ServiceStack Succinctly

by Zoran Maksimovic

Syncfusion | Technology Resource Portal

ServiceStack Succinctly

By
Zoran Maksimovic

Foreword by Daniel Jebaraj

Syncfusion
Deliver innovation with ease

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

Technical Reviewer: Ricardo Peres
Copy Editor: Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Zoran Maksimovic is a solution architect and software developer with more than 14 years of professional experience. He is passionate about programming and web platforms, especially on Microsoft technologies. He has spent most of his professional life working as a consultant on various projects for clients based in Switzerland, Italy, Germany, and France.

Zoran is currently based in Zürich, Switzerland and works for Credit Suisse as a Technical Lead on a project in the FX Structured Products and Derivatives area.

He also contributes to his personal blog www.agile-code.com and is active on Twitter as @zoranmax.

When not coding or writing, Zoran enjoys playing guitar, baroque music, good food, Italian wine, and spending time with his family. He is married and is the father of a little lady named Sofia.

Introduction

Microsoft's ASMX, ASP.NET Web API, and WCF are very solid frameworks for building web services. However, there are a number of open-source alternatives that can be used and ServiceStack is one of them.

ServiceStack is an open-source[1] framework that complements, extends, and substitutes the aforementioned built-in Microsoft .NET technologies.

Purpose of the Book

The purpose of this book is to make you aware of the ServiceStack framework and how it can be used to implement high-quality software with minimum effort.

My hope is that, after reading this book, you will understand the framework and use it in your production systems.

Target Audience

This book is intended for software developers with a good understanding of the Microsoft .NET framework (as all examples are written in C#) and web technologies in general.

A basic understanding of Hypertext Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP), Representational State Transfer (REST), JavaScript Object Notation (JSON), and Extensible Markup Language (XML) is required.

Additional Information and Resources

Additional information about ServiceStack can be found directly on the ServiceStack website at www.servicestack.net.

If you want to know more about the technologies mentioned in this book, check out the following websites:

- HTTP: http://www.ietf.org/rfc/rfc2616.txt
- REST: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[1] At the time of this writing, as announced by ServiceStack owner Demis Bellot: "To ensure its long-term sustainability, ServiceStack will transition to an annually-supported paid product similar to Xamarin's products. Starting from the next major v4.0 release of ServiceStack.* on NuGet, which will be the first commercial-only, binary distribution (with an exception for OSS projects and a "free-tier" for small projects)."

- JSON: http://json.org
- XML: http://www.w3.org/TR/xml
- SOAP: http://www.w3.org/TR/soap12-part1

Source Code

At the time of this writing, ServiceStack's source code is open source and is hosted on GitHub at https://github.com/ServiceStack/ServiceStack.

ServiceStack Groups and Communities

There are several groups on the web where additional information is hosted and shared, and most questions answered. The following table contains the most important ones.

Table 1: ServiceStack additional information

Twitter	https://twitter.com/servicestack
Google Plus	https://plus.google.com/u/0/communities/112445368900682590445
Google Groups	http://groups.google.com/group/servicestack
JabbR	https://jabbr.net/#/rooms/servicestack
StackOverflow	http://stackoverflow.com/questions/tagged/servicestack

Software Requirements

To get the most out of this book and the included examples, you will need to have a version of the Visual Studio IDE installed on your computer and Microsoft .NET 4. At the time of this writing, the latest available stable edition of Visual Studio Express is Visual Studio 2013. You can download Visual Studio Express 2013 for free directly from Microsoft's website at http://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx.

Conventions Used in the Book

There are specific formats you will see throughout this book to illustrate tips and tricks or other important concepts.

Note: This will identify things to note throughout the book.

Tip: This will identify tips and tricks throughout the book.

Source Code

All examples in this book are written in C#. The source code will always be shown in the following format.

```
[Route("/movies/{Id}", "GET")]
public class GetMovieRequest
{
    public string Id { get; set; }
}
```

Microsoft .NET types or Uniform Resource Identifiers (URIs) that are mentioned in the text will be shown using a monospace font: **`JsonSerializer`**.

Data will be usually displayed in the XML format simply because I believe that XML provides a very readable format that is easy to understand.

Resources

Code mentioned in this book can be checked out from the Bitbucket repository at https://bitbucket.org/syncfusiontech/servicestack-succinctly.

ServiceStack Version

All of the examples and explanations apply to version 3.9.59 of ServiceStack, which is the latest stable version at the time of this writing.

Chapter 1 ServiceStack Overview

What is ServiceStack?

ServiceStack is an open-source, Microsoft .NET and Mono REST web services framework and an alternative to the WCF, ASP.NET MVC, and ASP.NET Web API frameworks. ServiceStack is particularly suitable for the development of REST web services.

Authors of ServiceStack define it as:

> *"Thoughtfully architected, obscenely fast, thoroughly enjoyable web services for all."*

And I agree.

In a nutshell, ServiceStack supports the following features:

- REST and SOAP endpoints.
- Autoconfiguration of data formats including XML, JSON, HTML, CSV, and JSV.
- Plain-old CLR objects (POCO) as input and output objects.
- Validation and smart fluent syntax.
- Inversion of Control (IoC) container.
- Object-Relational Mapping (ORM).
- Caching mechanism (Memcached and Redis supported).
- Logging framework.
- Self-contained—no external libraries needed.

We will take a look at most of these features in the following chapters.

Why Should You Consider Using ServiceStack?

If you are not already impressed with the list of features that ServiceStack supports, the following list contains a few points on why you should use ServiceStack in your next project:

- **Extremely fast**: Based on the benchmarks at http://www.servicestack.net/benchmarks, ServiceStack excels when it comes to the real-world speed of object (de)serialization.
- **Simplicity**: Defining endpoints, hosting, routing, and configuration are simpler compared to WCF or ASP.NET Web API.
- **Coherence**: It follows the same philosophy across different styles of services, REST or SOAP.
- **Clean configuration**: No XML configuration files and no code-generated proxies.

ServiceStack Components

ServiceStack is made up of a number of independent modules (see Figure 1) which can be used separately in projects without using the ServiceStack framework itself.

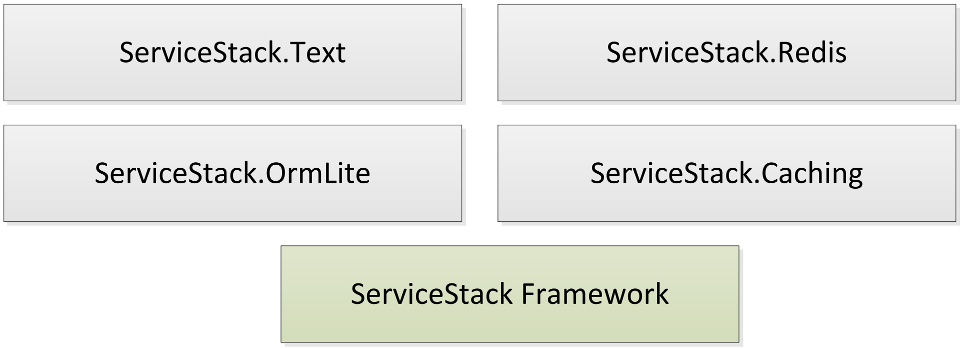


Figure 1: ServiceStack components

ServiceStack.Text

`ServiceStack.Text` is an independent, dependency-free serialization library that is used by ServiceStack internally to do any text processing. **`ServiceStack.Text`** contains a large number of features, and some of the most important ones are:

- JSON serialization and deserialization through **`JsonSerializer`**.
- JSV format serialization and deserialization through **`TypeSerializer`**.
- CSV format serialization and deserialization through **`CsvSerializer`**.
- **`StringExtensions`** for XML, JSON, CSV, and URL encoding, **`BaseConvert`**, Rot13, Hex escape, etc.
- Supports custom builds for .NET 3.5+, Mono, MonoTouch and MonoDroid, Silverlight 4 and 5, Xbox, and Windows Phone 7.

If you were to use **`ServiceStack.Text`** in your project and needed the JSON serializer, the following code example shows how to do so.

```
public static void JsonSerializationExample()
{
    var person = new {LastName = "Doe", Name = "John", Age = 36};

    var personJson = JsonSerializer.SerializeToString(person);

    Console.WriteLine(personJson);
}

// Output produced (JSON Format):
// {"LastName":"Doe","Name":"John","Age":36}
```

TypeSerializer

`TypeSerializer` is one of the fastest and most compact text serializers available for .NET. Out of all the serializers benchmarked, it is the only one to remain competitive with protobuf-net's very fast implementation of Protocol Buffers, Google's high-speed binary protocol.[2]

```
public static void TypeSerializerExample()
{
    var person = new { LastName = "Doe", Name = "John", Age = 36 };

    var personJsv = TypeSerializer.SerializeToString(person);

    Console.WriteLine(personJsv);
}

// Output produced (JSV Format):
// {LastName:Doe,Name:John,Age:36}
```

`TypeSerializer` uses a hybrid CSV and JavaScript-like, text-based format that is optimized for both size and speed. Authors have called this format JSV, which is a combination of JSON and CSV.

Source code for **`TypeSerializer`** can be downloaded at https://github.com/ServiceStack/ServiceStack.Text, or the package can be directly installed by using NuGet with the following command:

```
PM> Install-Package ServiceStack.Text
```

ServiceStack.Redis

`ServiceStack.Redis` is an open-source client for the Redis (http://www.redis.io) in-memory database. **`ServiceStack.Redis`** simplifies interfacing with Redis significantly and, as mentioned earlier, is built on ServiceStack but can be used separately.

Note: Redis is an open source, Berkeley Software Distribution (BSD) licensed, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets. In order to achieve its outstanding performance, Redis works with an in-memory data set that can be persisted to disk once in a while or by appending each command to a log.

[2] TypeSerializer benchmark: http://mono.servicestack.net/benchmarks/NorthwindDatabaseRowsSerialization.1000000-times.2010-02-06.html

Source code for Redis can be downloaded from https://github.com/ServiceStack/ServiceStack.Redis, or the package can be directly installed by using NuGet with the following command:

```
PM> Install-Package ServiceStack.Redis
```

ServiceStack.OrmLite

ServiceStack includes its own ORM library, which is a convention-based, configuration-free, lightweight ORM that uses standard plain-old CLR object (POCO) classes and data annotation attributes to infer its table schema.

`ServiceStack.OrmLite` currently supports several relational databases: Microsoft SQL Server, MySQL, PostgreSQL, Oracle, Firebird, SQLite32, SQLite64, and SQLite.Mono. **`ServiceStack.OrmLite`** is compatible with both Microsoft .NET and Mono.

Source code for **`ServiceStack.OrmLite`** can be downloaded from https://github.com/ServiceStack/ServiceStack.OrmLite. Alternatively, packages can be directly installed by using NuGet with the following commands.

Microsoft SQL Server provider:

```
PM> Install-Package ServiceStack.OrmLite.SqlServer
```

Oracle provider:

```
PM> Install-Package ServiceStack.OrmLite.Oracle
```

ServiceStack.Caching

ServiceStack provides **ICacheClient**, a unified caching interface, for a number of different cache providers:

- **In Memory:** Uses RAM as the caching mechanism.
- **Redis**: An open-source, BSD-licensed, advanced key-value store.
- **Memcached**: An interface to Memcached, which is a high-performance, distributed memory, object-caching system intended for use in speeding up dynamic web applications by alleviating database load.
- **Azure Client**: Used to interface with Microsoft Azure AppFabric Caching.
- **AWS Cache Client**: Used to interface with Amazon's DynamoDB back-end hosted on Amazon Web Services.
- **Disk**: Writes on the hard disk.

ServiceStack Philosophy

ServiceStack is influenced by Martin Fowler's Data Transfer Object pattern[3] (DTO) and promotes **message based communication**. This pattern simplifies manipulation of request and response data, and enables decoupling of message structures from domain layer entities.

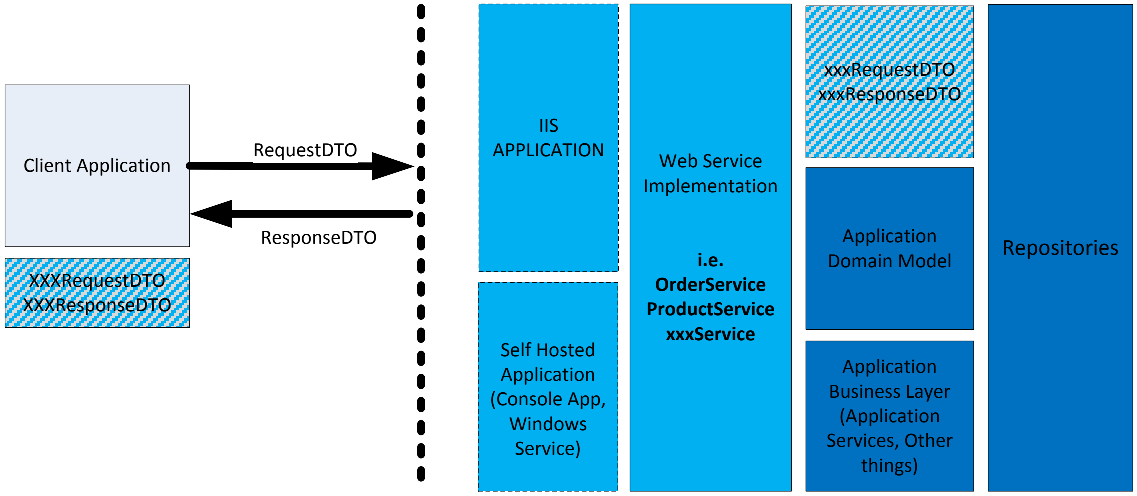


Figure 2: Data transfer objects

Exposing DTOs, and not the application domain model, gives the freedom to refactor internal implementation of the service without breaking external clients and keeps a clean interface.

By definition, a ServiceStack web service will have, at minimum, the following:

Table 2: Basic web service components

Class	Description
Request DTO object	The input of a service method. It represents the **action** to be performed. Usually the name of the class contains a **verb** (e.g., **GetOrderRequest**, **DeleteItem**).

[3] For more information about the DTO pattern, please read Martin Fowler's article: http://martinfowler.com/eaaCatalog/dataTransferObject.html.

Class	Description
Service	Implements the internal logic and acts as a “controller”. Usually has some or all of the following HTTP verbs implemented: **GET**, **POST**, **PUT**, **DELETE**, **PATCH**, **OPTIONS**, **HEAD**, or **Any()**,which represents all of them.
Response DTO object	Represents the result of an action. Usually the data returned should be named with a **noun** (e.g., **MoviesResponse**, **Orders**, **ProductResponse**).

Request and Response Pipeline

ServiceStack is built on top of the ASP.NET **`System.Web.IHttpHandler`** interface. Fortunately, the new implementation has reduced complexity (compared to WCF configuration) and introduces POCO objects in almost every aspect of the framework itself.

ServiceStack implements a simple framework with which to work and enables a great deal of extensibility to the basic functionality. For instance, you can create **Request and Response filters**, which can be global or only applied to a service, or you can extend the formatters.

The following figure shows the Request and Response pipeline.

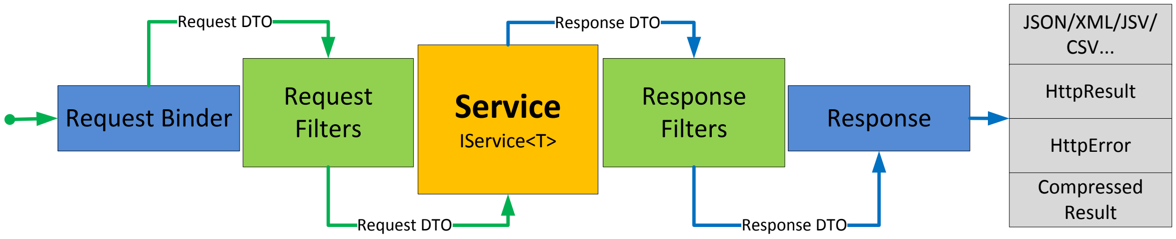


Figure 3: Request and Response pipeline

Hypermedia as the Engine of Application State (HATEOAS)

REST services, to be considered complete, should implement the HATEOAS[4] constraint. The goal of HATEOAS is that the clients interact with the service entirely through hypermedia (i.e. links) that are provided dynamically (at run time) by the service itself. In a perfect world, the client should not need any prior knowledge about how to interact with the service; the only thing it should need to know is how to follow the links (or, in other words, to understand the hypermedia itself).

Consequently, this principle makes the client and the application loosely coupled, which makes it very different from SOAP-based services where the communication between the client and the server is made through some fixed interfaces.

Certainly this book cannot cover the full HATEOAS implementation, but as we are going to see, we will expose some basic information to enable the hypermedia constraint.

[4] More information about HATEOAS can be found on Wikipedia at http://en.wikipedia.org/wiki/HATEOAS.

Chapter 2 ServiceStack Basics

In this chapter, we will go through various concepts and background information about ServiceStack including routing, dependency injection, content negotiation, and validation and error handling.

Application Host

The base entry point for ServiceStack is the **AppHost** (application host). The concept of the application host is to define a central point for configuring and wiring the request to the service. There can be only one application host per application.

In general, a ServiceStack service can be hosted on Internet Information Services (IIS) as a console application, Windows Service, or mixed with an ASP.NET or ASP.NET MVC application.

Web applications will implement the **AppHostBase** class while the console and Windows Service applications will implement **AppHostHttpListenerBase**.

In a web application, the application host has to be started only once when the application starts. This can be done in the **Global.asax.cs** file by calling the **Init** method in the **Application_Start** method. The following code shows the minimum that has to be configured.

```
public class Global : HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        new ServiceAppHost().Init();
    }
}

public class ServiceAppHost : AppHostBase
{
    public ServiceAppHost()
        : base("Order Management", typeof (ServiceAppHost).Assembly)
    {
    }

    public override void Configure(Container container)
    {
    }
}
```

To configure a console application, the style is practically the same. There is no difference apart from inheriting from a different base class and starting the application in the **`Main()`** method rather than in the **`Global.asax.cs`** file.

```
public class ServiceAppHost : AppHostHttpListenerBase
{
    public ServiceAppHost() :
        base("Order Management System", typeof (ServiceAppHost).Assembly)
    {
    }

    public override void Configure(Container container)
    {
    }
}

static void Main()
{
    ServiceAppHost  appHost = new ServiceAppHost();
    appHost.Init();
    appHost.Start("http://*:80/");
    Console.Read();
}
```

Service

Historically, there are several ways in which the web service can be created. Interfaces used in earlier versions of ServiceStack used **`IRestService`** and **`IService<T>`**. However, currently the recommended way is to use **`ServiceStack.ServiceInterface.Service`**, which is the approach used in this book.

```
public class OrderService : ServiceStack.ServiceInterface.Service { }
```

As shown in the following code, the **`Service`** base class implements and exposes several very useful methods and properties such as **`Request`** and **`Response`**, which are the two most used objects in the service and give a great deal of manipulation and inspection possibilities.

```
public class Service : IService, IRequiresRequestContext, IServiceBase, IResolver,
IDisposable
{
    public virtual IResolver GetResolver();
    public virtual IAppHost GetAppHost();
    public virtual Service SetResolver(IResolver resolver);
    public virtual T TryResolve<T>();
    public virtual T ResolveService<T>();
    protected virtual TUserSession SessionAs<TUserSession>();
    public virtual void PublishMessage<T>(T message);
    public IRequestContext RequestContext { get; set; }
    protected virtual IHttpRequest Request { get; }
    protected virtual IHttpResponse Response { get; }
    public virtual ICacheClient Cache { get; }
    public virtual IDbConnection Db { get; }
    public virtual IRedisClient Redis { get; }
    public virtual IMessageProducer MessageProducer { get; }
    public virtual ISessionFactory SessionFactory { get; }
    public virtual ISession Session { get; }
}
```

Request

`Request` implements the .NET built-in **`IHttpRequest`** interface and therefore exposes all that we might need to know about the current request. Among the other things to note is the ability to inspect **`Headers`**, **`QueryString`**, and the current **`HttpMethod`**.

Imagine the following HTTP request.

```
GET http://<servername>/orders?page=1
Accept: application/json
```

In the service, we can easily retrieve all of the information.

```
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public object Get(GetOrdersRequest request)
    {
        //value = 1
        var pageQueryStringValue = this.Request.QueryString["page"];
        //value = application/json
        var acceptHeaderValue = this.Request.Headers["Accept"];
        //value = GET
        var httpMethod = this.Request.HttpMethod;
        return null;
    }
}
```

Response

`Response` represents the object that the client will eventually receive. Among others, one of the most useful methods exposed by the **`Response`** object is the **`AddHeader`** method, which manipulates the headers returned to the client. Let's look at an example of the **`AddHeader`** usage.

```
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public object Get(GetOrdersRequest request)
    {
        this.Response.AddHeader("Location", "http://<servername>/orders");
        return new HttpResult {StatusCode = HttpStatusCode.OK};
    }
}
```

The following code shows the **`Result`** returned to the client.

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: application/json
Location: http://<servername>/orders
Date: Sun, 28 Jul 2013 22:45:42 GMT
```

Web Service Method Return Types

A web service method can return one of the following:

- Response DTO object serialized to the response type (JSON, XML, PNG).
- Any basic .NET value.
- **`HttpResult`**: Used whenever full control of what client recieves is needed.
- **`HttpError`**: Used to return the error message to the client.
- **`CompressedResult`** (**`IHttpResult`**) for a customized HTTP response.

The following two methods both produce the same result.

```
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public List<GetOrderResponse> Get(GetOrdersRequest request)
    {
        return new List<GetOrderResponse>();
    }

    public HttpResult Get(GetOrdersRequest request)
    {
        return new HttpResult(response: new List<GetOrderResponse>(),
                              contentType: "application/json",
                              statusCode: HttpStatusCode.OK);
    }
}
```

REST HTTP Methods

The ServiceStack framework virtually supports all of the available HTTP methods. The web service method to be executed will be determined at run time by combining the routes and requested HTTP method. ServiceStack will execute a method of the service that corresponds to the actual HTTP method name. The HTTP verb **`GET`** will be executed by **`Get()`**, **`POST`** will be executed by **`Post()`**, and so on.

The following table contains some reminders about the basic HTTP verbs, what they do, and when they are used.

Table 3: HTTP verbs

`GET`	• Retrieves a resource. • It is safe (guaranteed not to cause side effects), idempotent,[5] and cacheable. • Should never change the state of a resource.
`POST`	• Creates a new resource. • Unsafe; the effect of this verb is not specifically defined by HTTP. • Not idempotent.
`PUT`	• Updates an existing resource. • Can be used to create a new resource when client knows the URI. • Can be called *n*-number of times and always produces the same effect (idempotent).
`DELETE`	• Removes an existing resource. • Can be called *n*-number of times and always produces the same effect (idempotent).
`PATCH`	• Not safe, not idempotent. • Like **`PUT`**, but allows full and partial updates of a resource.

The service can have the same HTTP verb implemented multiple times, but the input parameters should be different as with any other normal method overload.

```
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public object Get     (SomeRequest request) {...}
    public object Get     (SomeRequest2 request) {...}

    public object Post    (SomePostRequest request) {...}
    public object Post    (SomePostRequest2 request) {...}
}
```

[5] Safe and idempotent definitions: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Content Negotiation

If not specified otherwise, ServiceStack offers mainly three ways of negotiating the content type of the resource:

- The HTTP Accept header.
- The **`format`** query string parameter.
- The file extension (e.g., http://<servername>/orders**.json**).

Use of the HTTP **`Accept`** header is considered by many to be a recommended and more elegant way of negotioating the content type, and it is the HTTP standard. However, it's a hot topic and there is debate about whether to pass the format instructions directly in the URI. I think that it is good to have them both, as both have their pros and cons.

Since you should already be familiar with how to use the HTTP **`Accept`**[6] header, we won't go into details. The following table shows the options available in ServiceStack.

Table 4: Content negotiation in ServiceStack

Query String Style	File Extension Style	Accept Header
`…/orders?format=json`	**`…/orders.json`**	Accept: application/json
`…/orders?format=xml`	**`…/orders.xml`**	Accept: application/xml
`…/orders?format=jsv`	**`…/orders.jsv`**	Accept: application/jsv
`…/orders?format=csv`	**`…/orders.csv`**	Accept: application/csv

Tip: It's possible to disable the file extension style by setting `Config.AllowRouteContentTypeExtensions = false` in the AppHost instance.

There are different ways of defining the response content type:

- Forcing the content type for every request to **`JSON`** by configuring the application host.

```
public class ServiceAppHost : AppHostBase
{
    public ServiceAppHost()
```

[6] Accept header specification: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

```
        : base("Order Management", typeof(ServiceAppHost).Assembly)
    {
        base.Config.DefaultContentType = ContentType.Json;
    }
}
```

- Specifying the content type at the service method’s level, either by using an **`AddHeader`** filter or by specifying the **`ContentType`**.

```
[AddHeader(ContentType = ContentType.Json)]
public object Get(GetOrdersRequest request) { /*..*/}
```

Or, alternatively:

```
public object Get(GetOrdersRequest request)
{
    base.Response.ContentType = ContentType.Json;
    return /*..*/
}
```

- Returning a decorated response via **`HttpResult`**.

```
public object Get(GetOrdersRequest request)
{
    return new HttpResult(responseDTO, ContentType.Json);
}
```

Routing

Routing is the process of selecting paths along which to send a request. To determine which action to perform for a request, ServiceStack has to keep the list of routes and this has to be instructed specifically when the application starts. There are several ways in which the route can be registered:

- Using a default route.
- Creating a custom route by using **`RouteAttribute`** or Fluent API.
- Dynamic paths.
- Autoregistered paths.

Default Routes

By default, for every Request DTO, ServiceStack will create a default route in the following form:

```
/api?/[xml|json|html|jsv|csv]/[reply|oneway]/[servicename]
```

Let’s suppose we want to support a custom route, http://<servername>/orders, and we expose a Request DTO called **`GetOrders`**. In this case, without specifying any route, ServiceStack will create http://<servername>/xml/reply/GetOrders automatically.

Custom Routes

A route can be declared as a class attribute directly at the Request DTO level or in the **`AppHostBase`** by using the Fluent API. The route has an option to define one or more HTTP verbs.

Route Attribute

By using the **`RouteAttribute`**, a route can be declared directly at the Request DTO object.

```
[Route("/orders", "GET POST", Notes="…", Summary="…")]
public class GetOrders { }
```

Fluent API

Instead of using the **`RouteAttribute`**, the same can be achieved by defining the route in the application host declaration.

```
public class ServiceAppHost : AppHostBase
{
    public ServiceAppHost()
        : base("Order Management", typeof(ServiceAppHost).Assembly)
    {
        Routes
            .Add<GetOrders>("/orders", "GET")
            .Add<CreateOrder>("/orders", "POST")
            .Add<GetOrder>("/orders/{Id}", "GET")
            .Add<UpdateOrder>("/orders/{Id}", "PUT")
            .Add<DeleteOrder>("/orders/{Id}", "DELETE")
    }
}
```

Dynamic Paths

ServiceStack's routing mechanism offers a way to dynamically bind the parameters sent in the URL and to deserialize the object once it reaches the service. In the following code example, we can see that in the route there is an **`{Id}`** declaration. At run time, the value sent as **`{Id}`** will be deserialized as the **`GetOrder.Id`** property.

```
[Route("/orders/{Id}", "GET")]
public class GetOrder
{
    public string Id { get; set; }
}
```

Autoregistered Paths

By using the **`Routes.AddFromAssembly(typeof(OrderService).Assembly)`** method, we can automatically map and define all routes.

```
using ServiceStack.ServiceInterface;

//Request DTO (notice there is no route defined as the Attribute!)
public class GetOrderRequest
{
    public string Id { get; set; }
}

// Service
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public List<OrderResponse> Get(GetOrdersRequest request) { … }
    public object Post(CreateOrder request) { … }
}

// Autoregistering the routes in the application host.
public class  ServiceAppHost : AppHostBase
{
    public  ServiceAppHost (): base("Order Management",
        typeof(OrderService).Assembly)
    {
        Routes.AddFromAssembly(typeof (OrderService).Assembly);
    }
}
```

By using reflection, ServiceStack will check the **`OrderService`**, determine the request's parameter type, and autogenerate the route.

For the previous code example, the route generated will be **`/GetOrders`** (**`GET`**) and **`/CreateOrder`** (**`PUT`**).

Route Wildcards

There is a way to define a wildcard in the route; this is especially useful when the route becomes too complex.

The following is an example with a route that uses a wildcard.

```
Routes.Add("/orders/{CreationDate}//{Others*}", "GET");
```

In this case, the request would be **`/orders/2012-11-12/SomeOther/InfoGoes/Here`**.

This will be translated and deserialized as follows.

`CreationDate = 2012-11-12; Others = “SomeOther/InfoGoes/Here”`

So, the **`Others`** keyword will be taken as it is, and it can then be further processed in the application code.

Where to Place the Route Declaration

My personal preference is to use the **`AppHost`** as the only place where all the paths should be declared. While there is nothing wrong with using the the **`RouteAttribute`**, I like to have all of the routes declared together in one place.

If the Request and Response DTOs POCO objects are placed in one assembly, it can be distributed without any external dependency to the clients. If using the **`RouteAttribute`** in this case, it would mean that ServiceStack libraries would need to be included as a dependency and shipped together to the client.

IoC Container

ServiceStack has built-in support for dependency injection. To achieve this, it uses a slightly modified version of the **Funq**[7] framework. Funq is fast and easy to use. ServiceStack has enhanced the basic version of Funq with lifetime scope of the injected objects and therefore supports:

- **ReuseScope.Default**: Default scope which is equivalent to ReuseScope.Hierarchy.
- **ReuseScope.Hierarchy**: Similar to Singleton Scope. Instances are reused within a container hierarchy. Instances are created (if necessary) in the container where the registration was performed and are reused by all descendent containers.
- **ReuseScope.Container**: Singleton scope (an instance is used per application lifetime).
- **ReuseScope.Request**: Request scope (an instance is used per request lifetime).

[7] Funq website: https://funq.codeplex.com

- **ReuseScope.None**: Transient scope (a new instance is created every time).

All of the configuration can be done directly in the application host and declaring the objects is not very different than in any other framework.

```
public class  ServiceAppHost : AppHostBase
{
   public override void Configure(Container container)
    {
        container.Register<IOrderRepository>(new OrderRepository());
        container.Register<IProductMapper>(x => new ProductMapper ())
                              .ReusedWithin(ReuseScope.Container);
    }
}
```

References will be **automatically** injected if the service exposes a public property or if it has a constructor, with the parameters being one of the registered IoC types.

```
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public IOrderRepository OrderRepository { get; set; }

    public OrderService(IProductRepository productRepository){ /*..*/}
}
```

Custom Containers

ServiceStack enables the integration of third-party IoC frameworks through the **`IContainerAdapter`** interface. This only exposes two methods.

```
public interface IContainerAdapter
{
    T Resolve<T>();
    T TryResolve<T>();
}
```

There are several implementations already available for Microsoft Unity[8], Ninject[9], StructureMap[10], Castle Windsor,[11] and Autofac[12].

Microsoft Unity Container Adapter installation package:

[8] Microsoft Unity IoC adapter: www.agile-code.com/blog/servicestack-ioc-with-microsoft-unity
[9] Ninject is available at: www.ninject.org
[10] StructureMap website: docs.structuremap.net
[11] Castle Windsor website: www.castleproject.org
[12] Autofac website: http://autofac.org

```
PM> Install-Package ServiceStack.ContainerAdapter.Unity
```

Ninject Container Adapter installation package:

```
PM> Install-Package ServiceStack.ContainerAdapter.Ninject
```

Validation

ServiceStack has a validation framework built around the Fluent Validation library.[13]

When the validation is enabled, the validation framework will check the predefined rules before the service method gets invoked. In the case of a validation error, the **`ErrorResponse`** object will be returned with the error details (as shown in Figure 4). It's good to know that the validation is performed on the server side.

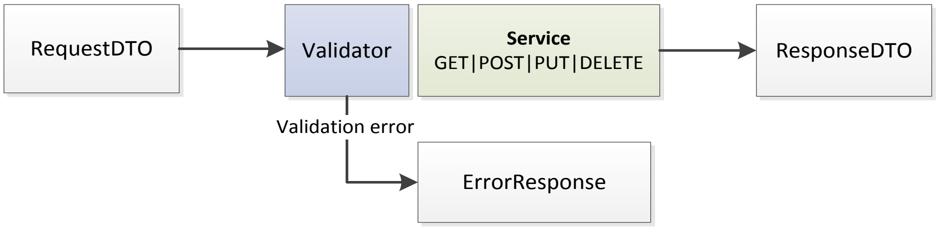


Figure 4: Validation

To enable the validation, only two operations are needed:

- Create the validator implementation class: In our case, create the **`GetOrderValidator`** class. This class in particular will only be responsible for validating the **`GetOrdersRequest`** Request DTO object. The **`Validator`** class has to inherit from the **`AbstractValidator`** class and specify the Request DTO as the generic parameter type.
- Register the **`ValidationFeature`** and the validator implementation in the application host. This will enable the validation framework and register the particular validator class, in our case **`GetOrderValidator`**. Registration is done by using the **`Plugins.Add()`** method.

[13] Fluent Validator Library: https://fluentvalidation.codeplex.com

In the following code example, the validator would only be used in the case of GET or POST methods, and this is defined with the **RuleSet** method. **RuleFor** instead specifies the rule to be applied for a given DTO property. As you will see, a customized error message can be specified.

```
public class GetOrderValidator : AbstractValidator<GetOrdersRequest>
{
    public GetOrderValidator()
    {
        //Validation rules for GET request.
        RuleSet(ApplyTo.Get | ApplyTo.Post, () =>
            {
                RuleFor(x => x.Id)
                   .GreaterThan(2)
                   .WithMessage("OrderID has to be greater than 2");
            });
    }
}

public class ServiceAppHost : AppHostBase
{
    public ServiceAppHost()
         : base("Order Management", typeof(ServiceAppHost).Assembly)
{
    //Enabling the validation.
    Plugins.Add(new ValidationFeature());
    Container.RegisterValidator(typeof(GetOrderValidator));
}
```

In case of an invalid request, such as **GET/orders/1**, the framework will return the following error (in cases where the format specified is XML).

```
<ErrorResponse>
  <ResponseStatus>
    <ErrorCode>GreaterThan</ErrorCode>
    <Message>OrderID has to be greater than 2</Message>
    <StackTrace i:nil="true" />
    <Errors>
      <ResponseError>
        <ErrorCode>GreaterThan</ErrorCode>
        <FieldName>Id</FieldName>
        <Message>'Id' must be greater than '2'.</Message>
      </ResponseError>
    </Errors>
  </ResponseStatus>
</ErrorResponse>
```

Client Tools

Since ServiceStack exposes standard RESTful web services which are based on pure HTTP, any HTTP-capable client is able to access and consume it. It doesn’t matter which programming language or framework is used; the important thing is the ability to enable communication by using HTTP.

There are several .NET clients currently available: RestSharp,[14] which is an open-source implementation, and several built-in Microsoft .NET ones like HttpClient,[15] WebClient,[16] and HttpWebRequest.[17]

The ServiceStack framework, however, provides its own client implementations that are highly optimized for ServiceStack (e.g., exception handling and routing). There are different implementations of the client, which can be a generic C# client, Silverlight client, JavaScript client, Dart[18] client, or MQ client.[19]

Clients are optimized for the content type. So there is a **`JsonServiceClient`**, **`XmlServiceClient`**, **`JsvServiceClient`**, and two SOAP clients, **`Soap12ServiceClient`** and **`Soap11ServiceClient`**, for the two current SOAP versions. The difference between the clients is the serializer/deserializer being used. As the following example shows, using a ServiceStack client is relatively easy.

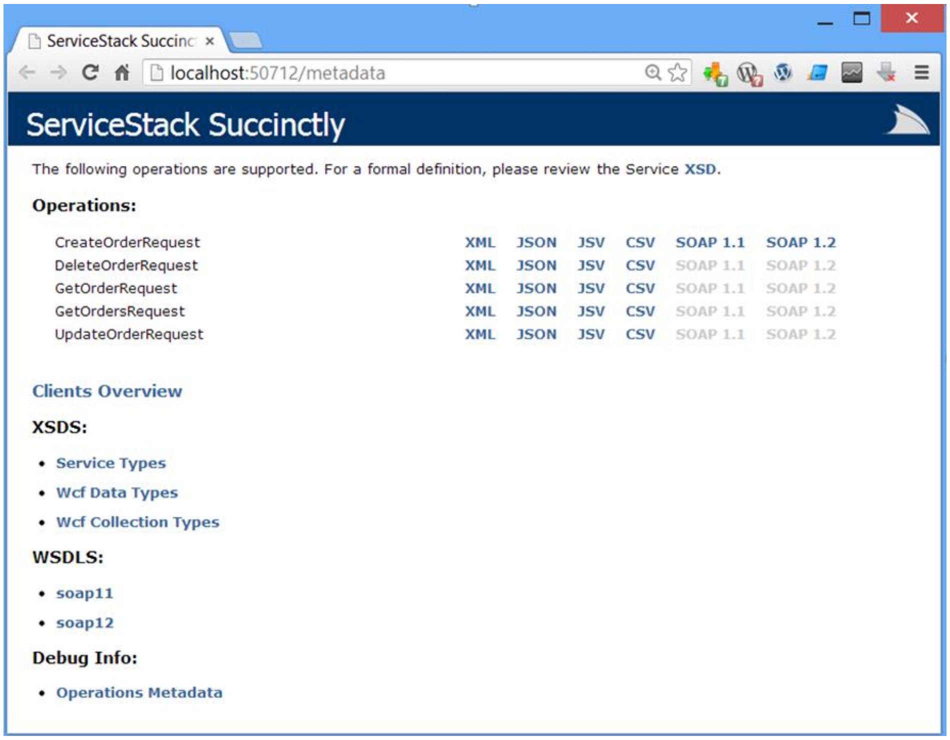
```
JsonServiceClient client = new JsonServiceClient("http://localhost:50712/");
OrderResponse order = client.Get<OrderResponse>("/orders/1");
```

Metadata Page

By default, when accessing the web application, ServiceStack exposes a **`metadata`** page. The **`metadata`** page contains information about the operations, available content types, Web Services Description Language (WSDL), and other objects exposed in the current application host. It is extremely useful because it acts as documentation for the available operations and used types (see Figure 5).

For every format (XML, JSON, etc.) there is an example of an input and output object. This means that we can see what the service accepts as the Request and Response DTOs.

[14] RestSharp: http://restsharp.org
[15] HttpClient: http://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx
[16] WebClient: http://msdn.microsoft.com/en-us/library/system.net.webclient.aspx
[17] HttpWebRequest: http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.aspx
[18] Dart Client: https://github.com/ServiceStack/ServiceStack/wiki/Dart-Client
[19] MQ Clients: https://github.com/ServiceStack/ServiceStack/wiki/Messaging



ServiceStack Succinctly

The following operations are supported. For a formal definition, please review the Service XSD.

Operations:

CreateOrderRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2
DeleteOrderRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2
GetOrderRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2
GetOrdersRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2
UpdateOrderRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2

Clients Overview

XSDS:

- Service Types
- Wcf Data Types
- Wcf Collection Types

WSDLS:

- soap11
- soap12

Debug Info:

- Operations Metadata

Figure 5: Metadata page

The page is enabled by default. It is available at http://<servername>/metadata, and can be disabled by placing the following code in the application host’s **`Configure`** method.

```
SetConfig(new EndpointHostConfig
{
    EnableFeatures = Feature.All.Remove(Feature.Metadata)
});
```

Chapter 3 Order Management System

In the previous chapter, we went through the basics of ServiceStack—what it is and which functionalities it supports.

In this chapter, we will spend some time carefully building our domain model and explaining what we want to achieve with the service we build. It's very important to understand the various parts in this chapter because the implementation that follows in Chapter 5 will be much clearer.

Order Management System

In order to demonstrate the usage of ServiceStack with a concrete example, we will create a simple order management system which will be far from complete but good enough for getting started.

The order management system will be implemented by three RESTful web services that will enable its clients to:

- **OrderService**: Create, retrieve, update, and delete orders.
- **OrderItemService**: Manage respective OrderItems (retrieve, add, and delete).
- **ProductService**: Create, retrieve, update, and delete **`Products`**.

The order management service will then be expanded, and more features and functionalities will be added, such as authentication, validation, and dependency injection, just to name a few.

Application Domain Model

The object model tries to be simple. At its heart, there are only a few classes: **`Order`**, **`OrderItem`**, **`Product`**, and **`Status`**.

Orders (Order Class)

One **`Order`** contains a list of **`OrderItems`**. The **`Id`** of an order is automatically generated by the system. The **Version** field is a special field that can be used to manage the *optimistic concurrency* when saving data to the repository—a topic which is beyond the scope of this book. An **`Order`** can be *Active* or *Inactive*, and this is reflected through its **`Status`**.

Order Items (OrderItem Class)

An item represents part of an order; it can be any food or beverage (e.g., pizza, beer, or coffee). An item contains a reference to a **`Product`** and has a **`Quantity`** attribute.

Product

A **Product** represents a part of the **Order** and it’s a reference class. A **Product** can be *Active* or *Inactive*, and this is reflected through its **Status**.

Status

The **Status** class is a reference data class. In our application, it will have only two possible statuses: *Active* and *Inactive*.

```
public class Order
{
    public int             Id           { get; set; }
    public bool            IsTakeAway   { get; set; }
    public DateTime        CreationDate { get; set; }
    public List<OrderItem> Items        { get; set; }
    public Status          Status       { get; set; }
    public int             Version      { get; set; }
}

public class OrderItem
{
    public int     Id       { get; set; }
    public Product Product  { get; set; }
    public int     Quantity { get; set; }
    public int     Version  { get; set; }
}

public class Product
{
    public int    Id      { get; set; }
    public string Name    { get; set; }
    public int    Version { get; set; }
    public Status Status  { get; set; }
}

public class Status
{
    public int    Id   { get; set; }
    public string Name { get; set; }
}
```

Modeling the URIs and HTTP Verbs

The service exposed has to define a very clear Uniform Resource Identifier (URI) structure and the respective HTTP verbs. Keep in mind that not all the verbs need to be implemented, but we choose what is appropriate.

Table 5 lists the operations exposed by the **OrderService**.

Table 5: Exposed OrderService operations

URI	HTTP Verb	Description
`/orders`	`GET`	Gets the full list of all orders and respective items.
`/orders`	`POST`	Creates a new order.
`/orders/1234`	`GET`	Gets the details of a single order.
`/orders/1234`	`PUT`	Updates an existing order.
`/orders/1234`	`DELETE`	Deletes an existing order.

The web service **`OrderItemService`** exposes operations related to the **`Order`**’s **`Items`** collection as shown in Table 6.

Table 6: Exposed OrderItemService operations

URI	HTTP Verb	Description
`/orders/1234/items`	`GET`	Gets a full list of items assigned to an order.
`/orders/1234/items/1`	`GET`	Gets an item assigned to a specific order.

`ProductService` exposes operations related to the **`Product`** object as shown in Table 7.

Table 7: Exposed ProductService operations

URI	HTTP Verb	Description
`/products`	`GET`	Gets a full list of all available products.
`/products`	`POST`	Creates a new product.
`/products/1234`	`GET`	Gets the details of a single product.
`/products/1234`	`PUT`	Updates an existing product.

URI	HTTP Verb	Description
`/products/1234`	`DELETE`	Deletes an existing product.

Repositories: Data Access

For the examples to be very short, we won't deal with any data access code in this book. All pseudo-database communication will go through specific repositories[20] (**`OrderRepository`**, **`ProductRepository`**). Therefore, we will handle our orders in memory.

Instances of the repositories will be directly injected at run time to the specific web service by using ServiceStack's default Funq IoC container.

Visual Studio Project Structure

There are many ways to keep the assemblies separated, and this usually depends on the project's needs. When working with services, I usually apply the following three rules:

- The facade layer should be a thin layer without any business logic.
- Keep the **application** implementation separated from the **hosting** application. This is very useful in case we want to reuse the application logic and expose it in a different way (e.g., as a desktop app, web service, or web application).
- Keep the Request and Response DTOs in a separate assembly as this can be shared with the client (especially useful for .NET clients).

[20] For more information about the Repository pattern, see http://msdn.microsoft.com/en-us/library/ff649690.aspx.

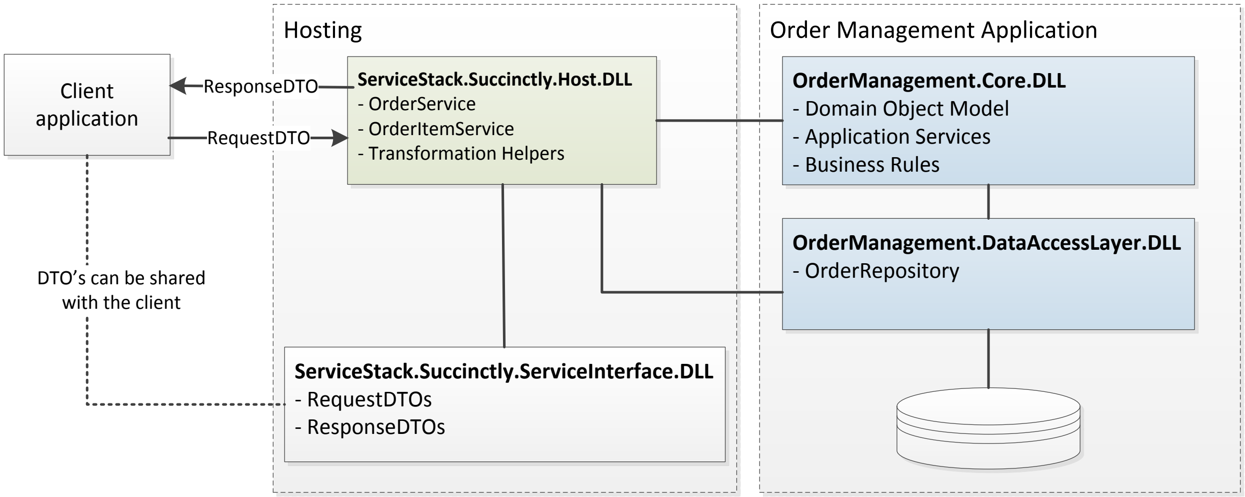


Figure 6: Project organization

ServiceStack.Succinctly.Host

`ServiceStack.Succinctly.Host` is the entry point of the application and the only component that communicates with the client. It contains the exposed services and all of the necessary plumbing, routing, instrumentation, etc. This is exactly where we are going to use the ServiceStack framework functionalities.

ServiceStack.Succinctly.ServiceInterface

`ServiceStack.Succinctly.ServiceInterface` contains Request and Response DTOs. This library can be directly shared with the client application to ease the communication and transformation of data. This library shouldn't contain any application logic.

OrderManagement.Core

`OrderManagement.Core` contains the domain model of the application and the required business logic, application logic, etc.

OrderManagement.DataAccessLayer

`OrderManagement.DataAccessLayer` contains the logic to access the database and the related repositories.

Chapter 4 Solution Configuration

It's finally time to start coding. In this chapter, we will see how to create the **`ServiceStack.Succinctly.Host`** project and enable the ServiceStack framework.

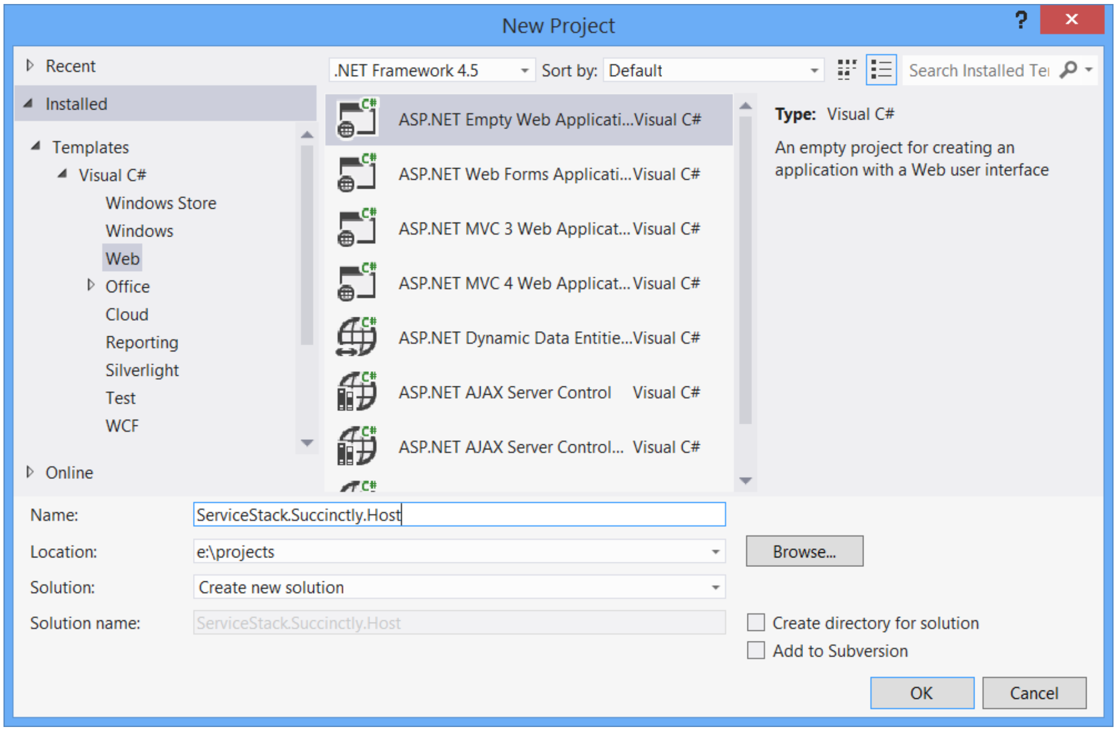
The first thing to do is to create a new solution and create a list of projects as specified in the following table.

Table 8: Project list

Visual Studio Project Name	Visual Studio Project Type
`ServiceStack.Succinctly.Host`	ASP.NET Empty Web Application
`ServiceStack.Succinctly.ServiceInterface`	Windows Class Library
`OrderManagement.Core`	Windows Class Library
`OrderManagement.DataAccessLayer`	Windows Class Library

Step 1: Create a New ASP.NET Empty Web Application

Use Visual Studio to create a new ASP.NET Empty Web Application project and call it **`ServiceStack.Succinctly.Host`**.


New Project
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Web
Office
Cloud
Reporting
Silverlight
Test
WCF
Online
Search Installed Te
ASP.NET Empty Web Applicati... Visual C#
ASP.NET Web Forms Applicati... Visual C#
ASP.NET MVC 3 Web Applicat... Visual C#
ASP.NET MVC 4 Web Applicat... Visual C#
ASP.NET Dynamic Data Entitie... Visual C#
ASP.NET AJAX Server Control Visual C#
ASP.NET AJAX Server Control... Visual C#
Browse...
OK
Cancel

.NET Framework 4.5 Sort by: Default

Type: Visual C#

An empty project for creating an application with a Web user interface

Name: ServiceStack.Succinctly.Host

Location: e:\projects

Solution: Create new solution

Solution name: ServiceStack.Succinctly.Host

□ Create directory for solution

□ Add to Subversion

Figure 7: Creation of a new ASP.NET Empty Web Application project

Step 2: Install ServiceStack Binaries and Configure web.config

In order to run our first web service, we need to add ServiceStack as a project reference. This can be done manually or by using NuGet, which is the easiest way in my opinion.

By running the following command in Visual Studio’s Package Manager Console, the reference will be automatically added to the project:

```
PM> Install-Package ServiceStack
```

The following packages will be installed by default:

- ServiceStack
- ServiceStack.Common
- ServiceStack.Interfaces
- ServiceStack.OrmLite
- ServiceStack.OrmLite.SqlServer
- ServiceStack.Redis
- ServiceStack.ServiceInterface
- ServiceStack.Text

Tip: If you want to mix ServiceStack with your existing ASP.NET or ASP.NET MVC applications, there are two packages that were created specifically to help you install ServiceStack automatically:

```
PM> Install-Package ServiceStack.Host.Mvc
PM> Install-Package ServiceStack.Host.AspNet
```

After adding the required binary files to the project, we need to instruct our application to use the ServiceStack HTTP handler, which will act as an entry point of the application. This is done by putting the following configuration in the **`web.config`** file.

```
<configuration>
  <!-- Required for IIS 6.0 (and above) -->
  <system.web>
    <httpHandlers>
     <add path="*"
          type="ServiceStack.WebHost.Endpoints.ServiceStackHttpHandlerFactory,
ServiceStack"
          verb="*"/>
    </httpHandlers>
    <compilation debug="true"/>
  </system.web>

  <!-- Required for IIS 7.0 -->
  <system.webServer>
    <validation validateIntegratedModeConfiguration="false"/>
    <handlers>
     <add path="*"
          name="ServiceStack.Factory"
          type="ServiceStack.WebHost.Endpoints.ServiceStackHttpHandlerFactory,
ServiceStack"
          verb="*"
          preCondition="integratedMode"
          resourceType="Unspecified"
          allowPathInfo="true"/>
    </handlers>
  </system.webServer>
</configuration>
```

Step 3: Creating the Application Host

In order to run a ServiceStack web service, we have to start the application host when the IIS application starts. There can be only one application host per application pool and, therefore, creating multiple application hosts is not allowed. The following code represents the minimum **`AppHostBase`** implementation. The **`Global`** class refers to the **`Global.asax.cs`** file.

```
public class Global : HttpApplication
{
    public class ServiceAppHost : AppHostBase
    {
        public ServiceAppHost()
            : base("Order Management", typeof(ServiceAppHost).Assembly)
        {
        }

        public override void Configure(Container container)
        {
        }
    }

    protected void Application_Start(object sender, EventArgs e)
    {
        new ServiceAppHost().Init();
    }
}
```

Chapter 5 Service Implementation

In this chapter, we will describe in detail how to build each of the services mentioned in Chapter 3. We will also describe how to implement all the relative components that enable full support for the functionalities we want to implement.

Our solution will contain three services and all three will implement all of the HTTP methods. The following table contains the list of verbs and the respective routes that will be implemented.

Table 9: Services with respective operations

Service	Description
`OrderService`	Contains methods that insert, delete, create, and update orders. • **GET** `/orders` • **GET** `/orders/{id}` • **POST** `/orders` • **PUT** `/orders/{id}` • **DELETE** `/orders/{id}`
`ProductService`	Contains methods that insert, delete, create, and update products. • **GET** `/products` • **GET** `/products/{id}` • **POST** `/products` • **PUT** `/products/{id}` • **DELETE** `/products/{id}`
`OrderItemService`	Contains methods that manipulate OrderItems associated with an order. • **GET** `/orders/{id}/items` • **GET** `/orders/{id}/items/{id}`

For all three services, we will implement the following components:

- Service Model (Request and Response DTO) object definitions.
- Route specification.
- Mapper(s) implementation.
- Validator implementation.
- Service implementation.
- Configuration (application host) wiring everything together.

Additional Information

Link and Status DTOs

All of the Response DTO objects will use the **`Status`** and **`Link`** classes. I've included the code here as a reference.

```
public class Status
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Link
{
    public string Rel { get; set; }
    public string Href { get; set; }
    public string Title { get; set; }
    public string Type { get; set; }
}
```

The **`Link`** class will contain the hypermedia information related to the resource.

Test Project

To test the various services, create a new project called **`ServiceStack.Succinctly.Host.IntegrationTest`** that references the following assemblies:

- From the NuGet package: **`ServiceStack.Common`**, which will install the **`ServiceStack.Common`**, **`ServiceStack.Text`**, and **`ServiceStack.Interfaces`** DLLs.
- **`Microsoft.VisualStudio.QualityTools.UnitTestFramework`**, which is the standard Microsoft .NET assembly that contains testing attributes.
- **`ServiceStack.Succinctly.ServiceInterface`**, which is the project that contains our DTO implementation.

Removing Namespaces

To remove superfluous namespaces from XML returned objects, add the following code to the **`Assembly.cs`** in the **`ServiceStack.Succinctly.ServiceInterface`** project.

```
[assembly: ContractNamespace("", ClrNamespace="ServiceStack.Succinctly.ServiceInte
rface.OrderModel")]
[assembly: ContractNamespace("", ClrNamespace="ServiceStack.Succinctly.ServiceInte
rface.OrderItemModel")]
[assembly: ContractNamespace("", ClrNamespace="ServiceStack.Succinctly.ServiceInte
rface.ProductModel")]
```

```
[assembly: ContractNamespace("", ClrNamespace="ServiceStack.Succinctly.ServiceInte
rface")]
```

Product Service

In our solution, the **Product** class can be seen as a reference data class as it is a property of the **OrderItem** class. Since one **Product** can be assigned to several **OrderItems**, there is a need to have a separate service that specifically manages a **Product** itself. **ProductService** is a classic create, read, update, and delete (CRUD) service.

Service Model

As a particularity of the service, I've chosen to create two separate DTOs for managing inserts and updates. I've done so to show that there is such a possibility and that we may fine-tune the actual requests. Indeed, the **CreateProduct** class specifically doesn't have the **Id** because the client shouldn't know about the **Id** when creating a product. However, you may choose not to implement it in such a way, but to have a more consistent object instead (as we are going to see in the **OrderService** implementation).

The following five classes will be used as Request DTOs of the service. The **GetProducts** class intentionally does not have properties; it is mainly used for routing a request to the proper service method as we will see later.

```
public class GetProducts { }

public class GetProduct
{
    public int Id { get; set; }
}

public class CreateProduct
{
    public string Name   { get; set; }
    public Status Status { get; set; }
}

public class UpdateProduct
{
    public int    Id     { get; set; }
    public string Name   { get; set; }
    public Status Status { get; set; }
}

public class DeleteProduct
{
    public int Id { get; set; }
}
```

Almost all of the Service methods will return the **ProductResponse** object. **ProductResponse** is a mirror of the **Product** and, as we will see, it can contain more or less information. In our case, it contains a list of **Links** and no information about the **Product.Version**, which is only used for optimistic concurrency control.

ProductsResponse instead will hold a list of **ProductResponse**. This is helpful as we can reuse this object and enrich it with further attributes that can be useful for paging, navigation, etc.

```
public class ProductResponse
{
    public int        Id     { get; set; }
    public string     Name   { get; set; }
    public Status     Status { get; set; }
    public List<Link> Links  { get; set; }
}

public class ProductsResponse
{
    public List<ProductResponse> Products { get; set; }
}
```

Route Specification

In the application host (**Global.asax.cs**), we need to register the various routes related to the **Product** service. As we have seen in the previous chapters, this is done either in the application host's constructor or in the **Configure** method. I've chosen to use the constructor because I want to use the **Configure** method only for the IoC-related items.

```
using ServiceStack.Succinctly.Host.Extensions;

public ServiceAppHost():                    base("Order Management", typeof (Servi
ceAppHost).Assembly)
{
    Routes
      .Add<GetProducts>  ("/products",      "GET",    "Returns Products")
      .Add<GetProduct>   ("/products/{Id}", "GET",    "Returns a Product")
      .Add<CreateProduct>("/products",      "POST",   "Creates a Product")
      .Add<UpdateProduct>("/products/{Id}", "PUT",    "Updates a Product")
      .Add<DeleteProduct>("/products/{Id}", "DELETE", "Deletes a Product");
}
```

The ServiceStack **Routes.Add()** method currently doesn't expose the method signature that we have just seen. To achieve this, I've created an extension method. The **Routes** property implements the **IServiceRoute** interface and, once we know this, it's very easy to extend.

```
public static class RoutesExtensions
{
    public static IServiceRoutes Add<T> (this IServiceRoutes routes,
                             string restPath, string verbs, string summary)
    {
        return routes.Add(typeof (T), restPath, verbs, summary, "");
    }

    public static IServiceRoutes Add<T>(this IServiceRoutes routes,
               string restPath, string verbs, string summary, string notes)
    {
        return routes.Add(typeof(T), restPath, verbs, summary, notes);
    }
}
```

Product Mapper Implementation

We need to map the data back and forth from the domain object model to the service object model (DTOs). The best way to do so is to create a specific class that enables the application to transform the **`Product`** domain object to **`ProductResponse`** and **`CreateProduct/UpdateProduct`** to the **`Product`**. Because this can be quite a heavy workload in the case of big classes, I advise you to use some specific libraries such as AutoMapper[21] as this would decrease the amount of necessary code. Explaining how AutoMapper works is outside the scope of this book but, as we will see, it's pretty intuitive and easy to use.

Adding the AutoMapper library to the project is as easy as running the following NuGet command:

```
PM> Install-Package AutoMapper
```

The **`ProductMapper`** implements the **`IProductMapper`** interface, which will make it easier to be injected to the Service. Let's create the following file in the **`ServiceStack.Succinctly.Host`** project under the **`ServiceStack.Succinctly.Host.Mappers`** namespace.

The following code example shows the definition of the **`IProductMapper`** interface.

```
using OrderManagement.Core.Domain;
using ServiceStack.Succinctly.ServiceInterface.ProductModel;

namespace ServiceStack.Succinctly.Host.Mappers
{
    public interface IProductMapper
    {
       Product ToProduct(CreateProduct request);
       Product ToProduct(UpdateProduct request);
       ProductResponse ToProductResponse(Product product);
```

[21] AutoMapper code is available on GitHub at https://github.com/AutoMapper/AutoMapper

```
        List<ProductResponse> ToProductResponseList(List<Product> products);
    }
}
```

The following code is the implementation of the **`ProductMapper`**.

```
using OrderManagement.Core.Domain;
using SrvObjType = ServiceStack.Succinctly.ServiceInterface;
using SrvObj = ServiceStack.Succinctly.ServiceInterface.ProductModel;

namespace ServiceStack.Succinctly.Host.Mappers
{
    public class ProductMapper : IProductMapper
    {
        static ProductMapper()
        {
            Mapper.CreateMap<SrvObjType.Status, Status>();
            Mapper.CreateMap<Status, SrvObjType.Status>();
            Mapper.CreateMap<SrvObj.CreateProduct, Product>();
            Mapper.CreateMap<SrvObj.UpdateProduct, Product>();
            Mapper.CreateMap<Product, SrvObj.ProductResponse>();
        }

        public Product ToProduct(SrvObj.CreateProduct request)
        {
            return Mapper.Map<Product>(request);
        }

        public Product ToProduct(SrvObj.UpdateProduct request)
        {
            return Mapper.Map<Product>(request);
        }

        public SrvObj.ProductResponse ToProductResponse(Product product)
        {
           var productResponse = Mapper.Map<SrvObj.ProductResponse>(product);

           productResponse.Links = new List<SrvObjType.Link>
                {
                    new SrvObjType.Link
                        {
                            Title = "self",
                            Rel = "self",
                            Href = "products/{0}".Fmt(product.Id),
                        }
                };
           return productResponse;
        }

        //Transforms a list of products into a list of ProductResponses.
        public List<SrvObj.ProductResponse> ToProductResponseList(
                                                      List<Product> products)
        {
            var productResponseList = new List<SrvObj.ProductResponse>();
            products.ForEach(x =>
                         productResponseList.Add(ToProductResponse(x)));
            return productResponseList;
        }
    }
}
```

Note: All of our services will have a Mapper class by design because we want to separate the Request and Response DTOs (service model) from the application domain model.

Validation Implementation

As we saw in the previous chapter, we can create some custom validation logic. We will create some in this case because we want to make our implementation a bit stronger. For our current example, we will just make sure that when the **`Product`** is created or updated, we check that the **`Name`** property is set and is not longer than 50 characters. We will create the following two classes in the **`ServiceStack.Succinctly.Host.Validation`** namespace, and will register those two validators in the application host.

```
public class CreateProductValidator : AbstractValidator<CreateProduct>
{
    public CreateProductValidator()
    {
        string nameNotSpecifiedMsg = "Name has not been specified.";
        string maxLenghtMsg = "Name cannot be longer than 50 characters.";

        RuleFor(r => r.Name)
            .NotEmpty().WithMessage(nameNotSpecifiedMsg)
            .NotNull().WithMessage(nameNotSpecifiedMsg)
            .Length(1, 50).WithMessage(maxLenghtMsg);
    }
}

public class UpdateProductValidator : AbstractValidator<UpdateProduct>
{
    public UpdateProductValidator()
    {
        string nameNotSpecifiedMsg = "Name has not been specified.";
        string maxLenghtMsg = "Name cannot be longer than 50 characters.";

        RuleFor(r => r.Name)
            .NotEmpty().WithMessage(nameNotSpecifiedMsg)
            .NotNull().WithMessage(nameNotSpecifiedMsg)
            .Length(1, 50).WithMessage(maxLenghtMsg);
    }
}
```

Application Host Configuration

The following code shows the full implementation of the application host.

```
public class ServiceAppHost : AppHostBase
{
  public ServiceAppHost()
        : base("Order Management", typeof(ServiceAppHost).Assembly)
   {
    Routes
    .Add<GetProducts>  ("/products", "GET", "Returns a collection of Products")
    .Add<GetProduct>   ("/products/{Id}", "GET", "Returns a single Product")
    .Add<CreateProduct>("/products", "POST", "Create a product")
    .Add<UpdateProduct>("/products/{Id}", "PUT", "Update a product")
    .Add<DeleteProduct>("/products/{Id}", "DELETE", "Deletes a product")
    .Add<DeleteProduct>("/products", "DELETE", "Deletes all products");

     Plugins.Add(new ValidationFeature());
   }

    public override void Configure(Container container)
    {
        container.Register<IProductRepository>(new ProductRepository());
        container.Register<IProductMapper>(new ProductMapper());

        container.RegisterValidator(typeof(CreateProductValidator));
        container.RegisterValidator(typeof(UpdateProductValidator));
    }
}
```

Service Implementation

ProductService implements two **Get** methods, one **Post**, one **Put**, and one **Delete**. Every ServiceStack service has to inherit from the **ServiceStack.ServiceInterface.Service** class.

As shown in the following code example, **ProductService** has two properties: **ProductMapper** and **ProductRepository**, which will hold the instances that will be injected at run time by the IoC container.

```
public class ProductService : ServiceStack.ServiceInterface.Service
{
    public IProductMapper ProductMapper { get; set; }
    public IProductRepository ProductRepository { get; set; }

    public ProductResponse Get(GetProduct request){…}
    public List<ProductResponse> Get(GetProducts request){}
    public ProductResponse Post(CreateProduct request){…}
    public ProductResponse Put(UpdateProduct request) {…}
    public HttpResult Delete(DeleteProduct request){…}
}
```

Get Products by Id

When calling **GET /product/1**, the following method will be called. As you can see, the implementation is quite simple:

- Get the data from the repository.
- If nothing has been found, then return the **404 Not Found** status code (the **Response.StatusCode** attribute can be used to do so).
- If something is returned by the repository, then by using the **ProductMapper**, we transform the **Product** (transform the domain model to the **ProductResponse** service model).

```
public ProductResponse Get(GetProduct request)
{
    var product = ProductRepository.GetById(request.Id);
    if (product == null)
    {
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        return default(ProductResponse);
    }

    //Transform to ProductsResponse and return.
    return ProductMapper.ToProductResponse(product);
}
```

In order to call the **GET** service method in the previous example, we can use any web client. The following example shows a unit test (using Microsoft Test Framework) that uses the built-in ServiceStack **JsonServiceClient** to perform a **GET** on the **/products/{Id}** URI.

```
[TestMethod]
public void GetProductByProductId_ReturnsValidProduct()
{
    //ARRANGE ---
    int PRODUCT_ID = 1;
    var client = new JsonServiceClient("http://localhost:50712/");

    //ACT  ------
    var product = client.Get<ProductResponse>("/products/" + PRODUCT_ID);

    //ASSERT ----
    Assert.IsTrue(product!=null);
    Assert.IsTrue(product.Id == PRODUCT_ID);
}
```

If we were to use a browser to navigate to http://localhost:50712/products/1.xml, we would get the following response.

```xml
<ProductResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Id>1</Id>
  <Links>
    <Link>
      <Href>products/1</Href>
      <Rel>self</Rel>
      <Title>self</Title>
      <Type i:nil="true"/>
    </Link>
  </Links>
  <Name>Pizza</Name>
  <Status>
    <Id>1</Id>
    <Name>Active</Name>
  </Status>
</ProductResponse>
```

Returning All Products

In this case, the service accepts a **`GetProducts`** message which, on its own, doesn’t have any property. But, as we are going to see later, being the collection that is returned, we can build the paging, sorting, filtering, etc. For the time being though, we simply pass a request and the service method will:

- Get the data from the repository.
- Transform the data from the domain model to the service model (DTO).

```csharp
//Returns all the Products.
public ProductsResponse Get(GetProducts request)
{
    //Get data from the database.
    List<Product> products = ProductRepository.GetAll();

    //Transform to ProductsResponse and return.
    return new ProductsResponse()
        {
            Products = ProductMapper.ToProductResponseList(products)
        };
}
```

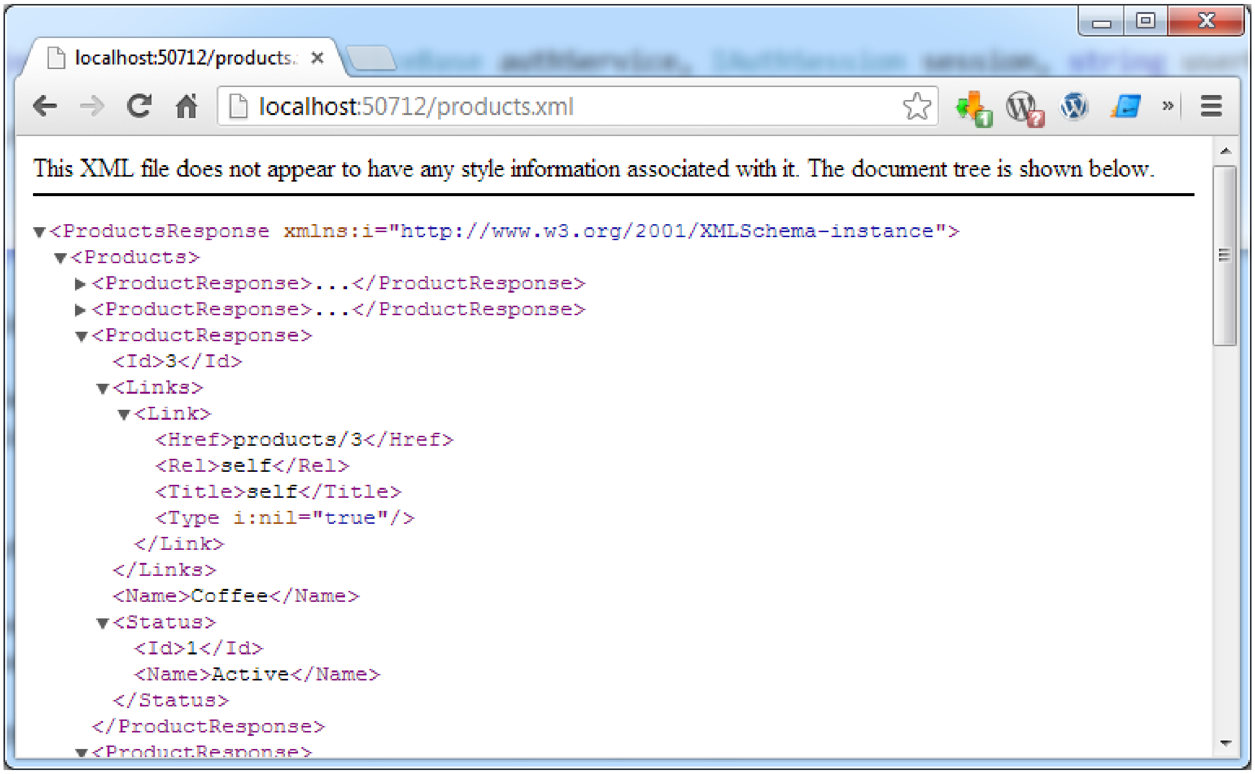
To call the **`/products`** in order to get the list of available products, the client would look like the following.

```csharp
[TestMethod]
public void GetAllProducts_ReturnsValidProductList()
{
    //ARRANGE ---
    var client = new JsonServiceClient("http://localhost:50712/");

    //ACT  ------
    var products = client.Get<ProductsResponse>("/products");

    //ASSERT ----
    Assert.IsTrue(products != null);
    Assert.IsTrue(products.Products.Count > 0);
}
```

And, as shown in the web browser:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<ProductsResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  ▼<Products>
    ▶<ProductResponse>...</ProductResponse>
    ▶<ProductResponse>...</ProductResponse>
    ▼<ProductResponse>
       <Id>3</Id>
     ▼<Links>
       ▼<Link>
          <Href>products/3</Href>
          <Rel>self</Rel>
          <Title>self</Title>
          <Type i:nil="true"/>
        </Link>
      </Links>
      <Name>Coffee</Name>
     ▼<Status>
        <Id>1</Id>
        <Name>Active</Name>
      </Status>
    </ProductResponse>
    ▼<ProductResponse>
```

Figure 8: List of products

Creating a New Product

When creating new **`Products`**, there are several things to note:

- To create objects, the **`POST`** method is used.
- Once the **`Product`** is created and successfully sent to the repository (database), we will return to the client the full **`Product`** as we did with the **`GET /products/1`**.
- To note that the object has been created, we will add the **`Location`** in the header and inform the client of the object's new location.
- Status code **`201 Created`** will be returned to the client.

```
//Returns all the orders.
public ProductResponse Post(CreateProduct request)
{
   //Transform the request to Domain.Product.
   var domainProduct = ProductMapper.ToProduct(request);

   //Storing data to database.
   var newProduct = ProductRepository.Add(domainProduct);

   //Transform to ProductResponse.
   var response = ProductMapper.ToProductResponse(newProduct);

   //Manipulate the header and StatusCode.
   Response.AddHeader("Location", Request.AbsoluteUri + "/" + newProduct.Id);
   Response.StatusCode = (int)HttpStatusCode.Created;

   return response;
}
```

To insert a new product using the ServiceStack client, we will test that the Response is returning **`201 Created`** HTTP status and that the **`Location`** header has been returned with the new URI.

```
[TestMethod]
public void CreateNewProduct_ReturnsObjectAnd201CreatedStatus()
{
    //ARRANGE ---
    WebHeaderCollection headers = null;
    HttpStatusCode statusCode = 0;
    const string PRODUCT_NAME = "Cappuccino";
    const string SITE = "http://localhost:50712";
    const string PRODUCTS = "/products";
    const string URI = SITE + PRODUCTS;

    var client = new JsonServiceClient(SITE)
        {
            //Grabbing the header once the call is ended.
            LocalHttpWebResponseFilter =
                httpRes =>
                {
                    headers = httpRes.Headers;
                    statusCode = httpRes.StatusCode;
                }
        };

    var newProduct = new CreateProduct
        {
            Name = PRODUCT_NAME,
            Status = new Status {Id = 1}
        };

    //ACT  ------
    var product = client.Post<ProductResponse>(PRODUCTS, newProduct);
    //ASSERT ----
    Assert.IsTrue(headers["Location"] == URI + "/" + product.Id);
    Assert.IsTrue(statusCode == HttpStatusCode.Created);
    Assert.IsTrue(product.Name == PRODUCT_NAME);
}
```

After the **POST** method has been called, the following is the response from the service, header, and body. As you can see, the **201 Created** verb and the **Location** headers are specified and returned to the client.

```
HTTP/1.1 201 Created
Cache-Control: private
Content-Type: application/xml
Location: http://localhost:50712/products/10
Server: Microsoft-IIS/8.0
X-Powered-By: ServiceStack/3.956 Win32NT/.NET
Date: Tue, 13 Aug 2013 22:15:42 GMT
Content-Length: 277

<ProductResponse>
  <Id>10</Id>
  <Links>
    <Link>
      <Href>products/10</Href>
      <Rel>self</Rel>
      <Title>self</Title>
      <Type i:nil="true" />
    </Link>
  </Links>
  <Name>Cappuccino</Name>
  <Status>
    <Id>1</Id>
    <Name>Active</Name>
  </Status>
</ProductResponse>
```

Updating a Product

To update a product, our service will implement the **PUT** method which will accept the **UpdateProduct** message.

Before doing any work, we check if the resource we are updating is available in the repository. If it is not, we then force the **404 Not Found** status code which indicates that the resource is not available. In a successful scenario, either the **200 (OK)** or **204 (No Content)** response codes should be sent to indicate successful completion of the request. We will return **200 (OK)** and the full body of the message if the call is successful.

```
public ProductResponse Put(UpdateProduct request)
{
    var domainObject = ProductRepository.GetById(request.Id);
    if (domainObject == null)
    {
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        return null;
    }

    //Transform to Domain.Product.
    var domainProduct = ProductMapper.ToProduct(request);

    //Store data to database.
    var updatedProduct = ProductRepository.Update(domainProduct);

    //Transform to ProductResponse and return.
    return ProductMapper.ToProductResponse(updatedProduct);
}
```

To test the product update, where only the status of the resource would be updated to **`Inactive`**:

```
[TestMethod]
public void UpdateProduct_ReturnsUpdatedObject()
{
    //ARRANGE ---
    HttpStatusCode statusCode = 0;
    const string PRODUCT_NAME = "White Wine";
    const string SITE = "http://localhost:50712";
    const string PRODUCT_LINK = "/products/2";

    var client = new JsonServiceClient(SITE)
        {
            //Grabbing the header once the call is ended.
            LocalHttpWebResponseFilter =
                httpRes =>
                {
                    statusCode = httpRes.StatusCode;
                }
        };

    var updateProduct = new UpdateProduct
        {
            Name = PRODUCT_NAME,
            Status = new Status {Id = 2} // Id = 2 means inactive.
        };

    //ACT  ------
    var product = client.Put<ProductResponse>(PRODUCT_LINK, updateProduct);

    //ASSERT ----
    Assert.IsTrue(statusCode == HttpStatusCode.OK);
    Assert.IsTrue(product.Name == PRODUCT_NAME);
    Assert.IsTrue(product.Status.Id == 2);
}
```

The difference between the original and the new (updated) resource is only the **`Status.Id`**.

Original Resource	**Server Response**

`<ProductResponse>`
`  <Id>2</Id>`
`  <Links>`
`    <Link>`
`      <Href>products/2</Href>`
`      <Rel>self</Rel>`
`      <Title>self</Title>`
`      <Type i:nil="true"/>`
`    </Link>`
`  </Links>`
`  <Name>White Wine</Name>`
`  <Status>`
`    <Id>1</Id>`
`    <Name>Active</Name>`
`  </Status>`
`</ProductResponse>`
	`HTTP/1.1 200 OK`
	`Cache-Control: private`
	`Content-Type: application/xml`
	`Server: Microsoft-IIS/8.0`
	`Date: Tue, 13 Aug 2013 22:19:55 GMT`
	`Content-Length: 278`
	
	`<ProductResponse>`
	`  <Id>2</Id>`
	`  <Links>`
	`    <Link>`
	`      <Href>products/2</Href>`
	`      <Rel>self</Rel>`
	`      <Title>self</Title>`
	`      <Type i:nil="true"/>`
	`    </Link>`
	`  </Links>`
	`  <Name>White Wine</Name>`
	`  <Status>`
	`    <Id>2</Id>`
	`    <Name>Inactive</Name>`
	`  </Status>`
	`</ProductResponse>`

Deleting a Product

In case the resource is not found, we can return the **`404 Not Found`** status code. But you might choose not to return an error and instead return **`200 OK`**. In this example, I've chosen to inform the client about the nonexistence of the resource. When returning **`201 NoContent`**, we won't return any body.

```
//Deletes a product.
public HttpResult Delete(DeleteProduct request)
{
    var domainObject = ProductRepository.GetById(request.Id);
    if (domainObject == null)
    {
        Response.StatusCode = (int)HttpStatusCode.NotFound;
    }
    else
    {
        ProductRepository.Delete(request.Id);
        Response.StatusCode = (int)HttpStatusCode.NoContent;
    }
    return null;
}
```

The following code tests the **`Delete`** method implemented in the service.

```
[TestMethod]
public void DeleteProduct_ReturnsNoContent()
{
    //ARRANGE ---
    HttpStatusCode statusCode = 0;
    const string SITE = "http://localhost:50712";

    var client = new JsonServiceClient(SITE)
        {
            //Grabbing the header once the call is ended.
            LocalHttpWebResponseFilter =
                httpRes =>
                {
                    statusCode = httpRes.StatusCode;
                }
        };

    //ACT  ------
    client.Delete<HttpResult>("/products/5");

    //ASSERT ----
    Assert.IsTrue(statusCode == HttpStatusCode.NoContent);
}
```

Order Service

`OrderService` will implement the following methods:

- **`GET`** `/orders`
- **`GET`** `/orders/{id}`
- **`POST`** `/orders`
- **`PUT`** `/orders/{id}`
- **`DELETE`** `/orders/{id}`

As we did in the **`ProductService`**, we will divide the implementation the same way.

Service Model

OrderService's service model is a bit more complex since we have a graph of objects that should mimic in some ways the domain object model. In this example, I've chosen to use the **Order** name for the DTO class when creating and updating an **Order** rather than the **CreateOrder** and **UpdateOrder** as I did in the previous chapter. This is mainly for brevity. Keep in mind that you should take care of the naming convention and follow it through all the services as much as possible (as it would be much easier for clients to understand your service's object model).

The following classes will be used as Request DTOs of the service.

```
public class GetOrders { }

public class GetOrder
{
    public int Id { get; set; }
}

public class DeleteOrder
{
    public int Id { get; set; }
}

//Class used for Create and Update.
public class Order
{
    public int             Id           { get; set; }
    public bool            IsTakeAway   { get; set; }
    public DateTime        CreationDate { get; set; }
    public Status          Status       { get; set; }
    public List<OrderItem> Items        { get; set; }
}

public class OrderItem
{
    public int     Id       { get; set; }
    public Product Product  { get; set; }
    public int     Quantity { get; set; }
}

public class Product
{
    public int    Id     { get; set; }
    public string Name   { get; set; }
    public Status Status { get; set; }
}
```

Almost all of the **Service** methods will return the **OrderResponse** object. **OrderResponse** is a representation of the **Order** which is a class within the domain object. But, as we will see, it can contain some more information. In our case, it contains a list of **Links**.

```
public class OrdersResponse
{
    public List<OrderResponse> Orders { get; set; }
}

public class OrderResponse
{
    public int Id { get; set; }
    public bool IsTakeAway { get; set; }
    public DateTime CreationDate { get; set; }
    public Status Status { get; set; }
    public List<OrderItemResponse> Items { get; set; }
    public List<Link> Links { get; set; }
}

public class OrderItemResponse
{
    public int Id { get; set; }
    public ProductResponse Product { get; set; }
    public int Quantity { get; set; }
    public List<Link> Links { get; set; }
}

public class ProductResponse
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Status Status { get; set; }
    public List<Link> Links { get; set; }
}
```

Route Specification

In the application host, we need to register the various Request DTOs to their relative URLs.

```
Routes
    .Add<GetOrder>   ("/orders/{Id}", "GET",    "Returns an Order")
    .Add<GetOrders>  ("/orders",      "GET",    "Returns Orders"  )
    .Add<Order>      ("/orders",      "POST",   "Creates an Order")
    .Add<Order>      ("/orders/{Id}", "PUT",    "Updates an Order")
    .Add<DeleteOrder>("/orders/{Id}", "DELETE", "Deletes an Order");
```

Order Mapper Implementation

In order to map the data back and forth from the domain object model to the service model (DTOs), as we did in the **ProductService** example, we will have a specific **OrderMapper** class for this.

The **OrderMapper** implements the **IOrderMapper** interface which will make it easier to be injected to the service.

```
using SrvObj = ServiceStack.Succinctly.ServiceInterface.OrderModel;
using Domain = OrderManagement.Core.Domain;

public interface IOrderMapper
{
   Domain.Order ToOrder(SrvObj.Order request);
   SrvObj.OrderResponse ToOrderResponse(Domain.Order order);
   List<SrvObj.OrderResponse> ToOrderResponseList(List<Domain.Order> orders);
   SrvObj.OrderItemResponse ToOrderItemResponse
                                   (int orderId, Domain.OrderItem orderItem);
   List<SrvObj.OrderItemResponse> ToOrderItemResponseList
                                 (int orderId, List<Domain.OrderItem> items);
}
```

The following example is the full implementation of the mapper.

```
public class OrderMapper : IOrderMapper
{
    static OrderMapper()
    {
        Mapper.CreateMap<Domain.Status, Status>();
        Mapper.CreateMap<Status, Domain.Status>();
        Mapper.CreateMap<SrvObj.Order, Domain.Order>();
        Mapper.CreateMap<SrvObj.OrderItem, Domain.OrderItem>();
        Mapper.CreateMap<SrvObj.Product, Domain.Product>();
        Mapper.CreateMap<Domain.Order, SrvObj.OrderResponse>();
        Mapper.CreateMap<Domain.OrderItem, SrvObj.OrderItemResponse>();
        Mapper.CreateMap<Domain.Product, SrvObj.ProductResponse>();
    }

    public Domain.Order ToOrder(SrvObj.Order request)
    {
        return Mapper.Map<Domain.Order>(request);
    }

    public SrvObj.OrderResponse ToOrderResponse(Domain.Order order)
    {
        var orderResponse = Mapper.Map<SrvObj.OrderResponse>(order);

        var orderSelfLink = "orders/{0}".Fmt(order.Id);
        orderResponse.Links = new List<Link>();
        orderResponse.Links.Add(SelfLink(orderSelfLink));
        orderResponse.Items.ForEach(x =>
            {
                var productId = x.Product.Id;
                var productLInk = "products/{0}".Fmt(productId);
                var itemsLink = orderSelfLink + "/items/{0}".Fmt(x.Id);
                x.Product.Links = new List<Link>();
                x.Product.Links.Add(SelfLink(productLInk));
                x.Links = new List<Link>();
                x.Links.Add(SelfLink(itemsLink));
            });
        return orderResponse;
    }

    private Link SelfLink(string uri)
    {
        return new Link
            {
                Title = "self",
                Rel = "self",
                Href = uri
            };
    }

    private Link ParentLink(string uri)
    {
        return new Link
            {
                Title = "parent",
                Rel = "parent",
                Href = uri
            };
    }

   public List<SrvObj.OrderResponse> ToOrderResponseList(List<Domain.Order> orders
)
    {
        var orderResponseList = new List<SrvObj.OrderResponse>();
        orders.ForEach(x => orderResponseList.Add(ToOrderResponse(x)));
        return orderResponseList;
    }

    public SrvObj.OrderItemResponse ToOrderItemResponse(int orderId,
                                                        Domain.OrderItem item)
    {
        var orderItemReponse = Mapper.Map<SrvObj.OrderItemResponse>(item);

        var productId = orderItemReponse.Product.Id;
        var orderLink = "orders/{0}".Fmt(orderId);
        var itemsLink = "/items/{0}".Fmt(item.Id);
        var productLink = "products/{0}".Fmt(productId);

        orderItemReponse.Links.Add(SelfLink(orderLink + itemsLink));
        orderItemReponse.Links.Add(ParentLink(orderLink));
        orderItemReponse.Product.Links.Add(SelfLink(productLink));

        return orderItemReponse;
    }

    public List<SrvObj.OrderItemResponse>
```

```
                   ToOrderItemResponseList(int orderId,
                                           List<Domain.OrderItem> items)
    {
      var orderItemResponseList = new List<SrvObj.OrderItemResponse>();
      items.ForEach(x => orderItemResponseList
                         .Add(ToOrderItemResponse(orderId, x)));
      return orderItemResponseList;
    }
}
```

Validation Implementation

The order validator is simple. It only checks if the order contains a not-empty **`OrderItems`** collection and it will simultaneously check that the **`CreationDate`** is not in the future.

```
public class OrderValidator : AbstractValidator<Order>
{
    public OrderValidator()
    {
        RuleFor(r => r.CreationDate)
            .LessThan(DateTime.Now.AddSeconds(10))
            .WithMessage("Creation Data shouldn't be in the future");

        RuleFor(r => r.Items)
            .NotEmpty()
            .WithMessage("Order Items should be specified");
    }
}
```

Application Host Configuration

The following code shows the full implementation of the application host. This comprehends the previously defined configuration for the **`ProductService`**.

```
public class ServiceAppHost : AppHostBase
{
    public ServiceAppHost()
        : base("Order Management", typeof (ServiceAppHost).Assembly)
    {
        Routes
         //Products
         .Add<GetProducts>("/products", "GET", "Returns Products")
         .Add<GetProduct>("/products/{Id}", "GET", "Returns a Product")
         .Add<CreateProduct>("/products", "POST", "Creates a Product")
         .Add<UpdateProduct>("/products/{Id}", "PUT", "Updates a Product")
         .Add<DeleteProduct>("/products/{Id}", "DELETE", "Deletes a Product")
         //Orders
         .Add<GetOrder>("/orders/{Id}", "GET", "Returns an Order")
         .Add<GetOrders>("/orders", "GET", "Returns Orders")
         .Add<Order>("/orders", "POST", "Creates an Order")
         .Add<Order>("/orders/{Id}", "PUT", "Updates an Order")
         .Add<DeleteOrder>("/orders/{Id}", "DELETE", "Deletes an Order");

        Plugins.Add(new ValidationFeature());
    }

    public override void Configure(Container container)
    {
        //Product dependencies
        container.Register<IProductRepository>(new ProductRepository());
        container.Register<IProductMapper>(new ProductMapper());

        //Orders dependencies
        container.Register<IOrderRepository>(new OrderRepository());
        container.Register<IOrderMapper>(new OrderMapper());
        container.Register<IStatusRepository>(new StatusRepository());

        //Validators
        container.RegisterValidator(typeof (CreateProductValidator));
        container.RegisterValidator(typeof (UpdateProductValidator));
        container.RegisterValidator(typeof (OrderValidator));
     }
}
```

The highlighted code is what is needed for the **OrderService**. As we may see, it is just a natural progression by following the same style of coding and service creation.

Service Implementation

OrderService implements two **Get** methods, one **Post**, one **Put**, and one **Delete**.

As shown in the following code example, **OrderService** has four public properties, namely **OrderMapper**, **OrderRepository**, **ProductRepository**, and **StatusRepository**, which will hold the instances that will be injected at run time by the IoC container.

```
public class OrderService : ServiceStack.ServiceInterface.Service
{
    public IOrderRepository   OrderRepository   { get; set; }
    public IProductRepository ProductRepository { get; set; }
    public IStatusRepository  StatusRepository  { get; set; }
    public IOrderMapper       OrderMapper       { get; set; }

    //Returns all the orders.
    public OrdersResponse Get(GetOrders request) { … }
    //Returns a single order.
    public OrderResponse Get(GetOrder request){ … }
    //Creates a new order.
    public OrderResponse Post(Order request) { … }
    //Updates an existing order /orders/{id}.
    public OrderResponse Put(Order request){ … }
    //Delete an order.
    public HttpResult Delete(DeleteOrder request){ … }
}
```

Getting Orders by OrderId

When calling **GET /orders/1**, the **Get(GetProduct request)** method will be called. As we have seen previously with the **ProductService**, the structure is very similar:

1. Get the data from the repository.
2. If nothing is found, return the **404 Not Found** error. This is done by setting the **Response.StatusCode** value.
3. If something is returned by the repository, use the **OrderMapper** to transform the **Order** (transform the domain model to the **OrderResponse** service model).

```
//Returns a single order.
public OrderResponse Get(GetOrder request)
{
    var domainObject = OrderRepository.GetById(request.Id);
    if (domainObject == null)
    {
        Response.StatusCode = (int) HttpStatusCode.NotFound;
        return null;
    }
    else
    {
        //Transform to OrderResponse and return.
        return OrderMapper.ToOrderResponse(domainObject);
    }
}
```

In order to call the **GET** service method used in the previous example, we will use the **XmlServiceClient** which, by default, would communicate with the **Content-Type: application/xml**. The following example shows a test that uses the built-in ServiceStack client to perform a **GET** on the **/orders/{Id}** URI.

```
[TestMethod]
public void GetOrdersByOrderId()
{
    //ARRANGE ---
    const int ORDER_ID = 1;
    var client = new XmlServiceClient("http://localhost:50712/");

    //ACT  ------
    var order = client.Get<OrderResponse>("/orders/" + ORDER_ID);

    //ASSERT ----
    Assert.IsTrue(order != null);
    Assert.IsTrue(order.Id == ORDER_ID);
}
```

Returning All Orders

In this case, the service accepts a **`GetOrders`** message. As we have seen in the **`ProductService`**, we follow the same procedure:

1. Get data from the repository.
2. Transform data from the domain to the service object model.

```
public OrdersResponse Get(GetOrders request)
{
    //Get data from the database.
    var orders = OrderRepository.GetAllOrders();

    //Transform to OrdersResponse and return.
    return new OrdersResponse()
    {
        Orders = OrderMapper.ToOrderResponseList(orders)
    };
}
```

To call **`GET /orders`** (to get the list of available orders), the client code would look like the following example.

```
[TestMethod]
public void GetAllOrders()
{
    //ARRANGE ---
    var client = new XmlServiceClient("http://localhost:50712/");

    //ACT  ------
    var orders = client.Get<OrdersResponse>("/orders");

    //ASSERT ----
    Assert.IsTrue(orders != null);
    Assert.IsTrue(orders.Orders.Count > 0);
}
```

Creating a New Order

When creating new **`Orders`**, there are several things to note:

- To create objects, the **`POST`** method is used.
- Before doing any further checks, we will make sure that a correct **`Domain.Order`** is generated, and therefore a **`Status`** and **`Product`** are retrieved from the repository and assigned to the object.
- Once the **`Order`** is created and successfully sent to the repository, we will return the full **`Order`** representation client as we would do with **`GET /orders/1`**.
- To note that the object has been created, we will add the location in the header and inform the client of the object's new location.
- Status code **`201 Created`** will be returned to the client.

```
public OrderResponse Post(Order request)
{
    //Transforming the IDs into real object objects.
    var newOrder = OrderMapper.ToOrder(request);
    newOrder.Status = StatusRepository.GetById(request.Status.Id);

    newOrder.Items.ForEach(x =>
        {
            x.Product = ProductRepository.GetById(x.Product.Id);
        });

    //Storing data to the database.
    newOrder = OrderRepository.Add(newOrder);

    //Transform to OrderResponse.
    var response = OrderMapper.ToOrderResponse(newOrder);

    //Manipulate the header and StatusCode.
    Response.AddHeader("Location", Request.AbsoluteUri + "/" + newOrder.Id);
    Response.StatusCode = (int)HttpStatusCode.Created;

    return response;
}
```

To insert a new **`Order`** using the ServiceStack client, we will test that the response is returning the **`201 Created`** HTTP status and that the **`Location`** header has been filled.

```
[TestMethod]
public void CreateNewOrder()
{
    //ARRANGE ---
    WebHeaderCollection headers = null;
    HttpStatusCode statusCode = 0;
    const string SITE = "http://localhost:50712";
    const string ORDERS = "/orders";
    const string URI = SITE + ORDERS;

    var client = new XmlServiceClient(SITE)
        {
            //Grabbing the header once the call is ended.
            LocalHttpWebResponseFilter =
                httpRes =>
                    {
                        headers = httpRes.Headers;
                        statusCode = httpRes.StatusCode;
                    }
        };

    var newOrder = new Order
      {
         CreationDate = DateTime.Now,
         IsTakeAway = true,
         Status = new Status {Id = 1}, //Active
         Items = new List<OrderItem>
           {
             new OrderItem {Product = new Product {Id = 1}, Quantity = 10},
             new OrderItem {Product = new Product {Id = 2}, Quantity = 10}
            }
       };

    //ACT  ------
    var order = client.Post<OrderResponse>(ORDERS, newOrder);

    //ASSERT ----
    Assert.IsTrue(headers["Location"] == URI + "/" + order.Id);
    Assert.IsTrue(statusCode == HttpStatusCode.Created);
    Assert.IsTrue(order.Items.Count == 2);
    Assert.IsTrue(order.Status.Id == 1); //Status is active.
}
```

After the **`Order`** is created, the following XML will be generated and headers will be returned to the client.

```
HTTP/1.1 201 Created
Cache-Control: private
Content-Type: application/xml
Location: http://localhost:50712/orders/5
Server: Microsoft-IIS/8.0
X-Powered-By: ServiceStack/3.956 Win32NT/.NET
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Wed, 14 Aug 2013 22:13:31 GMT
Content-Length: 722

<OrderResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <CreationDate>2013-08-13T00:00:00</CreationDate>
  <Id>5</Id>
  <IsTakeAway>false</IsTakeAway>
  <Items>
    <OrderItemResponse>
      <Id>8</Id>
      <Links>
        <Link>
          <Href>orders/5/items/8</Href>
          <Rel>self</Rel>
          <Title>self</Title>
          <Type i:nil="true" />
        </Link>
      </Links>
      <Product>
        <Id>1</Id>
        <Links>
          <Link>
            <Href>products/1</Href>
            <Rel>self</Rel>
            <Title>self</Title>
            <Type i:nil="true" />
          </Link>
        </Links>
        <Name>Pizza</Name>
        <Status>
          <Id>1</Id>
          <Name>Active</Name>
        </Status>
      </Product>
      <Quantity>10</Quantity>
    </OrderItemResponse>
  </Items>
  <Links>
    <Link>
      <Href>orders/5</Href>
      <Rel>self</Rel>
      <Title>self</Title>
      <Type i:nil="true" />
    </Link>
  </Links>
  <Status>
    <Id>1</Id>
    <Name>Active</Name>
  </Status>
</OrderResponse>
```

Updating an Order

To update an **`Order`**, our service will implement the **`PUT`** method, which will accept the **`Order`** message.

If an existing resource is modified, either the **200 (OK)** or **204 (No Content)** response codes should be sent to indicate successful completion of the request. In our case, if everything goes well, we don't specify anything because **200 OK** is the default response. If the **Order** doesn't exist, I've chosen to return a "Not Found" message.

```
//Updates an existing order /orders/{id}.
public OrderResponse Put(Order request)
{
    var domainObject = OrderRepository.GetById(request.Id);
    if (domainObject == null)
    {
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        return null;
    }

    var updatedOrder = OrderMapper.ToOrder(request);
    updatedOrder.Status = StatusRepository.GetById(request.Status.Id);
    updatedOrder.Items.ForEach(x =>
        {
            x.Product = ProductRepository.GetById(x.Product.Id);
        });

    //Store data to database.
    var order = OrderRepository.Update(updatedOrder);

    //Transform to OrderResponse and return.
    return OrderMapper.ToOrderResponse(order);
}
```

To test the **Order** update, we will change a couple of fields: **CreationDate**, **IsTakeAway**, **Quantity**, and the **Product** associated with the **OrderItem** with an **Id** of 6.

In the assert section, we will check that the newly created object has the values assigned to the updated **Order**.

```
[TestMethod]
public void UpdateOrder()
{
    //ARRANGE ---
    HttpStatusCode statusCode = 0;
    const string SITE = "http://localhost:50712";
    const string ORDERS_LINK = "/orders/1";
    const string URI = SITE + ORDERS_LINK;
    const int NEW_PRODUCT_ID = 5;
    DateTime NEW_CREATION_DATE = new DateTime(2013, 08, 08);

    var client = new XmlServiceClient(SITE)
    {
        //Grabbing the header once the call is ended.
        LocalHttpWebResponseFilter =
            httpRes =>
            {
                statusCode = httpRes.StatusCode;
            }
    };

    var updateOrder = new Order
        {
            CreationDate = NEW_CREATION_DATE,
            IsTakeAway = false,
            Items = new List<OrderItem>()
                {
                    new OrderItem
                        {
                            Id = 6,
                            //Setting a different product!
                            Product = new Product {Id = NEW_PRODUCT_ID},
                            //Setting a different quantity.
                            Quantity = 100
                        }
                },
            Status = new Status {Id = 1}
        };

    //ACT  ------
    var orderResponse = client.Put<OrderResponse>(ORDERS_LINK, updateOrder);

    //ASSERT ----
    Assert.IsTrue(statusCode == HttpStatusCode.OK);
    Assert.IsTrue(orderResponse.CreationDate == NEW_CREATION_DATE);
    Assert.IsTrue(orderResponse.IsTakeAway == false);
    Assert.IsTrue(orderResponse.Items.Count == 1);
    Assert.IsTrue(orderResponse.Items[0].Product.Id == NEW_PRODUCT_ID);
}
```

When testing the data in the browser by accessing **`GET /Orders/6`**, we can see that the **`Order`** has been updated.

Deleting an Order

When deleting an **`Order`**, the important thing is to return the "**`No content`**" status after the deletion has taken place. When returning the "**`No content`**" status, the message body shouldn't be returned back to the client.

```
public HttpResult Delete(DeleteOrder request)
{
    var domainObject = OrderRepository.GetById(request.Id);
    if (domainObject == null)
    {
        Response.StatusCode = (int)HttpStatusCode.NotFound;
    }
    else
    {
        //Delete order in the database.
        OrderRepository.Delete(request.Id);
        Response.StatusCode = (int)HttpStatusCode.NoContent;
    }

    //Not returning any body!
    return null;
}
```

The following code tests the **`Delete`** method implemented in the service.

```
[TestMethod]
public void DeleteOrder()
{
    //ARRANGE ---
    HttpStatusCode statusCode = 0;
    const string SITE = "http://localhost:50712";

    var client = new XmlServiceClient(SITE)
    {
        //Grabbing the header once the call is ended.
        LocalHttpWebResponseFilter =
            httpRes =>
            {
                statusCode = httpRes.StatusCode;
            }
    };

    //ACT  ------
    client.Delete<HttpResult>("/orders/2");

    //ASSERT ----
    Assert.IsTrue(statusCode == HttpStatusCode.NoContent);
}
```

OrderItem Service

`OrderItemService` will implement the following methods:

- **`GET`** `/orders/1234/items`
- **`GET`** `/orders/1234/items/{id}`

With this example, I want to show how to create a service that will enable navigating complex properties of a parent resource. While we can implement the creation or updating of an existing **`OrderItem`**, I think that is not as important since we have already seen how to create or update a resource in the previous two services. So, I’ve intentionally omitted the POST, PUT, and DELETE verbs for the **`OrderItem`**.

Service Model

OrderItemService’s service model is simple. The following classes will be used as the Request DTOs of the service.

```
public class GetOrderItem
{
    public int OrderId { get; set; }
    public int ItemId { get; set; }
}

public class GetOrderItems
{
    public int OrderId { get; set; }
}
```

We will reuse the **OrderItemResponse** that we have previously created for the **OrderService** because it perfectly suits the need, and create a new class, **OrderItemsResponse**, that will carry on a list of **OrderItem** responses.

```
public class OrderItemsResponse
{
    public List<OrderItemResponse> Items { get; set; }
}
```

Route Specification

In the application host, we need to register the various Request DTOs to their relative URIs.

```
Routes
.Add<GetOrderItem>("/orders/{OrderId}/items/{ItemId}","GET"," Get an OrderItem")
.Add<GetOrderItems>("/orders/{OrderId}/items", "GET", "Get a list of OrderItems");
```

Mapper Implementation

We will reuse the **OrderMapper** because it implements all of the necessary mappings.

Service Implementation

The following code example shows the public members of the **OrderItemService**. As in previous examples, public members will be injected by the IoC.

```
public interface IOrderItemService
```

```
{
    public IOrderRepository OrderRepository { get; set; }
    public IProductRepository ProductRepository { get; set; }
    public IStatusRepository StatusRepository { get; set; }
    public IOrderMapper OrderMapper { get; set; }

    OrderItemResponse  Get(GetOrderItem request){};
    OrderItemsResponse Get(GetOrderItems request){};
}
```

Returning All OrderItems Associated with a Specific Order

When calling **`GET /orders/1/items`**, the **`Get(GetOrderItem request)`** method will be called.

```
public OrderItemsResponse Get(GetOrderItems request)
{
    var order = OrderRepository.GetById(request.OrderId);

    List<Domain.OrderItem> orderItems = order.Items;

    if (orderItems == null || orderItems.Count == 0)
    {
        Response.StatusCode = (int) HttpStatusCode.NotFound;
        return null;
    }
    else
    {
        return new OrderItemsResponse
            {
                Items = OrderMapper
                .ToOrderItemResponseList(request.OrderId, orderItems)
            };
    }
}
```

The client code will check the existence of the items.

```
[TestMethod]
public void GetOrderItemsByOrderId()
{
    //ARRANGE ---
    int ORDER_ID = 1;
    string ITEMS_LINK = "/orders/" + ORDER_ID + "/items";

    var client = new XmlServiceClient("http://localhost:50712/");

    //ACT  ------
    var items = client.Get<OrderItemsResponse>(ITEMS_LINK);

    //ASSERT ----
    Assert.IsNotNull(items != null);
    Assert.IsNotNull(items.Items);
    Assert.IsTrue(items.Items.Count == 2);
}
```

Returning a Specific OrderItem

The particularity of this method is that **`GetOrderItem`** contains both the **`OrderId`** and **`ItemId`** properties.

```
public OrderItemResponse Get(GetOrderItem request)
{
    var order = OrderRepository.GetById(request.OrderId);

    Domain.OrderItem orderItem =
        order.Items
            .FirstOrDefault(x => x.Id == request.ItemId);

    if (orderItem == null)
    {
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        return null;
    }

    OrderItemResponse response =
        OrderMapper
        .ToOrderItemResponse(order.Id, orderItem);

    return response;
}
```

And it contains the test code that fully tests the URI **`GET /orders/1/items/2`**.

```
[TestMethod]
public void GetOrderItemByOrderId()
{
    //ARRANGE ---
    var ITEM_ID = "2";
    var client = new XmlServiceClient("http://localhost:50712/");
    string ITEM_LINK = "/orders/1/items/2";

    //ACT  ------
    var item = client.Get<OrderItemResponse>(ITEM_LINK);

    //ASSERT ----
    Assert.IsNotNull(item != null);
    Assert.IsTrue(item.Id.ToString() == ITEM_ID);
}
```

Conclusion

In this chapter, we have seen the implementation of all the verbs for the three web services as well as how to wire the requests to the right call.

We were able to test the implementation by using the unit tests, and we have seen how to use the **`JsonServiceClient`** and the **`XmlServiceClient`** to call the newly implemented services.

Chapter 6 Pagination

It's almost always a bad idea to return every available resource to the client at once. A technique that is usually referred to as **pagination** (or sometimes as **paging mechanism**, especially on webpages) is used for displaying a limited number of results where a returned data set is too big to be displayed in one place or at once.

Usually whenever pagination is implemented, this is followed by the information on how to get the next result set or how to get to the beginning of the list. RESTful services are not much different in this sense. It is possible to create a representation of resources by using pagination.

Pagination as a Representation

When building REST web services, we usually think about resources. The first question to pose is if the page or pagination is a resource on its own. In my opinion it is not, and, instead, the information about the pagination is just an integrated part of the resource itself. In other words, the information about how to page through, let’s say a collection of **`Products`**, will be embedded in the **`Product`** collection itself.

There are mainly three locations to place the information about pagination:

- Directly in the URI path: **`/products/page/1`**.
- Part of the URI query string: **`/products?page=1`**.
- Range HTTP headers: a topic we won’t be covering in this book, but is worth mentioning.[22]

As previously mentioned, the first option is not the best idea because pagination is not a resource (this option, through the URI, looks like a **`page`** is a resource). So, I would generally skip this way of representing the pagination.

The second option represents a standard way of solving this problem as it enables the encoding of paging application directly in the query string. In our examples, we are going to use this approach.

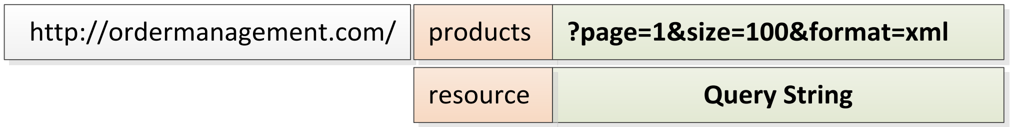


Figure 9: Pagination query options

[22] For more info about HTTP Range headers, visit http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.35.

In this chapter, we are going to see how to implement pagination for the already existing **Products** that we have implemented in the **`ProductService`**. The pagination makes sense only when returning a list of objects and, therefore, the perfect candidate for us would be the **`GET /products`** method which returns the list of available products.

Query String

There are several ways and styles of expressing the paging components when it comes to the naming convention for the query string parameters. In some examples, you will see **`offset`** and **`limit`** being mentioned. In other examples, you will see **`skip`** and **`take`**. But I will use a more intuitive and simple way, which is **`page`** and **`size`**:

- **`page`**: Represents the current page.
- **`size`**: Represents the number of items to be displayed in a page.

Discoverability

In order to inform the client application that there is a way to navigate through pages, we will use the **`Links`** section. Discoverability is part of the aforementioned HATEOAS constraint. The basic idea is the service exposes links to resources available. In our case, they are *next*, *previous*, *first*, and *last* so that the client is able to discover the URI associated with the resource and to further navigate to the resource. It is a bit similar to navigating a webpage through a browser. The links are the shown to us and, by following the links, we can discover new resources (in this case, pages).

Actually, the logic is straightforward; the client will know which URI is available through the **`Link.Rel`** attribute.

Table 10: Pagination links

Link.Rel	Description
`next`	Goes to the next page.
`previous`	Goes to the previous page.
`first`	Goes to the first page, which is the beginning of the list.
`last`	Goes to the last page, which is the end of the list.

Service Model

You might remember the **`GetProducts`** object that was used to retrieve a list of **`Products`**. We will expand this object to contain the two aforementioned query string parameters, and we will add a **`List<Link>`** property to the **`ProductsResponse`**.

```
public class GetProducts
{
    public int? Page { get; set; }
    public int? Size { get; set; }
}

public class ProductsResponse
{
    public ProductsResponse()
    {
        Links = new List<Link>();
    }

    public List<ProductResponse> Products { get; set; }
    public List<Link> Links { get; set; }
}

public class ProductResponse
{
    public ProductResponse()
    {
        Links = new List<Link>();
    }

    public int Id { get; set; }
    public string Name { get; set; }
    public Status Status { get; set; }
    public List<Link> Links { get; set; }
}
```

Data Access Layer

In order to add some structure to the pagination, we introduce in the data access layer a new class, **`PagedListResult`**.[23] An instance of this class will be returned from the repository when searching for products. The **`PagedListResult`** class is implemented as follows.

```
public class PagedListResult<TEntity>
```

[23] For the full implementation of this pattern, visit: http://www.agile-code.com/blog/entity-framework-code-first-filtering-and-sorting-with-paging-1.

```
{
    public bool HasNext { get; set; }
    public bool HasPrevious { get; set; }
    public int Page { get; set; }
    public int Size { get; set; }
    public long LastPage()
    {
        long lastPage = (Count/Size);
        if ((lastPage*Size) < Count) { lastPage++;}
        return lastPage;
    }
    public long Count { get; set; }
    public IList<TEntity> Entities { get; set; }
}
```

`ProductRepository` contains a new method called **`GetPaged()`**. As shown in the following code example, instead of only returning the products, we will enrich the object with extra information such as **`HasNext`**, **`HasPrevious`**, and **`Count`**, which represents the total number of items. This will later enable the service to properly create page links. The implementation is basic; it doesn't contain any code regarding sorting fields and direction in order to keep the example simple. The intention is not to show how to implement the repository code, but rather how to wire all the pieces together.

```
List<Product> Products = new List<Product>();

public PagedListResult<Product> GetPaged(int page, int size)
{
    var skip = (page == 0 || page == 1) ? 0 : (page - 1)*size;
    var take = size;

    IQueryable<Product> sequence = Products.AsQueryable();

    var resultCount = sequence.Count();

    //Getting the result.
    var result = (take > 0)
                        ? (sequence.Skip(skip).Take(take).ToList())
                        : (sequence.ToList());

    //Setting up the return object.
    bool hasNext = (skip > 0 || take > 0)
                    && (skip + take < resultCount);

    return new PagedListResult<Product>()
        {
            Page = page,
            Size = size,
            Entities = result,
            HasNext = hasNext,
            HasPrevious = (skip > 0),
            Count = resultCount
        };
}
```

Product Mapper

As shown previously when we looked at the **`ProductService`**, we will extend the **`ProductMapper`** class with a new method called **`ToProductsResponse()`** to which we will pass the **`PagedListResult<Product>`** structure. I have omitted the actual Links generation logic to keep the code succinct; I show only the **`NextLink()`** method. But the important thing to note here is that we will generate extra links as explained in the Discoverability section.

```
public ProductsResponse ToProductsResponse(PagedListResult<Product> products)
{
    var productList = ToProductResponseList(products.Entities.ToList());

    var productResponse = new ProductsResponse();
    productResponse.Products = productList;

    SelfLink(products, productResponse);
    NextLink(products, productResponse);
    PreviousLink(products, productResponse);
    FirstLink(products, productResponse);
    LastLink(products, productResponse);
    return productResponse;
}

private void NextLink(PagedListResult<Product> products,
                                            ProductsResponse productResponse)
{
    if (products.HasNext)
    {
        productResponse
            .Links.Add(NewLink("next", "products?page={0}&size={1}"
                           .Fmt(products.Page + 1, products.Size)));
    }
}
```

Service Implementation

Getting the list of products now requires two scenarios. In the first scenario, we want the paging, and in the second scenario, all the data is returned. This is not necessarily a real-world scenario, but I think that it is important to show this kind of implementation as well. If the page and size parameters are not supplied, the **`Get`** method will return the full list of products. Otherwise, the paging will kick in.

```
public ProductsResponse Get(GetProducts request)
{
    ProductsResponse response;
    int? page = request.Page;
    int? size = request.Size;

    if (page.HasValue && size.HasValue)
    {
        //Get data from the database.
        PagedListResult<Product> products =
                          ProductRepository.GetPaged(page.Value, size.Value);

        response = ProductMapper.ToProductsResponse(products);
    }
    else
    {
        var products = ProductRepository.GetAll();
        response = new ProductsResponse()
            {
                Products = ProductMapper.ToProductResponseList(products)
            };
    }
    return response;
}
```

And finally, when retrieving data, we can see that the links get updated every time there are available products. The response of the web service would look like the following if we were requesting **`GET /products?page=1&size=100`**.

```
<ProductsResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Links>
    <Link>
      <Href>products?page=1&size=100</Href>
      <Rel>self</Rel>
      <Title>self</Title>
      <Type i:nil="true"/>
    </Link>
    <Link>
      <Href>products?page=2&size=100</Href>
      <Rel>next</Rel>
      <Title>next</Title>
      <Type i:nil="true"/>
    </Link>
    <Link>
      <Href>products?page=1&size=100</Href>
      <Rel>first</Rel>
      <Title>first</Title>
      <Type i:nil="true"/>
    </Link>
    <Link>
      <Href>products?page=10&size=100</Href>
      <Rel>last</Rel>
      <Title>last</Title>
      <Type i:nil="true"/>
    </Link>
  </Links>
  <Products><!--omitted for brevity--></Products>
</ProductsResponse>
```

Chapter 7 Authentication

We have seen how to create the service, and now we want to control access to it. In this chapter, we will look at some options for securing a web application from unauthorized users.

Note: Authentication is all about knowing the identity of the user.
Authorization is deciding whether or not a user is allowed to perform an action.

ServiceStack contains a built-in authentication mechanism which enables services to authenticate users, register new users, and manage roles. The authentication feature is organized into four different components and an authentication domain object model:

- Authentication providers
- Authentication session cache clients
- User authentication repository
- Authentication services
- Domain object model: **`AuthUserSession`**, **`UserAuth`**, and **`UserOAuthProvider`**

How Does the Authentication Work?

The **authentication provider** is responsible for defining the way in which the authentication and authorization is going to happen. In order to retrieve the information about the user, the provider will directly communicate with the **authentication repository**, and after the successful retrieval of the user data and *de facto* authentication of the user, the **`Session`** for the given user will be stored in the cache for a limited time.

In the case of the OAuth[24] or OpenID[25] authentication, the information about the user will be directly provided by the service (Twitter, Google, Facebook, etc.).

[24] For more information about OAuth, visit http://oauth.net.
[25] For more information about OpenID, visit http://openid.net.

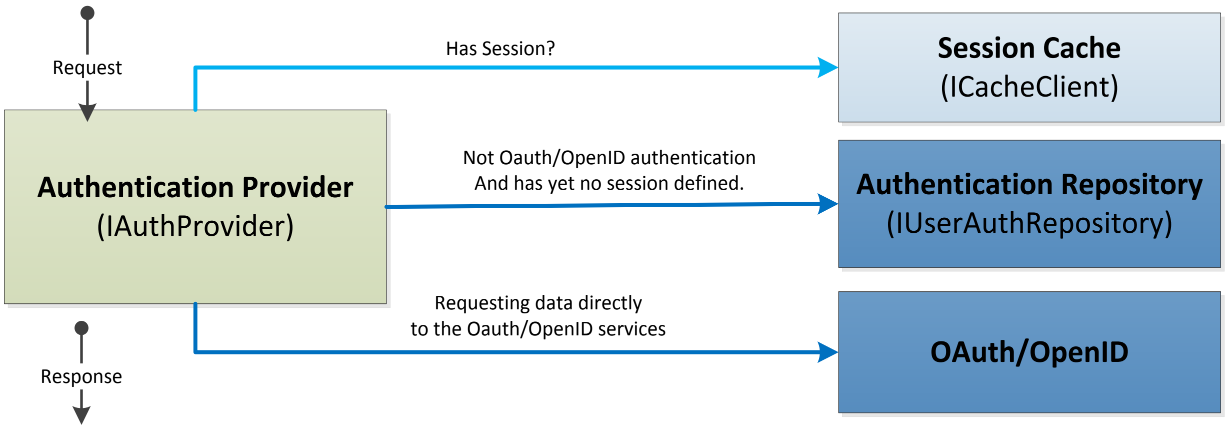


Figure 10: Authentication provider

Where do we start? As we have already seen several times, everything starts in the application host configuration where we need to enable the authentication feature. Before going into details and explaining all of the aforementioned components, let’s take a look at the minimum configuration needed in order to run the **`Basic Authentication`** with the **`OrmLite Repository`**.

```
public override void Configure(Container container)
{
    //1. Registering the authorization provider.
    Plugins.Add(new AuthFeature(() => new AuthUserSession(),
                                new IAuthProvider[]
                                    {
                                        new BasicAuthProvider()
                                    }));

    //2. Enabling the /register service.
    Plugins.Add(new RegistrationFeature());

    //3. Configuring the repository that uses a SQL Server back-end.
    var connString = "Data Source=;Initial Catalog=;User ID=;password=";

    var factory = new OrmLiteConnectionFactory(
        connString,
        SqlServerOrmLiteDialectProvider.Instance);

    var ormLiteRepository = new OrmLiteAuthRepository(factory);

    //Registering the repository.
    container.Register<IUserAuthRepository>(ormLiteRepository);

    //Should be run only once as this creates the necessary tables in the db.
    //There is a possibility to DropAndReCreate the entire table.
    ormLiteRepository.CreateMissingTables();
    //ormLiteRepository.DropAndReCreateTables();
    //Just for this example, we create a new user in code.
    if (ormLiteRepository.GetUserAuthByUserName("johnd") == null)
    {
        ormLiteRepository.CreateUserAuth(new UserAuth
            {
                UserName = "johnd",
                FirstName = "John",
                LastName = "Doe",
                Roles = new List<string> { RoleNames.Admin }
            }, "mypassword");
    }

    //4. Registering the session cache.
    container.Register<ICacheClient>(new MemoryCacheClient());
}

[Authenticate]
public class ProductService : ServiceStack.ServiceInterface.Service {…}
```

By using the **Plugins.Add(new AuthFeature()…)**, we are instructing ServiceStack to enable the authentication feature. This is where the session (**AuthUserSession**) and the providers (**IAuthProvider[]**) are registered as shown in the following code example.

```
Plugins.Add(new AuthFeature(() => new AuthUserSession(),
                                  new IAuthProvider[]
                                    {
                                        new BasicAuthProvider()
                                        //new DigestAuthProvider()
                                        //new CredentialsAuthProvider()
                                    }));
```

Authenticate Attribute

For ServiceStack to know which service requires authentication, it has to be marked with the **Authenticate** attribute.

```
[Authenticate (ApplyTo.Get | ApplyTo.Post]
public class ProductService : ServiceStack.ServiceInterface.Service
{
    //..
}
```

By default, all of the verbs of the service marked with the **Authenticate** attribute require authentication. However, it is possible to refine the control of what verb needs authentication by specifying the list of verbs directly in the attribute declaration (e.g., **ApplyTo.Get**).

OrmLiteRepository

`OrmLiteRepository` is the built-in authentication repository that uses the database as the storage for the authentication. The repository needs the **`OrmLiteConnectionFactory`** object which simply defines the connection string and the actual type of the database being used.

If instructed to do so, the repository will create all of the necessary tables in the database when calling the **`CreateMissingTables`** method. It is also possible to use **`DropAndReCreateTables`**. Both options are very useful when doing unit testing or on the application's first run, but I would suggest you avoid using this in a production system.

In order for the example to be realistic, we create a simple user so that when the application starts, we will have at least one user in the system that will be used to test the authentication.

Authentication Providers

Authentication providers constitute the heart of the authentication functionality; they define the actual type of the authentication to be used. The authentication provider communicates with the authentication repository in order to retrieve the information about the user or it communicates directly with the cache in order to retrieve information about the cached session.

As shown in the following class diagram, the main interface from which every provider inherits is the **`IAuthProvider`**, and the abstract base class that implements this interface is **`AuthProvider`**.

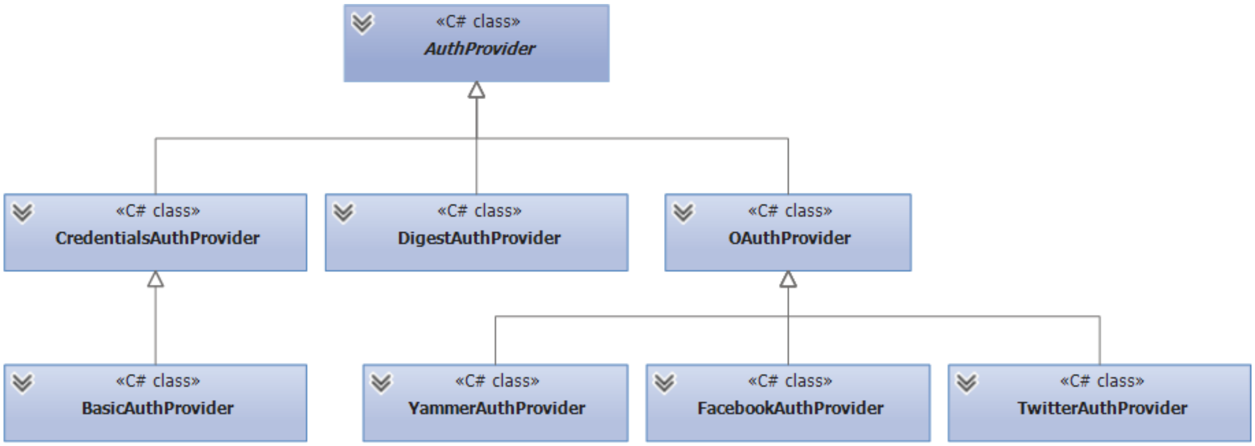


Figure 11: Authentication providers' hierarchy

ServiceStack currently supports the following providers.

Table 11: Supported authentication providers

Provider	Description
Basic authentication	Implemented as **`BasicAuthProvider`** class. It offers the standard basic authentication mechanism. Basic authentication relies on the user authentication repository (**`IUserAuthRepository`**) to retrieve the existing users.
Digest authentication	Implemented as **`DigestAuthProvider`** class. Digest authentication relies on the user authentication repository (**`IUserAuthRepository`**) to retrieve the existing users.
Credentials authentication	**`CredentialsAuthProvider`** provides a mechanism for authenticating with a username and password by posting the request to the **`/auth/credentials`** service. It relies on the user authentication repository (**`IUserAuthRepository`**) to retrieve the existing users.
OAuth[26] providers	ServiceStack implements Twitter, Facebook, and Yammer OAuth, which is a standardized open protocol to allow secure authorization from web, mobile, and desktop applications.
OpenID[27] providers	As a separate download from NuGet, the package **`ServiceStack.Authentication.OpenId`** has the providers for the OpenID protocol for: • **Google OpenID**: Allow users to authenticate with Google. • **Yahoo OpenID**: Allow users to authenticate with Yahoo. • **MyOpenID**: Allow users to authenticate with MyOpenID. • **OpenId**: Allow users to Authenticate with any custom OpenID provider.

[26] For more information about OAuth, visit http://oauth.net.

[27] For more information about OpenID, visit http://openid.net.

Extensibility

By inheriting from the **`CredentialsAuthProvider`** class, ServiceStack offers a great extensibility point for creating a customized authentication provider based on the username and password. This is as easy as overriding the **`TryAuthenticate`** method.

```
public class MyCustomAuthorizationProvider : CredentialsAuthProvider
{
    public override bool TryAuthenticate(IServiceBase authService,
                                          string userName, string password)
    {
        //Your implementation here.
    }
}
```

User Authentication Repository

Authentication repositories are used to manage users with your back-end database or other resource management systems. Through the repository, the authentication mechanism will be able to read, write, or authenticate the user against the repository. **`IUserAuthRepository`** is the base interface which the repository has to implement. The system already provides several implementations of repositories as shown in the following class diagram: Redis, InMemory, and OrmLite which, on its own, supports several mainstream databases (Redis, SQL Server, Oracle, MySQL, etc.).

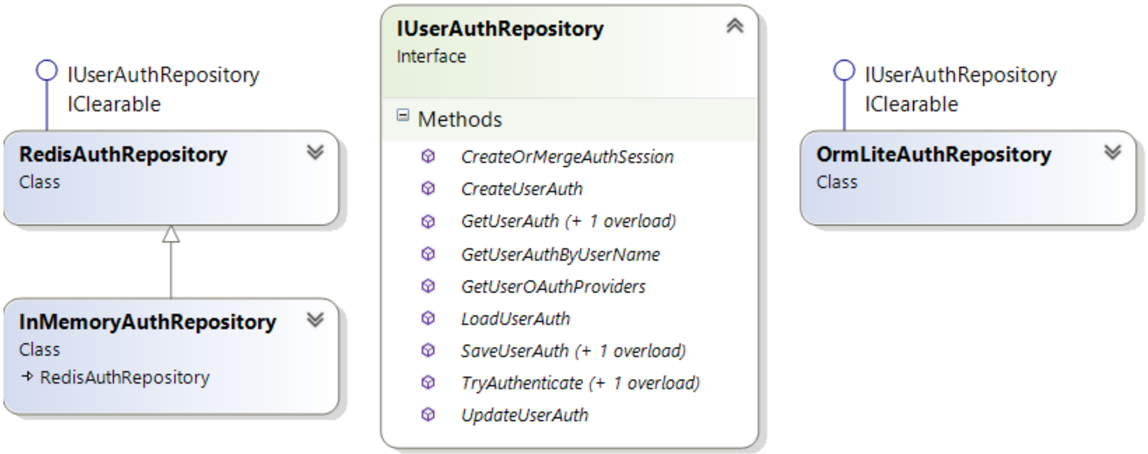


Figure 12: Authentication repository class diagram

The following table details a few other implementations that can be downloaded separately and are available on NuGet.

Table 12: Available authentication repositories

Repository (Database)	NuGet package
MongoDB	**ServiceStack.Authentication.MongoDB** through the class **MongoDBAuthRepository**
RavenDB	**ServiceStack.Authentication.RavenDB** through the class **RavenUserAuthRepository**
NHibernate	**ServiceStack.Authentication.NHibernate** through the class **NHibernateUserAuthRepository**

Authentication Session Cache

Once the authentication happens, the **AuthUserSession** model is populated and stored in the cache. The cache uses the standard ServiceStack caching mechanism through the **ICacheClient** interface. The cache is used to keep the information about the session of the currently authenticated user and it is managed internally by the ServiceStack framework.

How Does It Work?

After the successful authentication, the client will receive a cookie with a Session ID. The information stored in the cookie in the second request will be used to retrieve the correct session information from the **ICacheClient**. All this is done internally by ServiceStack. If we would like to check what the session currently contains, this information can be accessed by using the **Service.GetSession()** method as shown in the following code.

```
[Authenticate]
public class SomeService : ServiceStack.ServiceInterface.Service
{
   public object Get(ServiceRequest request)
   {
     IAuthSession session = this.GetSession();
     return new ServiceResponse() {Message = "You're " + session.FirstName };
   }
}
```

Services

There are mainly four services exposed that are related to the authentication, and as shown in the following figure, all inherit from the **`Service`** class which makes those very ordinary ServiceStack services.

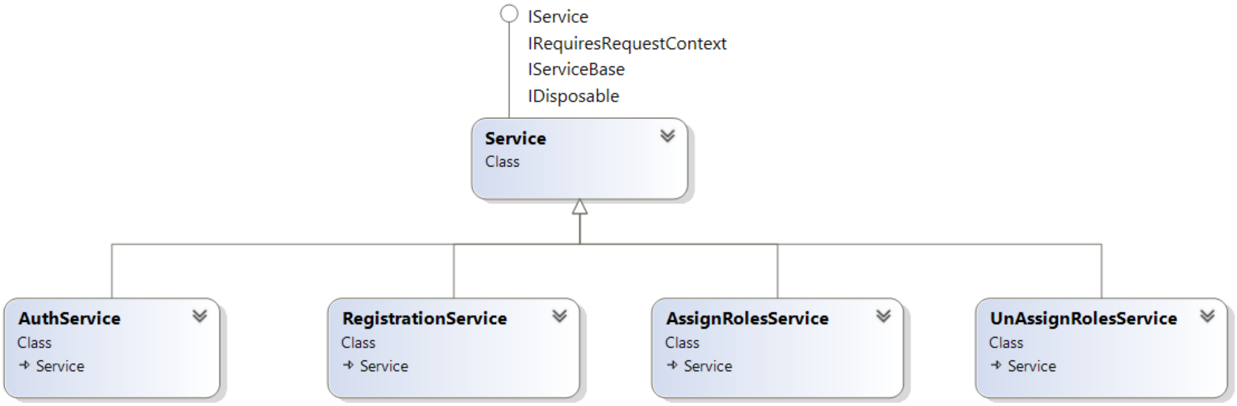


Figure 13: Authentication services

The following table contains the list of available services with their description and URI.

Table 13: Authentication services explained

Service	Exposed Path	Description
`AuthService`	`/auth/{provider}`	Executes the authentication and returns the currently logged user. The **`{Provider}`** parameter defines the name of the provider, and it can be one of the following.

URI	Provider Class
`/auth/credentials`	**`CredentialsAuthProvider`**
`/auth/basic`	**`BasicAuthProvider`**
`/auth/twitter`	**`TwitterAuthProvider`**
`/auth/facebook`	**`FacebookAuthProvider`**
`/auth/yammer`	**`YammerAuthProvider`**
`/auth/googleopenid`	**`GoogleOpenIdOAuthProvider`**
`/auth/yahooopenid`	**`YahooOpenIdOAuthProvider`**
`/auth/myopenid`	**`MyOpenIdOAuthProvider`**
`/auth/openid`	**`OpenIdOAuthProvider`** or any custom OpenID provider

Service	Exposed Path	Description
`AssignRolesService`	`/assignroles`	Assigns roles to an already existing user.
`UnAssignRolesService`	`/unassignroles`	Removes the roles from an already existing user.
`RegistrationService`	`/register`	Enables the creation of new users.

The registration service can be enabled or disabled through the application host by configuring the **Registration** feature.

```
Plugins.Add(new RegistrationFeature());
```

Authenticating the User

The following unit test shows how to authenticate an existing user and get the data from the **`ProductService`**. In this example, we will see how to use RestSharp (which is just another REST client library) in order to get the **`ProductResponse`**.

```
[TestMethod]
public void GetProductAndAuthenticateByUsingRestClient()
{
  var client = new RestClient("http://localhost:50712");
  client.Authenticator = new HttpBasicAuthenticator("johnd", "mypassword");

    var request = new RestRequest("products/1", Method.GET);

    var productResponse = client.Get<ProductResponse>(request);

    Assert.IsTrue(productResponse != null);
    Assert.IsTrue(productResponse.Data.Id == 1);
}
```

The following test method is doing exactly the same thing but using the ServiceStack client.

```
[TestMethod]
public void GetProductAndAuthenticateByUsingServiceStackClient()
{
    //ARRANGE ---
    var client = new JsonServiceClient("http://localhost:50712/" );
    client.UserName = "johnd";
    client.Password = "mypassword";

    //ACT  ------
    var product = client.Get<ProductResponse>("/products/1");

    //ASSERT ----
    Assert.IsTrue(product != null);
}
```

Basic Authentication with jQuery

Calling the service from jQuery is not that different from calling it from C#. By using the **`$.ajax`** function, it is possible to set the **`Authorization`** header and send the username and password.

For the full implementation, check the **ProductData.html** file provided in the sample project, available at https://bitbucket.org/syncfusiontech/servicestack-succinctly.

```
<script type="text/javascript">
 function make_base_auth(user, password) {
     var tok = user + ':' + password;
     var hash = btoa(tok);
     return "Basic " + hash;
 }

 function authenticate() {
     //Getting the username and password.
     var username = $("input#username").val();
     var password = $("input#password").val();

      $.ajax({
         type: "GET",
         url: "/products/1",
         async: false,
         dataType: 'json',
         beforeSend: function (xhr) {
         xhr.setRequestHeader('Authorization', make_base_auth(username,
                                                              password));
         },
         success: function (result) { //Displaying the values.
             $('#product_id').val(result.Id);
             $('#product_name').val(result.Name);
         },
     });
 }
</script>
```

Registering a New User

By using the **`RegistrationService`**, it is possible to register new users through the **`/register`** URI. The following unit test shows how to register a new user. The response of the **`POST`** action is a custom **`UserResponse`** which returns the new **`UserId`**, the current **`SessionId`**, and the **`UserName`** that registered the user—in this case, “johnd”.

```
[TestMethod]
public void RegisterNewUser()
{
    var client = new RestClient("http://localhost:50712");
    client.Authenticator = new HttpBasicAuthenticator("johnd", "mypassword");

    var request = new RestRequest("register", Method.POST);
    request.RequestFormat = DataFormat.Json;
    request.AddBody(new
        {
            UserName = "JaneR",
            FirstName = "Jane",
            LastName = "Roe",
            DisplayName = "Jane Roe",
            Email = "jane.roe@email.com",
            Password = "somepassword",
            AutoLogin = true
        });
    var response = client.Post<UserResponse>(request);

    Assert.IsTrue(response != null);
    Assert.IsTrue(response.Data.UserId != null);
}

public class UserResponse
{
    public string UserId { get; set; }
    public string SessionId { get; set; }
    public string UserName { get; set; }
}
```

Assigning Roles to an Existing User

By using the **`AssignRolesService`**, it is possible to change the **`Roles`** and **`Permissions`** of an existing user through the **`/assignroles`** URI as shown in the following unit test.

```
[TestMethod]
public void AssignRoles()
{
    var client = new RestClient("http://localhost:50712");
    client.Authenticator = new HttpBasicAuthenticator("johnd", "mypassword");

    var request = new RestRequest("assignroles", Method.POST);
    request.RequestFormat = DataFormat.Json;
    request.AddBody(new
        {
            UserName = "JaneR",
            Permissions = "some_permissions",
            Roles = "Admin, Reader"
        });

    var response = client.Post<AssignRoleResponse>(request);

    Assert.IsTrue(response != null);
}

public class AssignRoleResponse
{
    public string AllRoles { get; set; }
    public string AllPermissions { get; set; }
    public object ResponseStatus { get; set; }
}
```

Note: As unassigning roles follow the same structure, the actual code example is omitted.

Tip: The user that assigns or unassigns roles has to be an Admin.

Authorization

Together with authentication, ServiceStack also offers an authorization mechanism. There is a way to control which **`Role`** or **`Permission`** is allowed to execute a certain service or method.

We have seen that it is possible to register a user and, consequently, to assign roles and permissions. Therefore, whenever the user authenticates, all of this information will be available in the session.

How to Control the Access to the Service

There are mainly two attributes to look into: **`RequiredRole`** and **`RequiredPermission`**.

The attributes can be associated with:

- The service itself.
- Any service method.
- The Request DTO declaration.

```
[Authenticate]
public class ProductService : ServiceStack.ServiceInterface.Service
{
    [RequiredRole(RoleNames.Admin)]
    [RequiredPermission("some_permission")]
    public ProductResponse Get(GetProduct request) { … }

    //To execute this method a user has to be authenticated.
    public List<ProductResponse> Get(GetProducts request) { … }
}

//This is the equivalent declaration.
[Authenticate]
[RequiredRole(RoleNames.Admin)]
[RequiredPermission("some_permission")]
public class GetProduct { … }
```

```
[Authenticate]
public class GetProducts { … }
```

In case the user doesn’t have the correct **`Role`** or **`Permission`** assigned, the **`403 Invalid Role`** or **`403 Invalid Permission`** statuses will be returned.

Chapter 8 Caching

In Chapter 1, we briefly mentioned that ServiceStack contains built-in support for server-side caching.

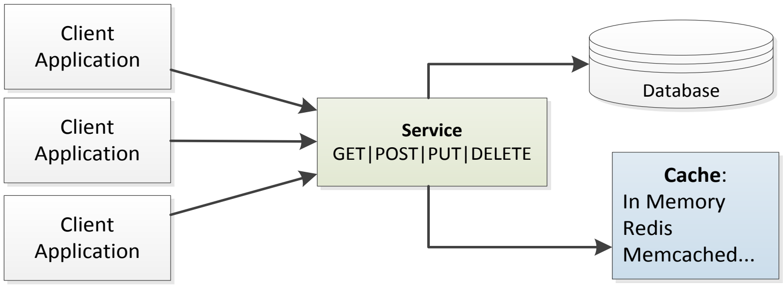


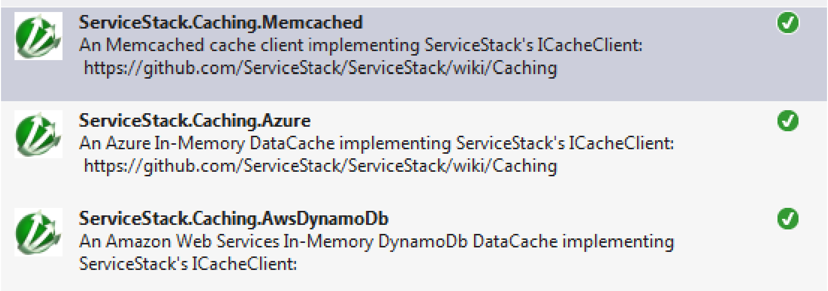
Figure 14: Caching in ServiceStack

ServiceStack provides **ICacheClient**, a clean caching interface, for a number of different cache providers:

- **In Memory:** Uses RAM as the caching mechanism.
- **Redis**: An open-source, BSD-licensed, advanced key-value store.[28]
- **Memcached**: A high-performance, distributed memory, object-caching system intended for use in speeding up dynamic web applications by alleviating database load.[29]
- **Azure Client**: Used to interface with Microsoft Azure.
- **AWS Cache Client**: Used to interface with Amazon's DynamoDB back end hosted on Amazon Web Services.
- **Disk**: Writes to the hard disk.

Which cache provider you use depends mainly on the needs of your application.

MemCached, Azure, or AwsDynamoDb providers need to be installed separately through NuGet.



ServiceStack.Caching.Memcached ✓
An Memcached cache client implementing ServiceStack's ICacheClient:
https://github.com/ServiceStack/ServiceStack/wiki/Caching

ServiceStack.Caching.Azure ✓
An Azure In-Memory DataCache implementing ServiceStack's ICacheClient:
https://github.com/ServiceStack/ServiceStack/wiki/Caching

ServiceStack.Caching.AwsDynamoDb ✓
An Amazon Web Services In-Memory DynamoDb DataCache implementing ServiceStack's ICacheClient:

Figure 15: NuGet caching providers

[28] The Redis website: http://redis.io
[29] The Memcached website: http://memcached.org

Configuring Caching

The **ICacheClient** interface is the starting point for all of the supported caching options as it defines the basic interface that every cache provider has to implement.

To start using the caching mechanism, it is as simple as registering the provider in the application host. Once registered, an instance of the **ICacheClient** will be injected to the service at run time, and will be available as a **Cache** property, which is part of the **ServiceBase** class.

The following example shows how to configure the various providers in the application host.

```
public override void Configure(Container container)
{
    //MemoryCacheClient
    container.Register<ICacheClient>(new MemoryCacheClient());

    //Redis
    container.Register<IRedisClientsManager>
                 (c => new PooledRedisClientManager("<redis_address_here>"));
    container.Register
                 (c => c.Resolve<IRedisClientsManager>().GetCacheClient())
                                              .ReusedWithin(ReuseScope.None);

    //MemCached
    container.Register<ICacheClient>
                      (new MemcachedClientCache(new[] {"<memcached_host>"}));

    //Azure
    container.Register<ICacheClient>(new AzureCacheClient("CacheName"));

    //AWS DynamoDB
    container.Register<ICacheClient>(new DynamoDbCacheClient(...))
}
```

Response Caching

To demonstrate how to use caching, we will reuse our **ProductService**. One important thing to note here is that the cache will contain the full response rather than the Response DTO itself.

In order to support this scenario, we have to change the output of the method to **object** instead of the existing **ProductResponse**. We should know two things about caching:

- Cache is basically a dictionary and it needs a unique key to identify a cached object. The key is a string and it can be generated by the **UrnId** helper class.
- **RequestContext.ToOptimizedResultUsingCache** is the method responsible for returning data from the cache.

The following code example contains the modified **Get()** method. The highlighted code will be executed in case nothing is found in the cache with the given key (this always happens the first time a particular item is requested from the cache) and, for all the subsequent calls, it will be ignored.

```
public object Get(GetProduct request)
{
    var key = UrnId.Create<GetProduct>(request.Id.ToString());

    var cachedProduct =
        RequestContext.ToOptimizedResultUsingCache(this.Cache, key, () =>
        {
            var product = ProductRepository.GetById(request.Id);
            if (product == null)
            {
                Response.StatusCode = (int) HttpStatusCode.NotFound;
                return default(ProductResponse);
            }
            return ProductMapper.ToProductResponse(product);
        });
    return cachedProduct;
}
```

Caching with Time Expiration

The basic caching method works fine for static objects that we know won't change over a long period of time. But for items that can change, there is an overload of the method that supports time expiration through a **TimeSpan**, which defines a period after which the cached item will be automatically deleted and therefore refreshed in the next call.

```
RequestContext.ToOptimizedResultUsingCache(this.Cache, key, new TimeSpan(0,1,0),
    () =>
        {
            //…
        }
```

Cached Items Removal

However, the time expiration method still doesn't ensure that the object returned from the cache is actually the latest version stored in the repository. The object can, theoretically, be updated (changed) during the time frame when the object is supposed to be in the cache.

The way to fix this is to manually ensure that the object is refreshed in the cache in case the object is changed. One possible way of doing this would be to change the **Put()** (update) method to remove the changed objects from the cache.

```
public ProductResponse Put(UpdateProduct request)
{
    var key = UrnId.Create<GetProduct>(request.Id.ToString());
    base.RequestContext.RemoveFromCache(base.Cache, key);
}
```

Chapter 9 Logging

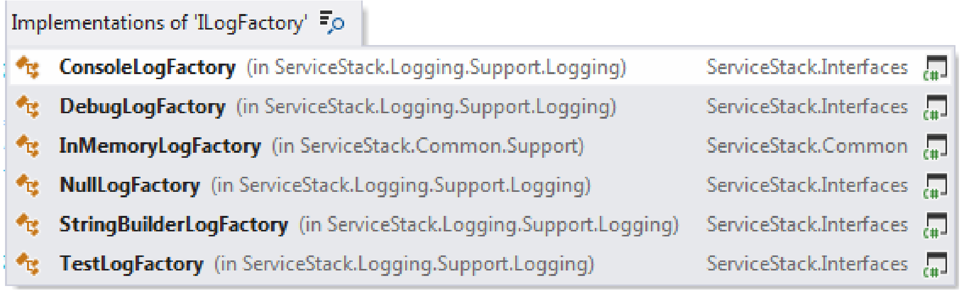
ServiceStack provides a logging framework that exposes the **`ILog`** interface and adapters for the major Microsoft .NET logging providers (log4net, NLog, etc.).

By depending on the **`ILog`** interface, the web service and the application logic are free from any direct dependency to any external logging library, which then can be injected at run time.

The following NuGet packages are available for download:

- ServiceStack.Logging.NLog
- ServiceStack.Logging.Elmah
- ServiceStack.Logging.Log4Net
- ServiceStack.Logging.Log4Netv129
- ServiceStack.Logging.EventLog

ServiceStack, however, contains some basic logger implementation as shown in the following figure.



Implementations of 'ILogFactory'

ConsoleLogFactory	(in ServiceStack.Logging.Support.Logging)	ServiceStack.Interfaces
DebugLogFactory	(in ServiceStack.Logging.Support.Logging)	ServiceStack.Interfaces
InMemoryLogFactory	(in ServiceStack.Common.Support)	ServiceStack.Common
NullLogFactory	(in ServiceStack.Logging.Support.Logging)	ServiceStack.Interfaces
StringBuilderLogFactory	(in ServiceStack.Logging.Support.Logging)	ServiceStack.Interfaces
TestLogFactory	(in ServiceStack.Logging.Support.Logging)	ServiceStack.Interfaces

Figure 16: Built-in logging factories

There are mainly three objects to look into:

- **`LogManager`**: The entry point for the logging configuration.
- **`ILog`**: Abstracts the provider by offering a generic interface.
- **`LogFactories`**: One per provider; instantiates a provider.

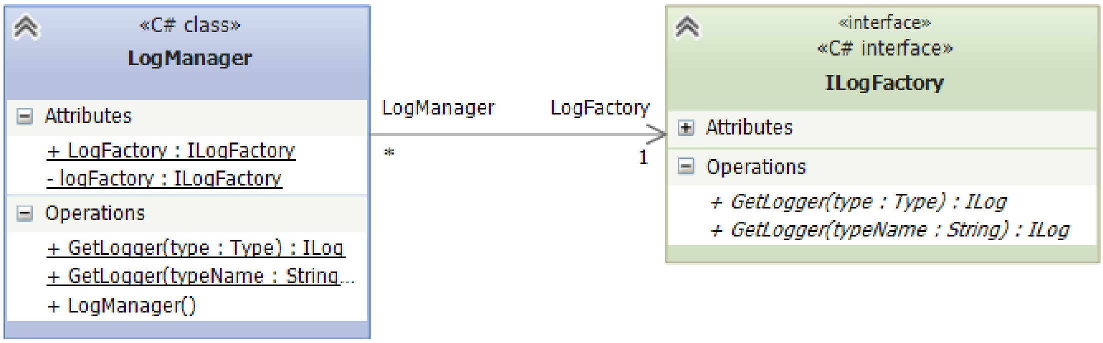

«C# class»
LogManager
Attributes
+ LogFactory : ILogFactory
- logFactory : ILogFactory
Operations
+ GetLogger(type : Type) : ILog
+ GetLogger(typeName : String...
+ LogManager()
LogManager
LogFactory
*
1
«interface»
«C# interface»
ILogFactory
Attributes
Operations
+ GetLogger(type : Type) : ILog
+ GetLogger(typeName : String) : ILog

Figure 17: Log classes

log4net Integration

log4net is probably the most often used logging library in the Microsoft .NET world. Many developers are familiar with its configuration and usage. In this chapter, we will see how to configure and start using the logging feature as well as how to use the log4net provider adapter.

To start using the log4net, the first thing to do is to install a package from NuGet: **`ServiceStack.Logging.Log4Net`**.

In the application host configuration, we need to register the fact that we want to use the log4net provider. This is done by creating a new instance of the **`Log4NetFactory`** and assigning the object to the **`LogManager.LogFactory`**.

```
public override void Configure(Container container)
{
    LogManager.LogFactory = new Log4NetFactory(true);
}
```

To get an instance of the logger that we just registered in the application host configuration, we will use the **`LogManager.GetLogger()`** method.

```
public class ProductService : ServiceStack.ServiceInterface.Service
{
    private ILog Logger { get; set; }

    public ProductService()
    {
        Logger = LogManager.GetLogger(GetType());
    }

    public ProductResponse Get(GetProduct request)
    {
        Logger.Debug("Getting the product");

        return new ProductResponse();
    }
}
```

Using the ILog Interface

If you think that instantiating the logger in every service is an overhead, then you can use the dependency injection and inject the logger to the class. In this case, we would need to change the previous example to include the registration of the **`ILog`** interface.

```
public override void Configure(Container container)
{
    LogManager.LogFactory = new Log4NetFactory(true);
    container.Register<ILog>(LogManager.GetLogger(""));
}
```

By changing the **ILog** (logger) property to become public, the IoC container will inject the instance at service creation.

```
public class ProductService : ServiceStack.ServiceInterface.Service
{
    public ILog Logger { get; set; }

    public ProductResponse Get(GetProduct request)
    {
        Logger.Debug("Getting the product");

        return new ProductResponse();
    }
}
```

RequestLogsFeature Plug-in

In addition to the aforementioned possibility of enabling the logging by exposing the **ILog** interface, ServiceStack contains a built-in plug-in that enables configurable, in-memory tracking of recent requests and error responses.

To enable the plug-in, we have to register the **RequestLogsFeature** by adding it to the plug-ins list in the application host.

```
public override void Configure(Container container)
{
    Plugins.Add(new RequestLogsFeature()
    {
        RequiredRoles = new string[]{}
    });
}
```

After starting the application, a new URI **http://<servername>/requestlogs** will be registered and immediately available for querying.

The plug-in exposes a list of configurable properties that can be changed and that affect what is logged.

Property	Description
`AtRestPath` Default value: /requestlogs	The location at which the logging output can be queried. Being a configurable option, one can set a new location (e.g., "**`/lastrequests`**").
`EnableSessionTracking` Default value: False	Tracks the information about the current user session. The log will contain the information exposed by the implementor of the **`IAuthSession`** mentioned in Chapter 7.
`EnableRequestBodyTracking` Default value: False	Enables the tracking of the raw body of the request (as in the request header).
`EnableResponseTracking` Default value: False	Enables the tracking of the returned object (result of the service call).
`EnableErrorTracking` Default value: True	Enables the tracking of the errors (result of the service call).
`Capacity` Default value: 1000	Defines a maximum number of (latest) requests to be tracked. Being configurable, this can be changed to a higher or lower value as defined in **`InMemoryRollingRequestLogger`**.
`RequiredRoles` Default value: "Admin"	Contains a list of roles that are allowed to query the URI. Being a collection, multiple roles can be configured. If left empty, everyone will be able to access the URI.
`RequestLogger` Default value: Instance of InMemoryRollingRequestLogger	Contains an instance of a system-defined logger that implements the **`IRequestLogger`** interface. It is possible to create your own implementation of the **`IRequestLogger`**. The system default's configured logger is **`InMemoryRollingRequestLogger`**.

Property	Description
ExcludeRequestDtoTypes Default value: typeof (RequestLogs)	Contains a list of DTOs for which we don’t want to track the logging.
HideRequestBodyForRequestDtoTypes Default value: typeof (Auth), typeof (Registration)	Request body of the types specified in the list won’t be tracked.

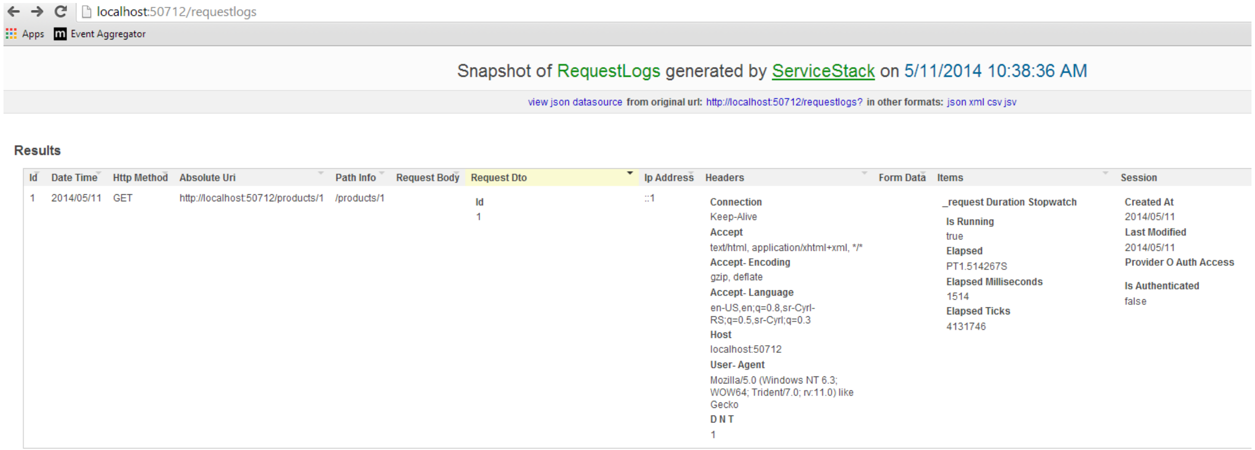
InMemoryRollingRequestLogger

InMemoryRollingRequestLogger is a built-in class that implements the **IRequestLogger** interface (**IRequestLogger** interface is only used in the **RequestLogs** plug-in). As its name suggests, the logger implements the tracking and stores the values in memory as **RequestLogEntry** entries.

The list of **RequestLogEntry** values is returned in the **/requestlog** URI. The **RequestLogEntry** class is defined as follows.

```
public class RequestLogEntry
{
    public long Id { get; set; }
    public DateTime DateTime { get; set; }
    public string HttpMethod { get; set; }
    public string AbsoluteUri { get; set; }
    public string PathInfo { get; set; }
    public string RequestBody { get; set; }
    public object RequestDto { get; set; }
    public string UserAuthId { get; set; }
    public string SessionId { get; set; }
    public string IpAddress { get; set; }
    public string ForwardedFor { get; set; }
    public string Referer { get; set; }
    public Dictionary<string, string> Headers { get; set; }
    public Dictionary<string, string> FormData { get; set; }
    public Dictionary<string, object> Items { get; set; }
    public object Session { get; set; }
    public object ResponseDto { get; set; }
    public object ErrorResponse { get; set; }
    public TimeSpan RequestDuration { get; set; }
}
```

And, when displayed in the browser, it looks like the following figure.



Snapshot of RequestLogs generated by ServiceStack on 5/11/2014 10:38:36 AM

view json datasource from original url: http://localhost:50712/requestlogs? in other formats: json xml csv jsv

Results

Id	Date Time	Http Method	Absolute Uri	Path Info	Request Body	Request Dto	Ip Address	Headers	Form Data	Items	Session
1	2014/05/11	GET	http://localhost:50712/products/1	/products/1		**Id** 1	::1	**Connection** Keep-Alive **Accept** text/html, application/xhtml+xml, */* **Accept- Encoding** gzip, deflate **Accept- Language** en-US,en;q=0.8,sr-Cyrl-RS;q=0.5,sr-Cyrl;q=0.3 **Host** localhost:50712 **User- Agent** Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko **D N T** 1		**_request Duration Stopwatch** **Is Running** true **Elapsed** PT1.514267S **Elapsed Milliseconds** 1514 **Elapsed Ticks** 4131746	**Created At** 2014/05/11 **Last Modified** 2014/05/11 **Provider O Auth Access** **Is Authenticated** false

Figure 18: RequestLogs result screen

Chapter 10 Profiling

Another built-in feature of ServiceStack is a profiler. The most common use of profiling information is to aid in finding bottlenecks and, obviously, in helping with application optimization. With a profiler, we are able to find out the time needed to perform a specific action or a sequence of actions.

Internally, ServiceStack uses a slightly changed version of the MVC MiniProfiler,[30] which is adapted for ServiceStack's needs. The MiniProfiler on its own provides support for OrmLite, which enables you to see the timing for actual database queries. We will see this later.

Start Using the Profiler

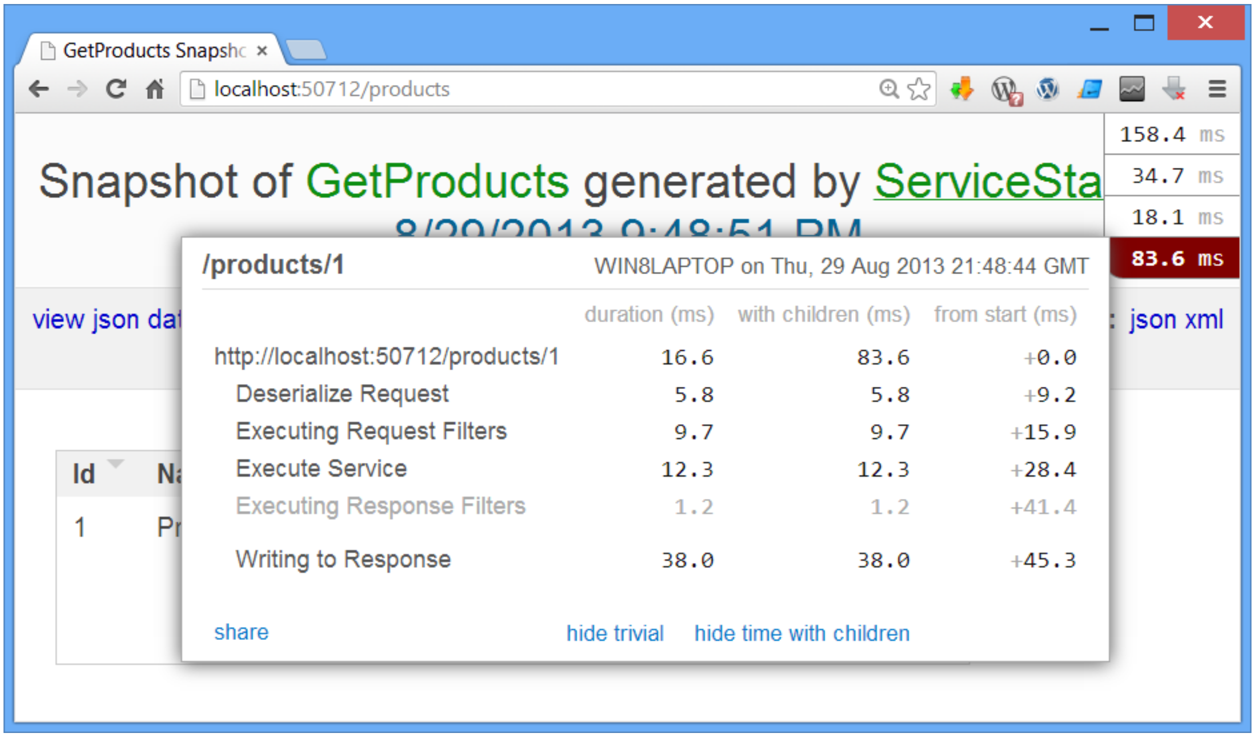
It is easy to start using the profiler. The only thing that needs to be done is to **`Start`** the profiler for each request and to **`Stop`** it when the request ends. In order to limit the profiler output, we may specify to start the profiler only if the request is made from the same location (typically, we need profiling when debugging or during the development).

The following code example shows how to start and stop the profiler in the **Global.asax.cs** file.

```
protected void Application_BeginRequest(object sender, EventArgs e)
{
    if (Request.IsLocal)
    {
        Profiler.Start();
    }
}

protected void Application_EndRequest(object sender, EventArgs e)
{
    Profiler.Stop();
}
```

[30] MiniProfiler website: http://miniprofiler.com



Snapshot of GetProducts generated by ServiceSta
8/29/2013 9:48:51 PM

/products/1 WIN8LAPTOP on Thu, 29 Aug 2013 21:48:44 GMT

	duration (ms)	with children (ms)	from start (ms)
http://localhost:50712/products/1	16.6	83.6	+0.0
Deserialize Request	5.8	5.8	+9.2
Executing Request Filters	9.7	9.7	+15.9
Execute Service	12.3	12.3	+28.4
Executing Response Filters	1.2	1.2	+41.4
Writing to Response	38.0	38.0	+45.3

Figure 19: Profiler

When running the application in the browser, we can directly see (and expand in the top right corner) the statistics related to the current request.

Very interesting statistics can be seen regarding the serialization, deserialization, and the actual execution time of the **Service** method itself.

Profiling OrmLite Queries

It is possible to profile the actual queries performed by the OrmLite provider.

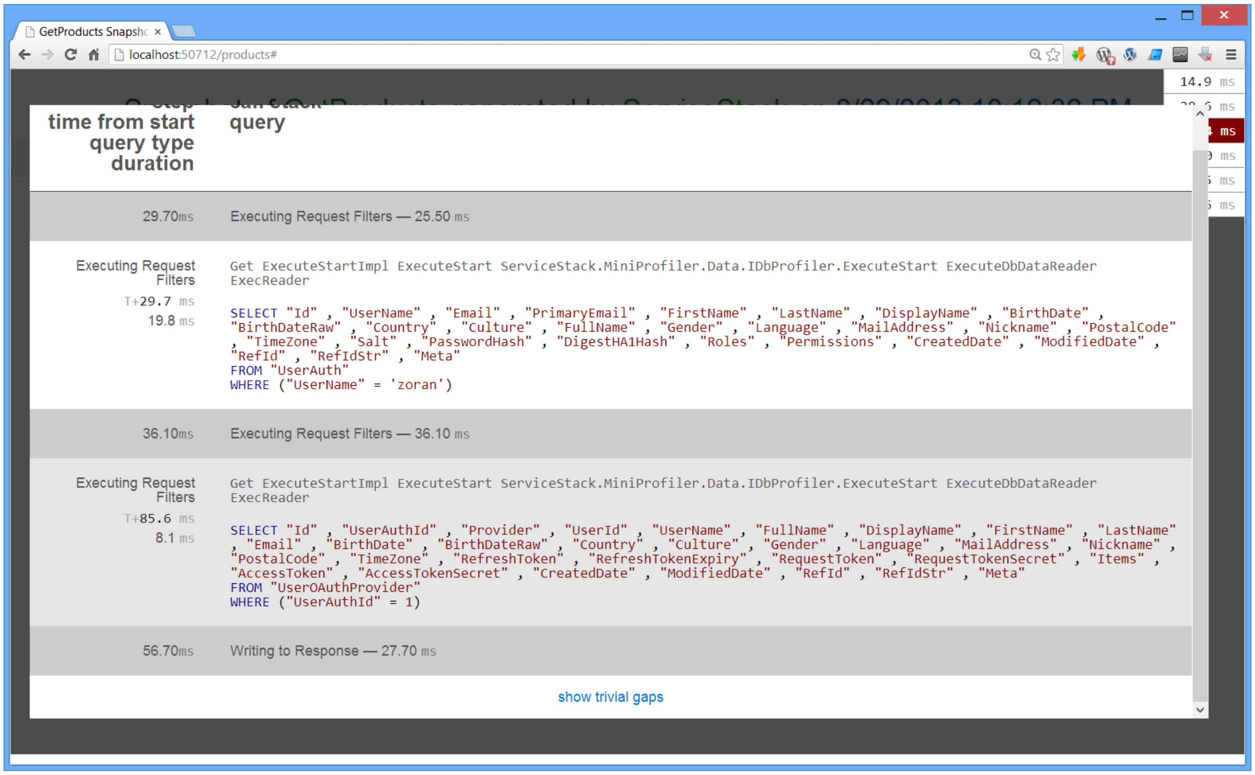
In order to do this, we need to provide the **ConnectionFilter** which will enable the **Profiler** to be attached to the query execution.

The following code shows how to instantiate a new **OrmLiteConnectionFactory**.

```
var connString = "Data Source=…;Initial Catalog=..;User ID=..;password=..";

var factory = new OrmLiteConnectionFactory(
    connString, SqlServerOrmLiteDialectProvider.Instance)
    {
        ConnectionFilter = x => new ProfiledDbConnection(x, Profiler.Current)
    };
```

When executing the query, we can see in the browser something similar to the following figure, where we can see the timing for each of the SQL queries executed.



```
time from start     query
     query type
       duration

        29.70ms     Executing Request Filters — 25.50 ms

Executing Request   Get ExecuteStartImpl ExecuteStart ServiceStack.MiniProfiler.Data.IDbProfiler.ExecuteStart ExecuteDbDataReader
          Filters   ExecReader
       T+29.7 ms
         19.8 ms    SELECT "Id" , "UserName" , "Email" , "PrimaryEmail" , "FirstName" , "LastName" , "DisplayName" , "BirthDate" ,
                    "BirthDateRaw" , "Country" , "Culture" , "FullName" , "Gender" , "Language" , "MailAddress" , "Nickname" , "PostalCode"
                    , "TimeZone" , "Salt" , "PasswordHash" , "DigestHA1Hash" , "Roles" , "Permissions" , "CreatedDate" , "ModifiedDate" ,
                    "RefId" , "RefIdStr" , "Meta"
                    FROM "UserAuth"
                    WHERE ("UserName" = 'zoran')

        36.10ms     Executing Request Filters — 36.10 ms

Executing Request   Get ExecuteStartImpl ExecuteStart ServiceStack.MiniProfiler.Data.IDbProfiler.ExecuteStart ExecuteDbDataReader
          Filters   ExecReader
       T+85.6 ms
          8.1 ms    SELECT "Id" , "UserAuthId" , "Provider" , "UserId" , "UserName" , "FullName" , "DisplayName" , "FirstName" , "LastName"
                    , "Email" , "BirthDate" , "BirthDateRaw" , "Country" , "Culture" , "Gender" , "Language" , "MailAddress" , "Nickname" ,
                    "PostalCode" , "TimeZone" , "RefreshToken" , "RefreshTokenExpiry" , "RequestToken" , "RequestTokenSecret" , "Items" ,
                    "AccessToken" , "AccessTokenSecret" , "CreatedDate" , "ModifiedDate" , "RefId" , "RefIdStr" , "Meta"
                    FROM "UserOAuthProvider"
                    WHERE ("UserAuthId" = 1)

        56.70ms     Writing to Response — 27.70 ms

                              show trivial gaps
```

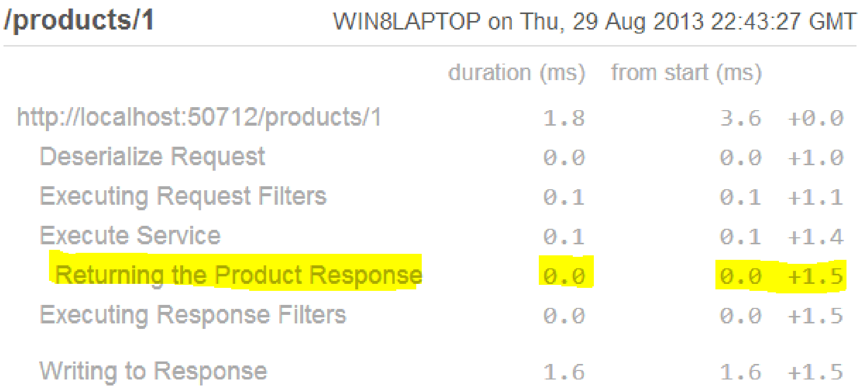
Figure 20: Profile SQL queries

Defining Custom Steps

It is also possible to define our own steps by giving a label which then will be displayed in the output that we have already seen. To do so, we need to use the **`Profiler.Step()`** method as in the following code example.

```
public ProductResponse Get(GetProduct request)
{
    using (Profiler.Current.Step("Returning the Product Response"))
    {
        //....
    }
}
```

In the output, we can see that “Returning the product response” is shown with the execution time.



```
/products/1                         WIN8LAPTOP on Thu, 29 Aug 2013 22:43:27 GMT

                                         duration (ms)   from start (ms)

http://localhost:50712/products/1                  1.8              3.6  +0.0
  Deserialize Request                              0.0              0.0  +1.0
  Executing Request Filters                        0.1              0.1  +1.1
  Execute Service                                  0.1              0.1  +1.4
    Returning the Product Response                 0.0              0.0  +1.5
  Executing Response Filters                       0.0              0.0  +1.5

  Writing to Response                              1.6              1.6  +1.5
```

Figure 21: Profiling with custom steps

Chapter 11 Extending ServiceStack

ServiceStack offers a good level of extensibility. There are two ways of extending ServiceStack: through **filters** and through **plug-ins**.

Filters

Filters are a simple concept; they allow us to execute some code before or after a service executes. Filters can be seen as a cross-cutting concern and allow a great deal of reusability of code. For instance, monitoring, authorization, authentication, logging, transaction processing, and validation can be implemented by filters. This is a great way of keeping the service code as clean as possible without implementing these same concepts over and over again.

There are two types of filters: Request filters and Response filters. We have seen this when we mentioned the Request and Response pipeline. A particularity of a filter is that it can have a priority, which means that we can control the order in which filters are going to be executed.

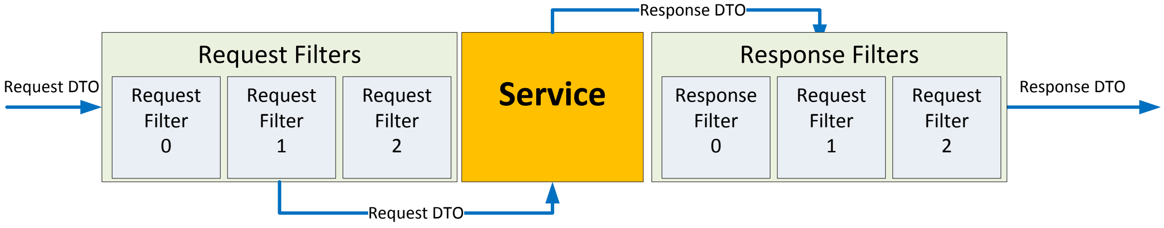


Figure 22: Request and Response filter ordering

Filters are applied as **attributes** to a Request and Response DTO or, alternatively, to the service itself. There are some predefined ServiceStack filters that are executed before the user-applied ones. This default behavior can be changed if the priority of a user filter is set to a negative number, in which case it gets executed beforehand.

In the example, we will see how to create a Request and a Response filter and how to apply it to the service. The Request filter will count how many times the GET method has been called, and the Response filter will record how many successful requests have been processed. This recorded information will be exposed through a service called **`StatsCounterService`**.

Request Filters

The Request filters are executed before the service gets called. There are two ways of declaring the new Request filter, either by implementing the **IHasRequestFilter** interface or by using a **RequestFilterAttribute** base class. Using the **RequestFilterAttribute** is simpler as it already implements the basic functionalities; we only need to override the **Execute()** method.

In the following example, we will create the **ServiceUsageStatsRequestFilter** class that counts the number of times the **Get()** method is called. To store the data, we will use the **MemoryCacheClient** (**ICacheClient**) that will be injected at run time, hence the property called **Cache**. The filter also exposes the **ServiceName** property that should be specified every time the attribute is assigned to a service.

```
public class ServiceUsageStatsRequestFilter : RequestFilterAttribute
{
    public string ServiceName { get; set; }
    //This property will be resolved by the IoC container.
    public ICacheClient Cache { get; set; }

    public override void Execute(IHttpRequest req, IHttpResponse res,
                                                   object requestDto)
    {
        string getKey = ServiceName + "_GET";

        if (req.HttpMethod == "GET")
        {
            var item = Cache.Get<long>(getKey);
            Cache.Set(getKey, (item + 1));
        }
    }
}
```

Response Filters

The Response filters are executed after the service has been executed. As for the Request filter, there are two ways of declaring the new Response filter: either by implementing the **IHasResponseFilter** interface or by using a **ResponseFilterAttribute**-derived attribute.

In the following example, we will create the **ServiceUsageStatsRequestFilter** class that counts the number of times a response status has been in the range of 200—which would generally mean that the call has been successful.

As for the Request filter, the same logic applies when it comes to the **Cache** and **ServiceName** declaration.

```
public class ServiceUsageStatsResponseFilter: ResponseFilterAttribute
{
    public string ServiceName { get; set; }
    //This property will be resolved by the IoC container.
    public ICacheClient Cache { get; set; }

    public override void Execute(IHttpRequest req, IHttpResponse res,
                                                        object responseDto)
    {
        string successKey = ServiceName + "_SUCCESSFUL";
        if (res.StatusCode >= 200 && res.StatusCode < 300)
        {
            var item = Cache.Get<long>(successKey);
            Cache.Set(successKey, item + 1);
        }
    }
}
```

Applying Filters

In order to enable the filters, we have to add the attributes to the service itself by applying the attributes as follows.

```
[ServiceUsageStatsRequestFilter(ServiceName = "ProductService")]
[ServiceUsageStatsResponseFilter(ServiceName = "ProductService")]
public class ProductService : ServiceStack.ServiceInterface.Service
{
    …
}
```

In this case, by default, filters will be called for every method call (GET, POST, PUT, etc.).

If we would like to limit the filter execution to only a certain method, we can use the **`ApplyTo`** switch.

```
[ServiceUsageStatsRequestFilter(ApplyTo = ApplyTo.Get | ApplyTo.Put,
ServiceName = "ProductService", Priority = 1)]
[ServiceUsageStatsResponseFilter(ServiceName = "ProductService")]
public class ProductService : ServiceStack.ServiceInterface.Service
{
    …
}
```

Another way of enabling a filter is by attaching it directly to the Request and Response DTO.

```
[ServiceUsageStatsRequestFilter(ServiceName = "ProductService")]
public class GetProduct
{
   public int Id { get; set; }
}

[ServiceUsageStatsResponseFilter(ServiceName = "ProductService")]
public class ProductResponse { … }
```

Service for Displaying Statistics

In order to give some meaning to the collected data, we can expose a service that will query the **Cache** where the data is stored. The service is simple. The following implementation is perhaps not elegant, but its purpose is to simply demonstrate the feature.

```
[Route("/Cache/{ServiceName}", "GET")]
public class GetCacheContent
{
    public string ServiceName { get; set; }
}

public class StatsItem
{
    public string ServiceName { get; set; }
    public long Value { get; set; }
}

public class StatsCounterService: ServiceStack.ServiceInterface.Service
{
    public ICacheClient Cache { get; set; }

    public object Get(GetCacheContent request)
    {
        var data =  Cache.GetAll<long>(new[]
            {
                request.ServiceName + "_GET",
                request.ServiceName + "_SUCCESSFUL",
            });

        List<StatsItem> items = new List<StatsItem>();
        data.Keys.ToList().ForEach(x => items.Add(new StatsItem()
            {
                ServiceName = x,
                Value = data[x]
            }));
        return items;
    }
}
```

When calling the service **GET http://localhost:50712/Cache/ProductService.json**, the following response will be returned.

```
[
   {   "ServiceName":"ProductService_GET", "Value": 1  },
   {   "ServiceName":"ProductService_SUCCESSFUL", "Value": 1 }
]
```

Plug-ins

As filters, plug-ins are used to extend ServiceStack. Unlike filters, they are meant to live within the request and response pipeline. Plug-ins can enhance the functionality of ServiceStack, for example, by enabling new content type handlers, adding new services, and automatically registering certain filters to be executed for each request.

The following is a list of predefined plug-ins already available in ServiceStack that can be enabled or disabled:

- Metadata feature: Provides the autogenerated pages for service inspection.
- CSV format: Provides automatic handling of the CSV content type.
- HTML format: Provides automatic handling of the HTML content type.
- Swagger integration: Swagger is a specification and a complete framework implementation for describing, producing, consuming, and visualizing RESTful web services. This will be covered in the Swagger Integration chapter.
- Validation: Enables validation.
- Authentication: Enables authentication.
- Request logger: Enables inspection of the latest requests and responses.

Plug-in Creation

In order to create a plug-in, ServiceStack exposes an **IPlugin** interface that a plug-in has to implement. The **IPlugin** interface is defined as follows.

```
public interface IPlugin
{
    void Register(IAppHost appHost);
}
```

As an example, let's implement a simple plug-in that enhances ServiceStack by offering an interface to query the system at run time and return a list of currently loaded plug-ins.

The first step is to create the plug-in itself as follows.

```
public class RegisteredPluginsFeature : IPlugin
{
    public void Register(IAppHost appHost)
    {
        appHost.Routes.Add(typeof(RegisteredPluginsDTO), "/ListPlugins", "GET");
        appHost.RegisterService(typeof(RegisteredPluginsService), "/ListPlugins");
    }
}
```

We can see that the plug-in mainly does the following:

- Implements the **`Register`** method of the **`IPlugin`** interface.
- Adds a new **`Route`**. As we would normally do for any other service, the new route is created to support the routing to the new service. This is done by mapping the **`RegisteredPluginsDTO`** class to the **`/ListPlugins`** URI for the GET method.
- Registers a new service. The service will be available to the given **`/ListPlugins`** URI. The new service is implemented as **`RegisteredPluginsService`**.

The following code example is the implementation of the service.

```
public class RegisteredPluginsDTO
{
    //Used only as a route to the service.
}

public class RegisteredPlugin
{
    public string Name { get; set; }
}

public class RegisteredPluginsService : ServiceStack.ServiceInterface.Service
{
    public object Get(RegisteredPluginsDTO request)
    {
        var plugins = base.GetAppHost().Plugins.ToList();
        var list = plugins.Select(x => new RegisteredPlugin()
        {
            Name = x.GetType().Name

        });

        return list.ToList();
    }
}
```

As you can see, the implementation is simple. In the **`GET`** method, we iterate through the list of the currently registered plug-ins and return it as **`List<RegisteredPlugin>`**.

Plug-in Registration

In order to be available to the system, the plug-in has to be registered. The registration is done in the application host (as we have seen in previous chapters) by using the **`Plugins`** collection.

```
public override void Configure(Container container)
{
    Plugins.Add(new RegisteredPluginsFeature());
}
```

Plug-in Usage

When running the application and pointing to the predefined plug-in URI **`/ListPlugins`** (in our case, **`http://localhost:50712/ListPlugins?format=xml`**), we can see the list of plug-ins registered in the system.

```
<ArrayOfRegisteredPlugin xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/ServiceStack.Succinctly.Host.Plugin
s">
  <RegisteredPlugin>
    <Name>HtmlFormat</Name>
  </RegisteredPlugin>
  <RegisteredPlugin>
    <Name>CsvFormat</Name>
  </RegisteredPlugin>
  <RegisteredPlugin>
    <Name>MarkdownFormat</Name>
  </RegisteredPlugin>
  <RegisteredPlugin>
    <Name>PredefinedRoutesFeature</Name>
  </RegisteredPlugin>
  <RegisteredPlugin>
    <Name>MetadataFeature</Name>
  </RegisteredPlugin>
  <RegisteredPlugin>
    <Name>RequestInfoFeature</Name>
  </RegisteredPlugin>
  <RegisteredPlugin>
    <Name>RegisteredPluginsFeature</Name>
  </RegisteredPlugin>
</ArrayOfRegisteredPlugin>
```

As you can see, the new URI can be queried as any other service we have seen in the previous examples, and it can return a result in XML, JSON, or any other format supported by the framework.

Chapter 12 Documenting Web Services

One of the important things to do when exposing RESTful web services is to provide good documentation. There's no real standard (or at least a *de facto* standard) to expose a REST contract (WADL[31] is an attempt). Many REST APIs expose a human-readable documentation, which, if manually edited, can be hard to keep perfectly synchronized with the API's current state. An alternative way is to create the documentation from the code itself.

ServiceStack currently supports two ways of documenting web services from the code:

- Metadata page
- Swagger integration[32]

Metadata Page

We have already mentioned that there is a metadata page that is automatically generated by the framework; it includes the main information about the operations and types included in the service.

In order for the metadata page to display the information required, the main source of information is the **`RouteAttribute`**. The same applies to the **`Route`** definition in the application host as the two offer the same functionality.

`RouteAttribute` provides five properties that can be set. The following is the public API of the **`RouteAttribute`** definition.

```
public class RouteAttribute : Attribute
{
    public object TypeId { get; set; }
    public string Path { get; set; }
    public string Verbs { get; set; }
    public string Notes { get; set; }
    public string Summary { get; set; }
}
```

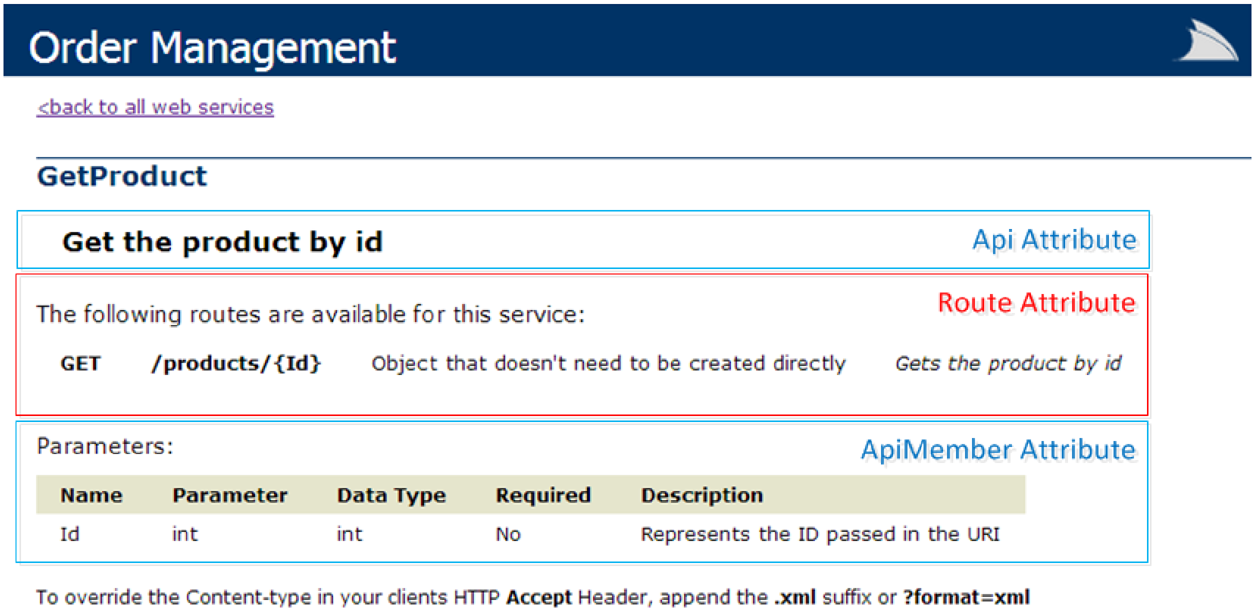
However, there is support for the new attributes **`Api`**, **`ApiResponse`**, **`ApiMember`**, and **`ApiAllowableValues`** which are now available to further enrich the objects. The new attributes will be mainly used when integrated with Swagger but, as we will see, some of them are available in the metadata page too.

[31] WADL: http://en.wikipedia.org/wiki/Web_Application_Description_Language
[32] Swagger's official webpage: https://developers.helloreverb.com/swagger/

Let’s look at an example of **`Route`** and **`Api`** attributes usage.

```
[Route("/products/{Id}", Verbs="GET", Notes = "Gets the product by id",
Summary = "Object that doesn't need to be created directly")]
[Api("Get the product by id")]
[ApiResponse(HttpStatusCode.NotFound,
"No products have been found in the repository")]
public class GetProduct
{
    [ApiMember(AllowMultiple = false,
    DataType = "int", Description = "Represents the ID passed in the URI",
    IsRequired = false, Name = "Id", ParameterType = "int", Verb = "GET")]
    public int Id { get; set; }
}
```

Once we run the application in the metadata page, we can see all of the information in the previous code example. As shown in the following figure, all of the information is available.



Order Management

<back to all web services

GetProduct

Get the product by id

The following routes are available for this service:

GET	/products/{Id}	Object that doesn't need to be created directly	*Gets the product by id*

Parameters:

Name	Parameter	Data Type	Required	Description
Id	int	int	No	Represents the ID passed in the URI

To override the Content-type in your clients HTTP **Accept** Header, append the **.xml** suffix or **?format=xml**

Figure 23: Metadata page

At the time of writing, **`ApiResponse`** and **`ApiAllowableValues`** are not displayed in the metadata page.

`ApiResponse` allows you to specify the different error response statuses that the service can return. It can be declared several times, once for each error.

The **`ApiAllowableValues`** attribute allows you to specify an allowable minimum and maximum numeric range, a list of named values, an Enum of named values, or a custom factory that returns a list of names.

Swagger Integration

The following description of Swagger is taken directly from the company webpage:

"*Swagger is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services.*"

ServiceStack has an API that enables integration with the Swagger framework. You can integrate Swagger in a matter of minutes.

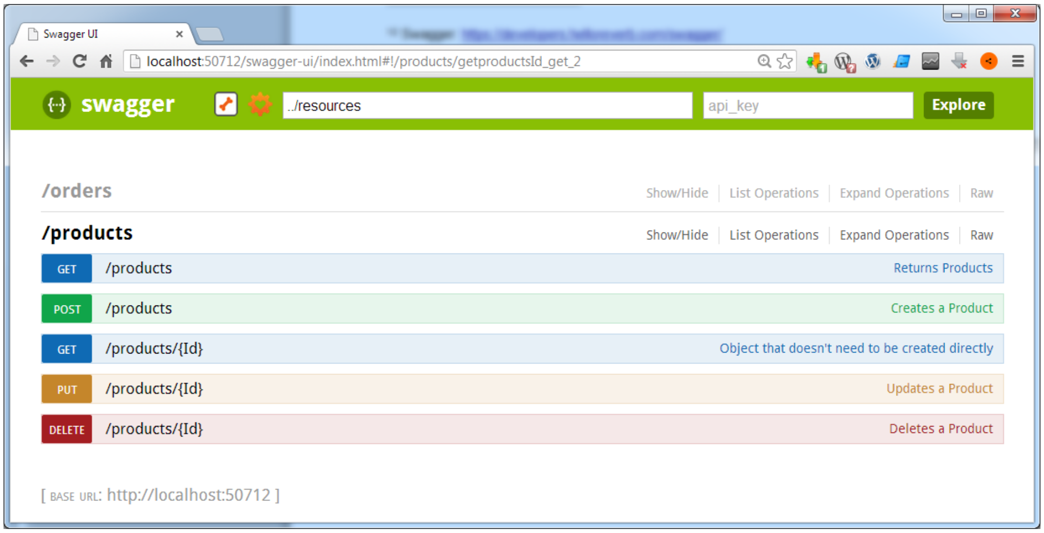
1. Install the **`ServiceStack.Api.Swagger`** NuGet package. This will create a **swagger-ui** folder where the Swagger JavaScript and HTML page is stored. Additionally, this will create two new services: **`/resources`** and **`/resource/name*`**. These two services are the data source for the Swagger UI.
2. Configure the application host by adding the **`SwaggerFeature`**.

```
Plugins.Add(new SwaggerFeature());
```

3. Configure the **`index.html`** page in the **swagger-ui** folder by changing the **`discoveryUrl`** value to **`"../resources"`**.

```
<script type="text/javascript">
  $(function () {
      window.swaggerUi = new SwaggerUi({
      discoveryUrl:"../resources",
```

4. Run the application. In the browser, navigate to **`/swagger-ui/index.html`**. As shown in the following figure, Swagger will give you a great deal of information about your API.



Swagger UI

localhost:50712/swagger-ui/index.html#!/products/getproductsId_get_2

swagger ../resources api_key Explore

/orders

/products

GET /products Returns Products

POST /products Creates a Product

GET /products/{Id} Object that doesn't need to be created directly

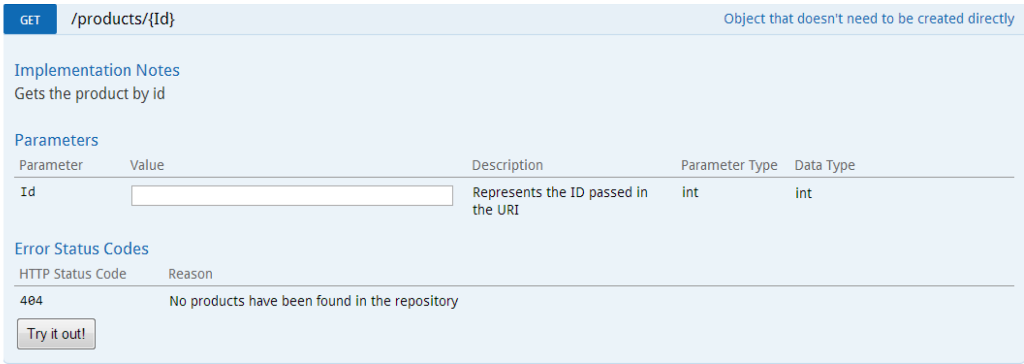
PUT /products/{Id} Updates a Product

DELETE /products/{Id} Deletes a Product

[BASE URL: http://localhost:50712]

Figure 24: Swagger interface

We can see that all of the information provided through the **`Api`** attributes is visible.



/products/{Id}

Object that doesn't need to be created directly

Implementation Notes

Gets the product by id

Parameters

Parameter	Value	Description	Parameter Type	Data Type
Id		Represents the ID passed in the URI	int	int

Error Status Codes

HTTP Status Code	Reason
404	No products have been found in the repository

Figure 25: Swagger getting products with parameters