SOAP (abbreviated)

POST http://localhost:61589/TeamTaskService.asmx HTTP/1.1

Content-Type: text/xml; charset=utf-8

SOAPAction: "http://tempuri.org/GetTasks"

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"><s:Body><GetTasks xmlns="http://tempuri.org/" xmlns:a="http://schemas.datacontract.org/2004/07/WebApi2Book.Windows.Legacy.Client.TaskServiceReference" xmlns:i="http://www.w3.org/2001/XMLSchema-instance"/></s:Body></s:Envelope>

… the following ILegacyMessageProcessingStrategy implementation would be used to do the work in this case because it supports the GetTasks operation (note the CanProcess method):

GetTasksMessageProcessingStrategy Class (abbreviated)

namespace WebApi2Book.Web.Api.LegacyProcessing.ProcessingStrategies

{

public class GetTasksMessageProcessingStrategy : ILegacyMessageProcessingStrategy

{

private readonly IAllTasksInquiryProcessor \_inquiryProcessor;

public GetTasksMessageProcessingStrategy(IAllTasksInquiryProcessor inquiryProcessor)

{

\_inquiryProcessor = inquiryProcessor;

}

public bool CanProcess(string operationName)

{

return operationName == "GetTasks";

}

…

Let's move on to explaining how we configured routing for this message handler.

## Configuring the Route

As we discussed in Chapter 5, attribute-based routing is a powerful feature to have at your disposal. However, attribute-based routing is designed for controllers and controller actions, not for message handlers. Fortunately, convention-based routing is still available, it supports routing directly to message handlers, and it can be used in the same application as attribute-based routing. Therefore, we modified the WebApiConfig class to appear as follows:

using System.Web.Http;

using System.Web.Http.Dispatcher;

using System.Web.Http.ExceptionHandling;

using System.Web.Http.Routing;

using System.Web.Http.Tracing;

using WebApi2Book.Common.Logging;

using WebApi2Book.Web.Api.LegacyProcessing;

using WebApi2Book.Web.Common;

using WebApi2Book.Web.Common.ErrorHandling;

using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api

{

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

config.Formatters.XmlFormatter.UseXmlSerializer = true;

ConfigureRouting(config);

config.Services.Replace(typeof(ITraceWriter),

new SimpleTraceWriter(WebContainerManager.Get<ILogManager>()));

config.Services.Add(typeof (IExceptionLogger),

new SimpleExceptionLogger(WebContainerManager.Get<ILogManager>()));

config.Services.Replace(typeof(IExceptionHandler), new GlobalExceptionHandler());

}

private static void ConfigureRouting(HttpConfiguration config)

{

config.Routes.MapHttpRoute(

name: "legacyRoute",

routeTemplate: "TeamTaskService.asmx",

defaults: null,

constraints: null,

handler: new LegacyAuthenticationMessageHandler(WebContainerManager.Get<ILogManager>())

{

InnerHandler = new LegacyMessageHandler { InnerHandler = new HttpControllerDispatcher(config) }

});

var constraintsResolver = new DefaultInlineConstraintResolver();

constraintsResolver.ConstraintMap.Add("apiVersionConstraint", typeof(ApiVersionConstraint));

config.MapHttpAttributeRoutes(constraintsResolver);

config.Services.Replace(typeof(IHttpControllerSelector),

new NamespaceHttpControllerSelector(config));

}

}

}

Besides adding the new "legacyRoute" route, you may have noticed that we also factored the routing code out into a separate method, ConfigureRouting. This was because the Register method was getting too complicated. As for the new route, the template was developed by examining the URLs of messages to the legacy task-management service (using Fiddler) to ensure that existing clients wouldn't break when they pointed their proxies to the new service.

Now let's turn our attention to the handler parameter. Here we have a route-specific handler, LegacyAuthenticationMessageHandler, used to perform custom legacy authentication. The LegacyAuthenticationMessageHandler implementation isn't particularly interesting (it's similar to the BasicAuthenticationMessageHandler discussed in Chapter 6), so we won't discuss it. However, the reason we configured LegacyAuthenticationMessageHandler here as a route-specific handler instead of a global handler (like BasicAuthenticationMessageHandler, for example) is so the handler would not burden other routes with unnecessary overhead; the handler is only relevant for this particular route.

Another reason is that we knew this would be a great way to demonstrate nested handlers. So, nested inside of the LegacyAuthenticationMessageHandler is our LegacyMessageHandler, which at run time is the next handler in the processing chain to receive the request message. Note how its inner handler is an HttpControllerDispatcher instance. Though nesting can go many levels deep, ultimately the nesting has to end, and it ends this way so that ASP.NET Web API is able to complete the processing pipeline diagrammed in Figure 5-1.

## Formatter

As we noted earlier, the LegacyMessageHandler returns a LegacyResponse object as part of the HttpResponseMessage. ASP.NET Web API will then serialize that object into the response message that the caller receives. Normally, this is all that is required. However, in the case of SOAP, the response message needs to be tweaked a bit so that it can consumed by the callers. We added the following custom formatter to do this tweaking:

using System;

using System.IO;

using System.Net;

using System.Net.Http;

using System.Net.Http.Formatting;

using System.Net.Http.Headers;

using System.Text;

using System.Threading.Tasks;

using System.Xml;

using System.Xml.Linq;

using WebApi2Book.Common;

using WebApi2Book.Common.Extensions;

namespace WebApi2Book.Web.Api.LegacyProcessing

{

public class LegacyMessageTypeFormatter : MediaTypeFormatter, ILegacyMessageTypeFormatter

{

private readonly ILegacyMessageParser \_legacyMessageParser;

public LegacyMessageTypeFormatter(ILegacyMessageParser legacyMessageParser)

{

\_legacyMessageParser = legacyMessageParser;

SupportedMediaTypes.Add(new MediaTypeHeaderValue(Constants.MediaTypeNames.TextXml));

}

public override bool CanReadType(Type type)

{

return false;

}

public override bool CanWriteType(Type type)

{

return type == typeof (LegacyResponse);

}

public override Task WriteToStreamAsync(Type type, object value, Stream writeStream, HttpContent content,

TransportContext transportContext)

{

return Task.Factory.StartNew(() => WriteResponseToStream((LegacyResponse) value, writeStream));

}

public void WriteResponseToStream(LegacyResponse legacyResponse, Stream writeStream)

{

var request = legacyResponse.Request;

var body = request.GetSoapBody();

var operationElement = \_legacyMessageParser.GetOperationElement(body);

var operationElementName = operationElement.Name;

var namespaceName = operationElementName.NamespaceName;

var operationName = operationElementName.LocalName;

var operationResultInnerElement = new XElement(

string.Concat("{", namespaceName, "}", operationName, "Result"));

var processResult = legacyResponse.ProcessingResult;

if (processResult != null)

{

operationResultInnerElement.Add(processResult);

}

var operationResultOuterElement = new XElement(

string.Concat("{", namespaceName, "}", operationName, "Response"));

operationResultOuterElement.Add(operationResultInnerElement);

operationElement.ReplaceWith(operationResultOuterElement);

using (var outWriter = new XmlTextWriter(writeStream, Encoding.UTF8))

{

request.WriteTo(outWriter);

outWriter.Flush();

}

}

}

}

Rather than going into all of the details of how the SOAP message body is being constructed, we will instead point out the following highlights:

* The formatter derives from the ASP.NET Web API's MediaTypeFormatter class. If you need to handle serializing and deserializing strongly-typed objects, you'll want to derive from this class.
* The supported media types should be specified in the constructor. SOAP is XML, so in this case we only specify XML as a supported media type.
* The CanReadType and CanWriteType methods are interrogated by ASP.NET Web API when it needs to deserialize or serialize data in a request message or response message, respectively. This formatter is only used to write LegacyResponse objects, so we implemented the methods to reflect this.

Finally, we needed to configure the formatter to make ASP.NET Web API aware of it. We configured it by adding the ConfigureFormatters method to the WebApiApplication class and invoking it from the Application\_Start method, as follows:

namespace WebApi2Book.Web.Api

{

public class WebApiApplication : HttpApplication

{

protected void Application\_Start()

{

GlobalConfiguration.Configure(WebApiConfig.Register);

ConfigureFormatters();

RegisterHandlers();

new AutoMapperConfigurator().Configure(WebContainerManager.GetAll<IAutoMapperTypeConfigurator>());

}

private void ConfigureFormatters()

{

var legacyFormatter = (MediaTypeFormatter)WebContainerManager.Get<ILegacyMessageTypeFormatter>();

GlobalConfiguration.Configuration.Formatters.Insert(0, legacyFormatter);

}

…

Note that we inserted as the first formatter in the list. This was to ensure that our specialized formatter was picked first at run time by ASP.NET Web API, instead of some other, general-purpose XML formatter (e.g., the default XML formatter).

This concludes our brief tour of how to support SOAP-based clients, so now it's demo time! We saw the request earlier in this section. Here's the corresponding response:

Get Tasks Response - SOAP (abbreviated)

HTTP/1.1 200 OK

Content-Type: text/xml

<?xml version="1.0" encoding="utf-8"?><s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"><s:Body><GetTasksResponse xmlns="http://tempuri.org/"><GetTasksResult><Task><TaskId>1</TaskId><Subject>Triage the important tasks</Subject><StartDate>2014-04-24T15:20:58</StartDate><DueDate d6p1:nil="true" xmlns:d6p1="http://www.w3.org/2001/XMLSchema-instance" /><CreatedDate>2014-04-24T11:19:41</CreatedDate><CompletedDate d6p1:nil="true" xmlns:d6p1="http://www.w3.org/2001/XMLSchema-instance" /><Status><StatusId>2</StatusId><Name>In Progress</Name><Ordinal>1</Ordinal></Status><Links><Link><Rel>self</Rel><Href>http://localhost:61589/api//tasks/1</Href><Method>GET</Method></Link></Links></Task>…

Notice how different it is from the Get Tasks Response - With Accept Header example from the Content Negotiation section. This custom-formatted response is encapsulated inside of a SOAP body, which is enclosed inside of a SOAP envelope and ready to be consumed by the existing legacy clients… all without requiring any modification to the client code. Requirements 2 and 3 are hereby satisfied.

# Summary

In this chapter we explored how to use ASP.NET Web API to simulteneously support legacy SOAP clients and extend reach to clients requiring various message formats. Over the course of our journey we were also introduced to some important ASP.NET Web API features that aren't necessarily tied to supporting legacy clients, including:

* Content Negotiation
* A message handler that takes the place of a controller
* Route-specific message handlers
* Mixing convention-based routing with attribute-based routing
* A custom formatter

Though we broke from our usual exhaustive explanation of the implementation and instead provided more of an overview, all of the source code is available on our project website for those interested in getting into the details of manipulating SOAP messages.

In the next chapter we'll tie up some loose ends, so to speak, by showing you how to consume the task-management service using a Single Page Application (SPA). We'll also demonstrate how to test the functionality we've developed. Don't worry, ASP.NET Web API, and our loosely-couple architecture, will actually help make testing enjoyable!