Webbtjänster med API er

Mål med lektionen!

- Titta på WCF klienter och förstå dessa.

Vad lektionen omfattar

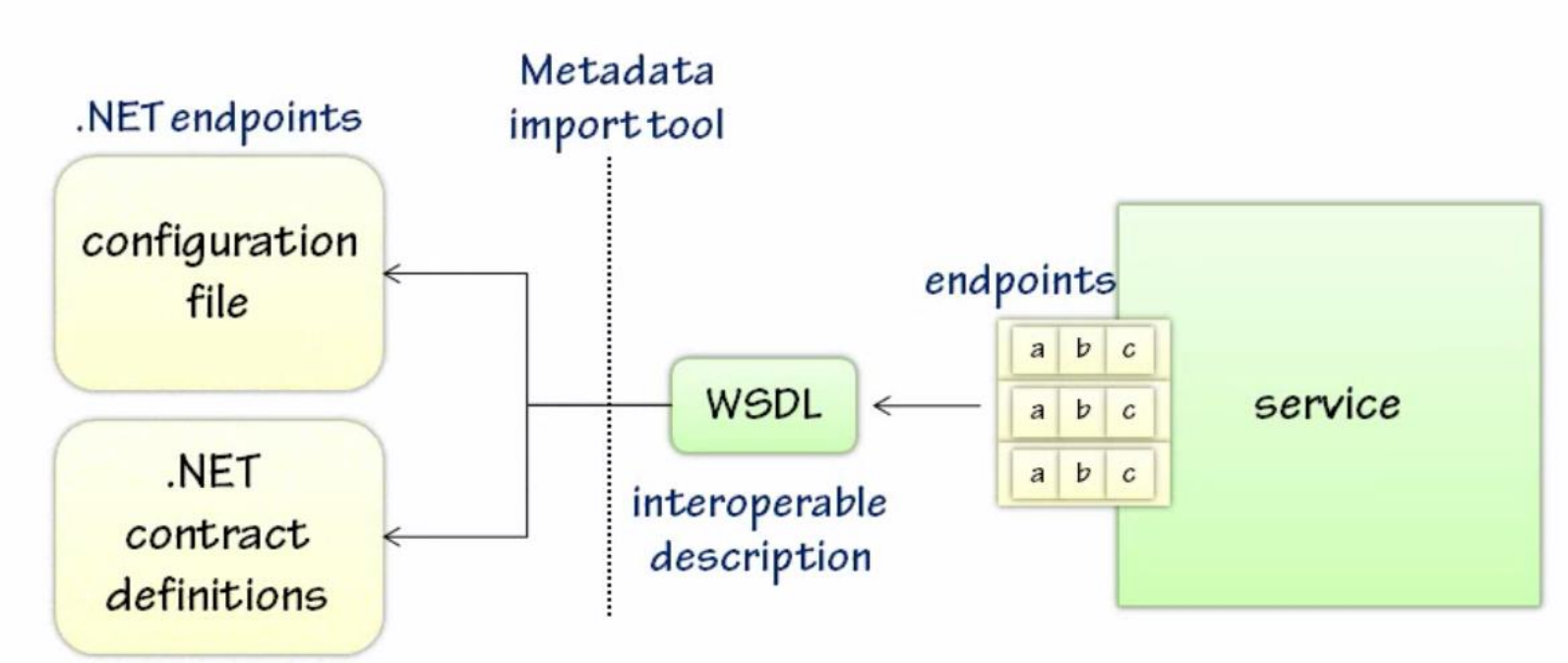
- WCF Clients

Komma åt endpoints

Vi har pratat om WCF i stort och vi har pratat om hur vi bygger tjänsten och hur vi hostar den. Nu är det dags att prata om klienten.

Då behöver vi först komma åt metadatan och då kan vi göra det på två sätt, metadatan finns i WSDL filen och via denna så kan vi generera två saker:

Komma åt endpoints



Vad vi får av WSDL

- Den ger en genererad kod fil som innehåller kontrakt definitionerna.
- Den ger en genererad konfigurations fil som vi kan använda som app.config fil, denna innehåller de endpoints som vi behöver.

Kort övning svcutils.exe

Öppna EvalServicelösningen och kör ConsoleHost applikationen.

Öppna sedan VisualStudio commandprompt, i kommandoprompten

- Stega er sedan fram till den mapp där ni har er lösning.
- Skriv sedan: svcutil http://localhost:8080/evals/mex.
- Tryck enter.

Kort övning svcutils.exe & wcf klient

I EvalService lösningen skapa ett nytt konsol projekt som ni döper till ”Client”.

Släng app.config filen, högerklicka på projektet och välj lägg till existerande items. Lägg till de två filerna som ni genererade nyss.

Referera in System.ServiceModel och System.Runtime.Serialization

Döp om output.config till app.config

Kompilera.

Programmera kanaler

Vi har nu fått tag på våra endpoints och genom att använda **WCF client-side programming model** så skall vi bygga en kanal som är kapabel att skicka ett meddelande till vår tjänst.

Att skapa en kanal består egentligen av några steg:

Programmera kanaler

1. Skapa och konfigurera en ChannelFactory
2. Skapa en faktisk instans av en kanal
3. Gör metodanrop genom kanalen
4. Stäng eller anropa Abort

Hur det kan se ut

```
ChannelFactory<IInvoiceService> factory =
    new ChannelFactory<IInvoiceService>("tcpEndpoint");

IInvoiceService channel = factory.CreateChannel();

channel.SubmitInvoice(invoice);

((IClientChannel)channel).Close();
```

```
ChannelFactory<IInvoiceService> factory =
    new ChannelFactory<IInvoiceService>(
        new BasicHttpBinding(),
        new EndpointAddress("http://server/invoiceservice"));
```

Client-side app.config

```
    <netNamedPipeBinding>
        <binding name="NetNamedPipeBinding_IEvalService" />
    </netNamedPipeBinding>
    <netTcpBinding>
        <binding name="NetTcpBinding_IEvalService" />
    </netTcpBinding>
    <wsHttpBinding>
        <binding name="WSHttpBinding_IEvalService">
            <reliableSession enabled="true" />
            <security mode="None" />
        </binding>
    </wsHttpBinding>
</bindings>
<client>
    <endpoint address="http://localhost:8080/evals/basic" binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_IEvalService" contract="IEvalService"
        name="BasicHttpBinding_IEvalService" />
    <endpoint address="http://localhost:8080/evals/ws" binding="wsHttpBinding"
        bindingConfiguration="WSHttpBinding_IEvalService" contract="IEvalService"
        name="WSHttpBinding_IEvalService" />
    <endpoint address="net.tcp://localhost:8081/evals" binding="netTcpBinding"
        bindingConfiguration="NetTcpBinding_IEvalService" contract="IEvalService"
        name="NetTcpBinding_IEvalService">
        <identity>
            <userPrincipalName value="devel@tahoedev.local" />
        </identity>
    </endpoint>
    <endpoint address="net.pipe://localhost/evals" binding="netNamedPipeBinding"
        bindingConfiguration="NetNamedPipeBinding_IEvalService" contract="IEvalService"
        name="NetNamedPipeBinding_IEvalService">
        <identity>
            <userPrincipalName value="devel@tahoedev.local" />
        </identity>
    </endpoint>
    <endpoint address="http://localhost:8080/evals/onemore" binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_IEvalService" contract="IEvalService"
        name="BasicHttpBinding_IEvalService1" />
</client>
```

Kanalens lifscykel

Vi skapar en kanalgenom att anropa CreateChannel metoden på ChannelFactory, när vi gör det så bygger WCF den underliggande runtime logiken för att kunna skicka meddelanden över nätverket till tjänsten. Den gömmer det mesta för oss genom att enkelt returnera ett objekt som implementerar vårt Service Kontrakt.

Så allt ni ”interfacar” med är vårt service kontrakt. Vi gör helt enkelt enbart metod anrop som översätts till meddelanden som skickas genom den underliggande kanal stacken.

Alla kanalinstanser implementerar också ett interface som heter IClientChannel som ger oss Close och Abort

Close() gör en behaglig/ordentlig nedstängning

- Den väntar för anrop som är in-progress för att de skall bli färdiga innan den stängs.
- Kan bli en utdragen process (det finns async varianter).
- Den stänger ned underliggande nätverks resurser.
- Kan kasta CommunicationException & TimeOutException

Abort() sliter ner allt i klientkanalen omedelbart.

- Aborts in-progress anrop och stänger ned kanalen bums!

Liten klient övning

Skapa nu en channelfactory och en channel samt en instans av Eval som ni ger lite värden.

Stoppa in er eval instans i ett anrop till SubmitEval på en channel instans.

Ta emot ett anrop ifrån GetEvals och skriv ut detta.

Stäng kanalen.

Undvika använda Channelfactory

Alla kanaler implementerar IClientChannel där vi har bla Close och Abort, men vi måste casta objektet till IClientChannel för att kunna använda dem/anropa dem.

När vi genererar kod så genereras det även en proxyklass som gör det hela lite lättare för oss. Det genereras en sådan här för varje ServiceContract typ som vi har.

Namnet byggs alltid upp enligt följande:

ServiceContract namn (minus I) + Client.

Kort övning igen

Ta fram samma klient ifrån EvalService lösningen och kommentera ut skapandet av Factory och skapandet av channel.

Skapa nu en ny channel som är av proxy klass typen.

Sedan när ni är klara med metod anropen till tjänsten så anropar ni close direkt på channeln istället för att casta den.

Konfigurering av vår klient kanal

Vår klient kanal kan konfigureras på både bindings och behaviors.

Några saker som kan konfigureras på bindningen:

- Skicka TimeOut värde (detta påverkar när det skall ske ett TimeOutException).
- Specifika transport/protokoll inställningar (behöver oftast vara i synk med servicen)

Klient behaviors kallas ”endpoint” behavior i WCF

- ClientViaBehavior är ett exempel som vi kan använda om vi vill använda oss av auto-routing.

Config exempel

```
<configuration>
    <system.serviceModel>
        <client>
            <endpoint name="httpEndpoint"
                address="http://server/invoiceservice"
                binding="basicHttpBinding"
                bindingConfiguration="MyConfiguration"
                contract="InvoiceServiceReference.IInvoiceService" />
        </client>
        <bindings>
            <basicHttpBinding>
                <binding name="MyConfiguration" sendTimeout="00:05:00">
                    <security mode="Transport">
                      <transport clientCredentialType="Basic"/>
                    </security>
                </binding>
            </basicHttpBinding>
        </bindings>
    </system.serviceModel>
</configuration>
```

Ökar timeout med 5 min

Basic auth över SSL

Config exempel

```
<configuration>
    <system.serviceModel>
        <client>
            <endpoint name="httpEndpoint"
                address="http://server/invoiceservice"
                binding="basicHttpBinding"
                behaviorConfiguration="viaBehavior"
                contract="InvoiceServiceReference.IInvoiceService" />
        </client>
        <behaviors>
            <endpointBehaviors>
                <behavior name="viaBehavior">
                    <clientVia viaUri="http://router/invoiceservice"/>
                </behavior>
            </endpointBehaviors>
        </behaviors>
```

Adressen som du vill att de utgående meddelandena skall routas genom

Exceptions

Där är i huvudsak två typer av exceptions som vi behöver vara beredda på/medvetna om.

Den ena är **CommunicationException** som är en slags basklass för exceptions. Vilket gör att vi kan göra en ”Gona catch em all” och fånga exceptions av denna typ. Men det finns ett annat exception som du nog gärna vill fånga och det är FaultException.

TimeOutException denna typ ärver inte av CommunicationException, och skickas när vi överskrider Timeout värdet.

Demo exceptions

Asynkrona WCF tjänster

WCF gör det möjligt att använda tjänsterna asynkront, när vi använder svcUtils.exe så skall vi använda /async switchen när vi genererar koden. Vilket ger oss Begin/End anropsmetoder för varje operation. Om vi använder oss av Add Service Reference featuren så finns det en checkbox/radiobtn att klicka i för detta.

Sedan .Net WCF 3.5 så har de lagt till en simplifierad asynkron programmerings modell som genererar dessa direkt åt oss, där finns metodanrop som slutar på Async samt events till dessa.

Async övning

Ta bort EvalService.cs samt kommentera bort allt i app.config som finns i taggarna ”bindings” och ”client”. Ta bort referenserna för System.ServiceModel och System.Runtime.Serialization.

Högerklicka på ”References” och välj ”Add Service Reference” skriv in adressen http://localhost:8080/evals/meta samt se till att ni genererar asynkrona metoder.

Ni kan även se till att ändra collection types till System.Collections.Generic.List för collection types och ändra i er kod. Dvs ta bort arrayen och sätt dit en lista samt ändra lenght till count.

Async övning

Istället för att anropa GetEval() så skall ni anropa GetEvalAsync() och på raden innan skall ni anropa GetEvalCompleted som får resultatet plus vad det var innan av new Eventhandler<GetEvalCompletedEventArgs>(channel_GetEvalCompleted);

Glöm inte att ”tabba” för att få metoden skapad åt er, i denna skall ni skriva ut e.Result.Count till konsolen.

Innan ni stänger kanalen i try satsen så lägg in en konsol utskrift som säger ”waiting” och en Console.ReadLine

Dynamiska endpoints

Genom att använda MEX så tillåts vi att hämta metadata programmatiskt. WCF tillhandahåller bla två klasser som gör detta .

En som heter MetadataResolver och en som heter MetadataExchangeClient. Båda gör i huvudsak samma sak, de går upp till tjänsten vid körning med hjälp av MEX protokollet och ber om metadatan. Tjänsten skickar då metadatan i svaret som sedan dessa två klasserna använder informationen i någon form av objektmodell.

Dynamiska endpoints

Om man vill ta den informationen som returneras och översätta den till en uppsättning endpoints som vi sedan kan använda då är det MetadataResolver klassen vi vill använda.

Detta kan vi sedan programmatiskt inspektera om vi vill och bygga en kanalstack baserat på våran valda endpoint. Detta ger oss lite andra möjligheter, vi behöver inte ha endpoint information som lagrats statiskt antingen i kod eller i konfigurationsfilen.

Dynamiska endpoints

Istället kan klienten alltid dynamiskt hämta metadatan vid körning.

Det fungerar ungefär som så att när klienten startar så ber den om metadata, laddar ner detta, väljer en endpoint och sedan kommunicerar med tjänsten.

För om tjänsten skulle behöva göra ändringar över tiden, så kommer klienterna/klienten inte gå sönder eftersom nästa gång den startar så hämtar den ny information.

Mex övning

Vi skall nu modifiera EvalService för att få denna dynamisk.

Först gå till EvalServiceLibrary-> EvalService.cs och kommentera bort Sleep. Bygg nu EvalServiceLibrary och kör exe filen so att ConsoleHost ligger och väntar.

Skapa först en utskrift som säger att vi hämtar endpoints via mex

Sedan skapa en instans av ServiceEndpointsCollection som tilldelas resultatet ifrån MetadataResolver.Resolve(typeof(service kontraktet), new EndpointAddress(”adressen till mex”));

Mer mex övning

Loopa sedan igenom för varje ServiceEndpoint som finns i samlingen och ha detta runt om era try-catch.

Sedan skall vi ändra lite i try delen, kanalen skall nu istället använda en konstruktor som tar två parametrar.

Dessa två är endpointens bindning och dess adress.

Kommentera ut asynk anropet och ta fram det ”gamla” som returnerade en lista som vi skrev ut count på (antalet evals).

REST vs SOAP

REST’s starka sida är när du exponerar ett publikt API över internet för att hantera CRUD operationer på data. REST är fokuserat på att komma åt namngivna resurser genom ett konsistent interface.

SOAP har sitt egna protokoll och fokuserar på exponera bitar av applikationslogik (inte data) som tjänster. SOAP fokuserar på namngivna operationer, varje operation implementerar lite business logik genom olika interfaces.

REST vs SOAP

Även om SOAP ofta refereras som ”Web Services” så är det en del som tycker att det är en felaktig betäckning, REST förser äkta ”Web Services” genom URI’s och HTTP.

Eftersom REST är standard HTTP så är det mycket enklare (enligt en del) att skapa klienter, utveckla API’er och dokumentationen brukar vara lättare att förstå, och det är inte många saker som REST inte gör enklare eller bättre än SOAP.

REST vs SOAP

REST tillåter många olika data format medans SOAP endast tillåter XML. Detta kan en del tycka att det ger mer komplexitet till REST eftersom du behöver hantera multipla format. JSON är oftast ett bättre val av/för data och den ”parsar” mycket snabbare.

REST har lite bättre prestanda och skalbarhet, REST läsningar kan ”cachas” det kan inte SOAP läsningar.

Varför SOAP?

Medans SOAP stödjer SSL (precis som REST) så stödjer det även WS-Security vilket lägger till en del säkerhets funktioner som man annars bara ser i större företags lösningar (enterprise lösningar).

Och stödjer identitet genom mellanhänder (tredjepart), inte bara point–to-point (SSL). Den förser också med en standard implementation för dataintegritet och datasekretess.

Genom att kalla det ”enteprise” gör det inte mer säkert, det bara stödjer en del säkerhetsverktyg, som typiska internet tjänster inte har någon större nytta av utan endast behövs i några få fall

Varför SOAP?

Behöver du ACID transaktioner för en tjänst då behöver du använda SOAP, medans REST inte stödjer transaktioner lika omfattande och inte stödjer ACID. Lyckligtvis så är nästan aldrig ACID transaktioner över internet vettiga, medans REST begränsas av HTTP som själv inte kan ge två-fas ”commit” över distribuerade transaktions resurser, men SOAP kan. Internet appar i allmänhet behöver inte denna nivå av transaktions tillförlitlighet, vilket ”enterprise” appar ibland gör.

Varför SOAP?

REST har ingen standard för meddelandesystem och förväntar sig att klienterna hanterar kommunikationsfel genom att försöka igen.

SOAP har ”succesful/retry” logik inbyggd och ger end-to-end tillförlitlighet även genom SOAP mellanhänder.