Windows Communication Foundation

Windows Communication Foundation (**WCF**) je programsko okruženje(deo **.NET** Framework-a) koje služi za međuprocesnu komunikaciju. **WCF** omogućava kreiranje servisno orjentisanih aplikacija, odnosno kreiranje klijenata, koji pristupaju servisima. Klijent, kao i servis, mogu biti pokrenuti unutar bilo kod **Windows** procesa.

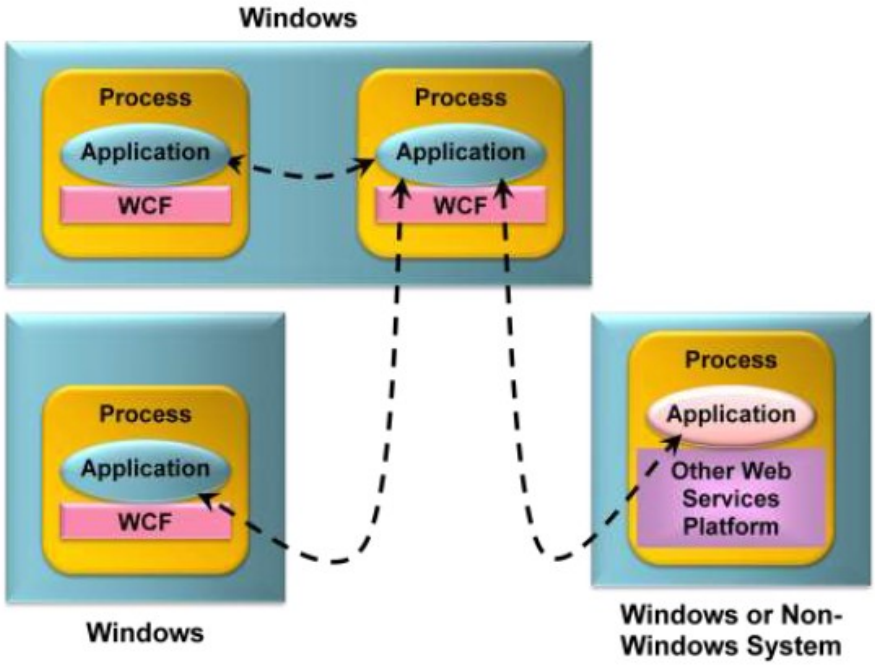
WCF predstavlja uniju mnoštva **Microsoft** tehnologija za distribuirano programiranje, poput:

- **ASP .NET Web Services (ASMX)**
- **.NET Remoting**
- **Enterprise Services**
- **Web Services Enhancements (WSE)**
- **Microsoft Message Queing (MMQ)**
- **Representational State Transfer (REST)**

Pre postojanja **WCF**-a, bilo je neophodno odabrati odgovarajuću komunikacionu tehnologiju za svaki deo aplikacije, što je nekada zahtevalo i sve gorenavedene tehnologije. Iako je ovo tehnički moguće izvesti, rezultujuća aplikacija bi bila kompleksna za implementiranje i zahtevna za održavanje. **WCF** omogućava upotrebu samo jedne tehnologije za iste zadatke, jer poseduje objedinjenu funkcionalnost svih navedenih tehnologija. Rezultat ove objedinjene funkcionalnosti je niža kompleksnost aplikacija. Iako **Microsoft** i dalje podržava ove ranije tehnologije, nove aplikacije koje bi koristile bilo koju od tih tehnologija se sada grade pomoću **WCF**-a.

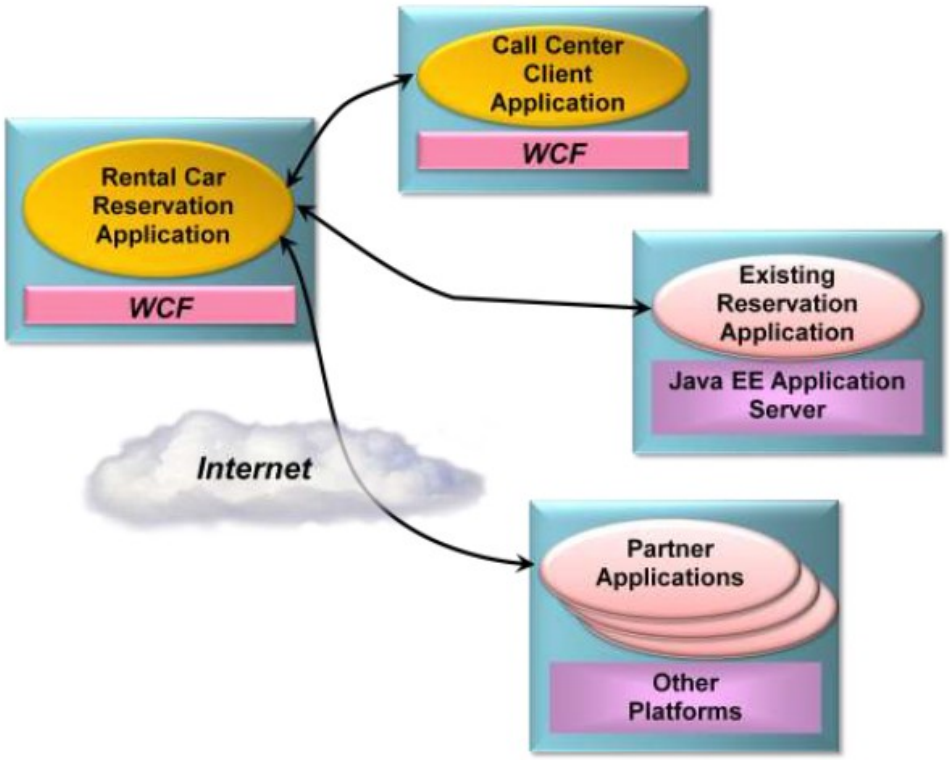
Aplikacije bazirane na **WCF**-u mogu da interaguju sa narednim tipovima aplikacija:

- **WCF**-baziranim aplikacijama pokrenutim unutar drugog procesa na istoj **Windows** mašini
- **WCF**-baziranim aplikacijama pokrenutim na drugim **Windows** mašinama
- Aplikacijama izgrađenim na drugim tehnologijama, poput **Java EE application** servera, koje podržavaju standardne **Web** servise. Ove aplikacije mogu biti pokrenute na **Windows** mačinama ili na mašinama sa drugim operativnim sistemima



Upotreba WCF-a

Ova sekcija pokazuje šta je potrebno za kreiranje i konzumiranje **WCF** servisa. Primer koji će biti korišten u ovu svrhu jeste primer rent-a-car aplikacije. U nastavku je data servisno orjentisana arhitektura ove aplikacije:



Tokom svog radnog veka, rent-a-car aplikaciji će verovatno pristupiti veliki broj drugih aplikacija. Međutim, u trenutku dizajniranja aplikacije, arhitekte znaju da će poslovnoj logici pristupiti tri različite vrste softvera:

1. Klijentska aplikacija na **Windows** desktop mašinama, koju će upotrebljavati zaposleni u call centru ove organizacije. Ova aplikacija će takođe biti kreirana pomoću **WCF**-a.
2. Već postojeća aplikacija za rezervaciju izgrađena na Java platformi. Postojeći sistemi moraju imati pristup logici nove aplikacije, kako bi klijenti imali uniformno iskustvo.
3. Partnerske aplikacije koje se pokreću na različitim platformama, od kojih se svaka nalazi unutar firme koja ima ugovoreno poslovanje sa rent-a-car firmom. Partneri podrazumevaju turističke agencije, aviokompanije, kao i mnoge druge firme kojima je potrebno da prave rent-a-car rezervacije.

Kreiranje WCF servisa

Svaki **WCF** servis ima tri onsovne komponente:

1. **Servisnu klasu**, koja implementira jednu ili više metoda
2. **Host proces**, unutar kojeg je servis pokrenut
3. Jedna ili više **krajnih tačaka**(*eng. endpoint*) preko kojih klijenti pristupaju servisu. Sva komunikacija sa **WCF** servisom se obavlja putem ovih tačaka.

Razumevanje **WCF** servisa zahteva prethodno razumevanje sva tri nabrojana koncepta. Naredne sekcije opisuju svaki od njih, počevši od servisne klase.

Implementiranje servisne klase

Servisna klasa je klasa kao i svaka druga, ali sa par dodataka. Ovi dodaci omogućavaju kreatoru klase da definiše jedan ili više ugovora, koje klasa implementira. Svaka servisna klasa implementira bar jedan servisni ugovor(*eng. service contract*), koji definiše operacije izložene(*eng. exposed*) od strane servisa. Ova klasa može obezbediti i eksplicitni ugovor o podacima(*eng. data contract*), koji definiše podatke koje te operacije prenose.

Definisanje servisnog ugovora

Svaka servisna klasa implementira metode za potrebe klijenata. Kreatori klase odlučuju koje metode će biti dostupne za poziv klijentima, tako što ih označe kao deo nekog servisnog ugovora. Kako bi ovo bilo učinjeno, koristi se **WCF**-definisan atribut **ServiceContract**. Zapravo, servisna klasa je samo klasa koja je označena sa **ServiceContract** atributom, ili implementira interfejs označen istim atributom.

Za početak, dat je kratak opis klase **RentalReservations**, koja će biti upotrebljena kroz primere u ovoj sekciji. Ova klasa predstavlja implementaciju osnovnih funkcionalnosti unutar rent-a-car aplikacije opisane u prethodnoj sekciji, a sastoji se od četiri metode:

1. **Check**, koja dozvoljava klijentima da provere dostupnost željene vrste vozila na određenoj lokaciji, za određeni datum. Ova metoda vraća **bool** vrednost koja govori da li je vozilo dostupno ili ne.
2. **Reserve**, koja dozvoljava klijentima da rezervišu željeni tip vozila na određenoj lokaciji, za određeni datum. Ova metoda vraća potvrdni broj.
3. **Cancel**, koja dozvoljava klijentu da otkaže postojeću rezervaciju na osnovu potvrdnog broja. Ova metoda vraća **bool** vrednost koja govori da li je otkazivanje uspelo ili ne.
4. **GetStats**, koja vraća broj trenutnih rezervacija. Za razliku od prethodnih metoda, **GetStats** se poziva samo od strane lokalnog administratora, a ne od strane klijenata.

Sa ovim na umu, data skraćena verzija opisanog koda:

```
using System.ServiceModel;

namespace RentalReservationsApp
{
     [ServiceContract]
     class RentalReservations
     {
          [OperationContract]
          public bool Check(int vehicleClass, int location, string dates)
          {
               bool availability;
               // code to check availability goes here
               return availability;
          }

          [OperationContract]
          public int Reserve(int vehicleClass, int location, string dates)
          {
               int confirmationNumber;
               // code to reserve rental car goes here
               return confirmationNumber;
          }
```

```
        [OperationContract]
        public bool Cancel(int confirmationNumber)
        {
            bool success;
            // code to cancel reservation goes here
            return success;
        }

        public int GetStats()
        {
            int numberOfReservations;
            // code to get the current reservation count goes here
            return numberOfReservations;
        }
    }
}
```

Atributi **ServiceContract**, kao i svi ostali atributi upotrebljeni unutar ovog primera su definisani u **System.ServiceModel** imenskom prostoru. Svaka metoda ove servisne klase, koja može biti pozvana od strane klijenta, mora biti označena sa atributom **OperationContract**. U ovom primeru, **Check**, **Reserve** i **Cancel** metode su obeležene ovim atributom, tako da će sve tri biti izložene klijentima ovog servisa. Bilo koja metoda koja nije obeležena sa **OperationContract** atributom, u ovo slučaju **GetStats**, neće biti uključena u servisni ugovor i stoga klijenti ovog servisa neće moći da je pozovu.

Prethodni primer ilustruje najjednostavniji način za kreiranje **WCF** servisne klase: direktim obeležavanjem klase **ServiceContract** atributom. Međutim, moguće je i eksplicitno specificirati servisni ugovor, korišćenjem interfejsa. Upotrebom ovog pristupa, servisna klasa **RentalReservations** bi izgledala ovako:

```
using System.ServiceModel;

namespace RentalReservationsApp
{
    [ServiceContract]
    interface IReservations
    {
        [OperationContract]
        bool Check(int vehicleClass, int location, string dates);

        [OperationContract]
        int Reserve(int vehicleClass, int location, string dates);

        [OperationContract]
        bool Cancel(int confirmationNumber);
    }

    class RentalReservations : IReservations
    {
        public bool Check(int vehicleClass, int location, string dates)
        {
            bool availability;
            // logic to check availability goes here
            return availability;
        }
```

```
        public int Reserve(int vehicleClass, int location, string dates)
        {
            int confirmationNumber;
            // logic to reserve rental car goes here
            return confirmationNumber;
        }

        public bool Cancel(int confirmationNumber)
        {
            bool success;
            // logic to cancel reservation goes here
            return success;
        }

        public int GetStats()
        {
            int numberOfReservations;
            // logic to determine reservation count goes here
            return numberOfReservations;
        }
    }
}
```

U ovom primeru, atributi **ServiceContract** i **OperationContract** se dodeljuju **IReservations** interfejsu i metodama koje on sadrži, a ne direktno klasi **RentalReservations**. Međutim, rezultat je isti. Korišćenje eksplicitnih interfejsa je malo komplikovanije, ali dozvoljava veću fleksibilnost. Kako klasa može implementirati više od jednog interfejsa, ona može implementirati i više od jednog servisnog ugovora. Izlaganjem više krajnjih tačaka, svake sa različitim servisnim ugovorom, servisna klasa može prikazati različite grupe servisa za različite klijente.

Definisanje ugovora o podacima

WCF servisna klasa specificira servisni ugovor, definišući koje će od njenih metoda biti izložene klijentima servisa. Svaka od tih operacija će obično preneti neke podatke, što znači da servisni ugovor podrazumeva i neki ugovor o podacima putem kojeg se opsuju informacije koje će biti razmenjene. U nekim slučajevima, ugovor o podacima se definiše implicitno kao deo servisnog ugovora. Ovo se odnosi na one servisne ugovore, u okviru kojih metode prenose osnovne tipove podataka. Međutim, servisi mogu da imaju i parametre kompleksnijih tipova, kao što su strukture. U slučajevima poput ovog, potreban je eksplicitni ugovor o podacima. Ugovori o podacima nam govore kako se tipovi iz memorije(*eng. in-memory*) konvertuju u formu koja je pogodna za prenos preko žice. Ovaj proces konverzije je poznat kao serijalizacija. Ugovori o podacima predstavljaju mehanizam za kontrolisanje načina serijalizacije.

U **WCF** servisnim klasama, ugovor o podacima se definiše upotrebom **DataContract** atributa. Klasa ili struktura obeležena ovim atributom može imati jednog ili više članova obeleženih **DataMember** atributom, ukazujući na to da bi taj član trebalo da bude uključen u serijalizovanu vrednost te klase/strukture. Na primer, zamislićemo da su liste parametara **Check** i **Reserve** metoda, iz **RentalReservation** klase, zamenjene strukturom **ReservationInfo**, koja sarži iste informacije. Kako bi se ova struuktura prosledila kao parametar pomoću **WCF**-a, neophodno je da se ona obeleži sa **DataContract** atributom, a njeni članovi sa **DataMember** atributima. Primer ove klase dat je u nastavku.

```
using System.Runtime.Serialization;

namespace RentalReservationsApp
{
     [DataContract]
     struct ReservationInfo
     {
          [DataMember]
          public int vehicleClass;

          [DataMember]
          public int location;

          [DataMember]
          public string dates;
     }
}
```

Za razliku od atributa koji su do sada definisani, atributi koji se upotrebljavaju prilikom izrade ugovora o podacima se nalaze unutar **System.Runtime.Serialization** imenskog prostora. Kada se instanca tipa **ReservationInfo** prosledi kao parametar metodi obeleženoj sa **OperationContract** atributom, sva polja obeležena sa **DataMember** atributom će biti prosleđena. Ukoliko se bilo koje polje ostavi neobeleženo, ono neće biti prosleđeno prilikom prosleđivanja tipa **ReservationInfo**.

Ono o čemu treba voditi računa prilikom korišćenja **WCF** ugovora, jeste da ništa ne postaje deo servisnog ugovora ili ugovora o podacima samo od sebe. Umesto toga, atributi **ServiceContract** i **DataContract** moraju biti eksplicitno upotrebljeni, kako bi se ukazalo na to koji tipovi imaju **WCF**-definisane ugovore, a njihovi članovi moraju biti dodatno obeleženi sa **OperationContract** i **DataMember** atributima, kako bi se znalo koji delovi ovih tipova treba da budu izloženi klijentima.

Odabir domaćina(hosta)

WCF pruža dve osnovne mogućnosti za hostovanje **WCF** servisa. Jedna opcija podrazumeva host proces kreiran od strane **IIS**(*eng. Internet Information Services*) tehnologije ili srodne tehnologije **WAS**(*eng. Windows Activation Service*). Drugi omogućava hostovanje servisa unutar proizvoljnog procesa.

Najjednostavniji način za hostovanje **WCF** servisa jeste oslanjanjem na **IIS** ili **WAS**. Obe tehnologije se odnose na virtuelne direktorijume, koji zapravo predstavljaju skraćene alijase za stvarne putanje u **Windows file** sistemu.

Kako bi se nagovestilo da bi određeni **WCF** servis trebalo hostovati pomoću **IIS** ili **WAS** tehnologije, u virtuelnom direktorijumu je neophodno napraviti fajl sa ekstenzijom **.svc**(svc je skraćenica sa service). Kada je ovo obavljeno i definiše se kranja tačka, preko koje će se vršiti komunikacija(što će biti prikazano u narednoj sekciji), zahtev klijenta za upotrebu neke od servisnih metoda će automatski kreirati instancu servisne klase, kako bi se izvršila specificirana operacija. Ta instanca će biti pokrenuta unutar procesa koji je obezbeđen od strane **IIS** ili **WAS** tehnologije.

Iako je oslanjanje na **IIS** ili **WAS** najjednostavniji izbor, aplikacije često moraju da izlože servise preko sopstvenog procesa, a ne nekog procesa koji nam je **Windows** obezbedio. Srećom, ovo nije teško uraditi. Naredni primer pokazuje kako napraviti jednostavnu konzolnu aplikaciju, koja hostuje **RentalReservations** klasu definisanu ranije:

```
using System.ServiceModel;

namespace RentalReservationsApp
{
     public class Program
     {
          static void Main(string[] args)
          {
               ServiceHost s = new ServiceHost(typeof(RentalReservations));
               s.Open();
               Console.Writeline("Press ENTER to end service");
               Console.Readline();
               s.Close();
          }
     }
}
```

S obzirom na to da klasa **Program** ima **Main** metodu, ona pokreće zaseban proces. Kako bi se hostovao **RentalReservations** servis, ova metoda mora da kreira novu instancu klase **ServiceHost**, prosleđujući tip **RentalReservations** servisne klase. Kada je instanca ove klase kreirana jedino što preostaje da se uradi kako bi servis postao dostupan jeste da se pozove **Open** metoda, nad ovom instancom. **WCF** će automatski sprovoditi zahteve klijenata do odgovarajućih metoda unutar **RentalReservations** klase.

Kako bi se **WCF** servisu omogućilo da dobija zahteve od klijenata, neophodno je da proces koji hostuje servis ne prestane sa izvršavanjem. U prethodnom primeru, ovo je odrađeno pomoću jednostavnog čekanja na unos sa tastature. Servis se eksplicitno zatvara pozivom **Close** metode. Ovo nije neophodno u jednostavnim primerima, gde će servis biti automatski zatvoren kada proces prestane sa izvršavanjem. S druge strane, u kompleksnijim situacijama, gde proces hostuje više **WCF** servisa, ima smisla zatvarati pojedinačne servise kada oni više nisu potrebni.

Definisanje krajnjih tačaka

Uz definisanje operacija servisne klase i specificiranje host procesa, **WCF** servis mora da izloži jednu ili više krajnjih tačaka. Svaka krajnja tačka specificira tri stvari:

- Adresu, koja nam govori gde određena krajnja tačka može biti pronađena. Adrese predstavljaju **URL**-ove koji identifikuju mašinu i određenu krajnju tačku na toj mašini.
- Povezivanje(*eng. binding*), koje nam govori kako toj krajnjoj tački može biti pristupljeno. Povezivanje određuje koje se kombinacije protokola koriste za pristup krajnjoj tački, uz neke druge stvari, poput toga da li je komunikacija pouzdana i koji bezbednosni mehanizmi mogu biti upotrebljeni.
- Ime ugovora, koje nam govori koji servisni ugovor **WCF** servisna klasa izlaže putem ove krajnje tačke. Klasa označena sa **ServiceContract** atributom, koja ne implementira interfejse, kao što je to bio slučaj sa **RentalReservations** klasom u prvom primeru, može da izloži samo jedan servisni ugovor. U ovom slučaju, sve krajnje tačke će izlagati isti servisni ugovor. Međutim, ukoliko klasa eksplicitno implementira dva ili više interfejsa obeležena **ServiceContract** atributom, različite krajnje tačke mogu izlagati različite servisne ugovore, gde je svaki ugovor definisan od strane drugog interfejsa.

Jednostavan način za pamćenje svih stvari potrebnih za **WCF** komunikaciju jeste njihovim posmatrenjem kao **ABC** stavki krajnjih tačaka: **address**, **binding**, **contract**.

Adrese su jednostavne za razumevanje - predstavljaju samo **URL** – a ugovori su već opisani. Povezivanje predstavlja kritični deo komunikacije i stoga ga je potrebno dodatno opisati. Pretpostavimo da kreator servisa želi da dozvoli klijentima da pristupe servisu koristeći **SOAP** putem **HTTP**-a ili **SOAP** putem **TCP**-a. Svaki od ovih vidova komunikacije predstavlja poseban tip povezivanja, pa bi servis morao da izloži dve krajnje tačke, kako bi pokrio oba tipa.

Kako bi povezivanje bilo što jednostavnije, **WCF** ima set predefinisanih tipova, gde svaki tip predstavlja određenu grupu opcija. Ovi standardni tipovi povezivanja mogu biti izmenjeni po potrebi, ili mogu biti kreirani potpuno novi tipovi. Međutim, najveći broj aplikacija će koristiti standardne tipove povezivanja, koje **WCF** pruža, a neki od najvažnijih su sledeći:

- **BasicHttpBinding** - Osnovni **HTTP** tip komunikacije koji obezbeđuje maksimalnu interoperabilnost, a zasnovan je na **WS-BasicProfile 1.1**.
- **WSHttpBinding** – napredniji **HTTP** protokol zasnovan na **WS-* protocols**.
- **WSDualHttpBinding** - Dupleks **HTTP** komunikacija. Server otvara drugi kanal kroz koji može pozivati metode na klijentu.
- **WSFederationBinding** - **HTTP** komunikacija u kojoj pristup podacima može biti kontrolisan preko prava pristupa koje dodeljuje specifičan organ za dodelu sertifikata.
- **NetTcpBinding** – Sigurna, pouzdana, komunikacija visokih performansi zasnovana na **TCP** protokolu i binarnim porukama.
- **NetNamedPipeBinding** – Sigurna, pouzdana, komunikacija visokih performansi izmedu **WCF** aplikacija koje su pokrenute na istom računaru.
- **NetMsmqBinding** – Komunikacija izmedu **WCF** aplikacija koja se zasniva na **Microsoft Message Queuing** (**MSMQ**) serveru.
- **MsmqIntegrationBinding** - Komunikacija izmedu **WCF** aplikacija i ostalih aplikacija preko **MSMQ**.
- **NetPeerTcpBinding** – Komunikacija izmedu **WCF** aplikacija preko **Windows Peer-to-Peer** mreža.

Za razliku od atributa, koji predstavljaju deo izvornog koda servisa, tip povezivanja može da se razlikuje za različite primene istog servisa.

Specificiranje krajnjih tačaka

Za razliku od ugovora, krajnje tačke se ne definišu pomoću atributa. Međutim, ponekad je korisno specificirati krajnju tačku unutar koda, tako da **WCF** pruža način za to. Servis koji se eksplicitno hostuje unutar **ServiceHost** objekta, može upotrebiti **AddEndpoint** metodu tog objekta, kako bi kreirao krajnju tačku. Ukoliko se ovo uradi za prethodni primer rent-a-car kompanije, **Main** metoda bi izgledala ovako:

```
static void Main(string[] args)
{
      ServiceHost s = new ServiceHost(typeof(RentalReservations));
      s.AddEndpoint(typeof(IReservations),
            new BasicHttpBinding(),
            "http://localhost:4000/reservations");
      s.Open();
      Console.Writeline("Press ENTER to end service");
      Console.Readline();
      s.Close();
}
```

Tri parametra koja se prosleđuju **AddEndpoint** metodi su ugovor, povezivanje i adresa nove krajnje tačke. Iako je definisanje krajnje tačke programskim putem moguće, najčešći način je upotrebom konfiguracionog fajla povezanog sa servisom. Definisanje krajnjih tačaka unutar konfiguracionog fajla ih čini jednostavnijim za izmenu, jer izmene ne zahtevaju modifikaciju i rekompajliranje izvornog koda. Za servise koji su hostovani unutar **IIS** ili **WAS** procesa, krajnje tačke se definišu unutar **web.config** fajla, dok se za ostale procese to radi putem **app.config** fajla. Ukoliko bi se konfiguracioni fajl koristio samo za **RentalReservations** servisnu klasu, izgledao bi ovako:

```
<system.serviceModel>
      <services>
            <service name="RentalReservationsApp.RentalReservations">
                  <endpoint
                        contract="IReservations"
                        binding="basicHttpBinding"
                        address="http://localhost:4000/reservations"/>
            </service>
      </services>
</system.serviceModel>
```

Konfiguracione informacije za sve servise implementirane od strane **WCF**-bazirane aplikacije se nalaze unutar **system.serviceModel** elementa. Ovaj element sadrži **services** element koji može da sarži jedan ili više **service** elemenata. Ovaj jednostavan primer sadrži samo jedan servis, tako da se **service** element pojavljuje samo jednom. Atribut **type** iz **service** elementa identifikuje servisnu klasu koja implementira servis na koji se odnosi ovaj konfiguracioni fajl, što je u ovom slučaju klasa **RentalReservations**. Takođe, specificira i ime **.NET** sklopa(*eng. assembly*) koji implementira ovaj servis, što je u ovom slučaju **RentalApp**. Svaki **service** element može da sadrži jedan ili više **endpoint** elemenata, pri čemu svaki od njih specificira određenu krajnju tačku, preko koje se može pristupiti **WCF** servisu. U ovom primeru, servis izlaže samo jednu krajnju tačku, zbog čega se pojavljuje samo jedano **endpoint** element. Ime ugovora krajnje tačke je **IReservations**, što je zapravo ime interfejsa koji definiše ugovor. Tip povezivanja koje se ovde koristi je **basicHttpBinding**. Ukoliko se pretpostavi da je servis hostovan koristeći **IIS** ili **WAS**, adresa se kreira automatski, tako da je nije potrebno specificirati unutar konfiguracionog fajla. Međutim, kao što pokazuje i ovaj primer, eksplicitno navođenje adrese je dozvoljeno.

Primer upotrebe krajnjih tačaka

Za ovaj primer vratićemo se na prvobitni opis rent-a-car aplikacije, gde se vrši komunikacija sa aplikacijama na različitim platformama. Kako bi se ispunili zahtevi raznolike komunikacije sa klijentima, aplikacija implementirana unutar **RentalReservations** servisne klase bi verovatno izložila nekoliko krajnjih tačaka, svaku sa različitim tipom povezivanja:

- Za komunikaciju sa klijentskom aplikacijom call centra zahtevaju se visoke performanse i puna funkcionalnost, koja uključuje jaku bezbednost i transakcije. Kako su i **RentalReservations** servisna klasa i call centar izgrađeni na **WCF**-u, dobar tip povezivanja za krajnju tačku kojoj pristupa ovaj klijent bi bio **NetTcpBinding**.
- Za komunikaciju sa **Java EE**-baziranom aplikacijom za rezervacije, zahteva se tip povezivanja koji koristi standardni **SOAP**. Ukoliko aplikacija i platforma na kojoj je pokrenuta ta aplikacija podržavaju neke ili sve **WS-*** specifikacije, tip povezivanja za krajnju tačku ovog klijenta bi mogao biti **WsHttpBinding**. Ovo bi omogućilo pouzdanu, bezbednu i transakcionu komunikaciju između dve aplikacije. Međutim, ukoliko aplikacija i platforma podržavaju samo standardni **SOAP**, krajnja tačka koju ta aplikacija koristi za pristup rent-a-car aplikaciji bi koristila **BasicHttpBinding**. Ako je potreban bezbedan transport, ovaj tip povezivanja bi mogao biti konfigurisan tako da koristi **HTTPS**.

- Za komunikaciju sa raznim partnerskim aplikacijama, mogle bi biti upotrebljene različite krajnje tačke u zavisnosti od potrebnog tipa povezivanja. Na primer, jednostavan **Web** servis klijent bi mogao da pristupi krajnjoj tački koja koristi **BasicHttpBinding**, dok bi oni klijenti koji podržavaju transportnu bezbednost koristili isti tip povezivanja konfigurisan za komunikaciju putem **HTTPS**-a. Partnerske aplikacije koje imaju sposobnost korišćenja **WS-*** tehnologija bi verovatno koristile **WsHttpBinding.**

Kreiranje WCF klijenta

Kreiranje jednostavnog WCF klijenta nije naročito komplikovano. U najjednostavnijem pristupu, sve što je potrebno uraditi jeste kreirati lokalnu zamenu servisa, **proxy**, koji je preko određene krajnje tačke povezan na ciljani servis, a zatim pozvati servisne operacije putem tog **proxy**-ja.

Kreiranje **proxy**-ja zahteva poznavanje ugovora koji se izlaže preko ciljane krajnje tačke, i upotrebu definicije tog ugovora. U **WCF**-u, ovaj proces je moguće izvesti korišćenjen **VS**-a ili **command line** alata **svcutil**. Ukoliko je servis implementiran korišćenjem **WCF**-a, ovi alati mogu da pristupe **DLL** fajlu servisa, **WSDL** definiciji servisa ili samom servisu, i da na osnovu njih kreiraju **proxy**.

Kako god da je generisan, klijent može da kreira novu instancu **proxy**-ja, i da na osnovu nje poziva metode servisa. Sledi primer klijenta za **RentalReservations** klasu, koji pravi rezervaciju kompaktnog automobila, na Heathrow aerodromu, u jesen 2009 godine:

```
using System.ServiceModel;

namespace RentalReservationsClient
{
     public class Program
     {
          static void Main(string[] args)
          {
               int confirmationNum;
               RentalReservationsProxy p = new RentalReservationsProxy();

               if (p.Check(1, 349, “9/30/09 - 10/10/09”))
               {
                    confirmationNum = p.Reserve(1, 349, “9/30/09 - 10/10/09”);
               }

               p.Close();
          }
     }
}
```

U ovom jednostavnom primeru kod za kompaktne automobile je **1**, lokacija aerodroma je označena sa **349**, a datum je dat na američki način. Sve što je protrebno jeste proveriti dostupnost automobila pozivom **Check** metode i rezervisati ga pozivanjem **Reserve** metode.

Preostaje da se specificira još jedna stvar od strane klijenta: tačna krajnja tačka, preko koje će se vršiti komunikacija. Klijent, kao i servis, mora da specificira adresu, povezivanje i ugovor, što se obično radi putem konfiguracionog fajla. Ukoliko je dostupno dovoljno informacija, **svcutil** alat će generisati odgovarajući klijentski konfiguracioni fajl za ciljani sevris.

Korišćenje **proxy**-ja je jednostavno, i često predstavlja pravi pristup. Međutim, to nije jedina opcija. Klijent može da komunicira direktno sa servisom, pomoću kanala. Korišćenjem **WCF ChannelFactory** klase, mogu se kreirati svi potrebni kanali, a zatim direktno pozivati servisi. Ovaj pristup nam daje više kontrole, ali je u isto vreme i kompleksniji. U nastavku je dat primer dve konzolne aplikacije, gde jedna aplikacija predstavlja servis, a druga direktno komunicira sa njim, putem kanala.

Server:

```
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace WCFServerApp
{
    public class Program
    {
        static void Main(string[] args)
        {
            Uri address = new Uri("http://localhost:4000/IWCFServer");
            BasicHttpBinding binding = new BasicHttpBinding();
            ServiceHost svc = new ServiceHost(typeof(WCFServer));
            svc.AddServiceEndpoint(typeof(IWCFServer), binding, address);
            svc.Open();
            Console.WriteLine("WCFServer ready");

            Console.ReadKey();
        }
    }

    public class WCFServer : IWCFServer
    {
        static Hashtable items = new Hashtable();

        public void Write(int id, double input)
        {
            items[id] = input;
            Console.WriteLine("Value written {0}={1} the body contains:{0}",
            id.ToString(), input.ToString());
        }

        public double Read(int id)
        {
            double value = -1;
            if (items.ContainsKey(id))
            {
                Console.WriteLine("Value read {0}={1} the body contains:{0}",
                    id.ToString(), value.ToString());
                value = (double)items[id];
            }

            return value;
        }
    }
```

```
    [ServiceContract]
    public interface IWCFServer
    {
        [OperationContract]
        void Write(int id, double input);

        [OperationContract]
        double Read(int id);
    }
}
```

Klijent:

```
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace WCFClientApp
{
    public class Program
    {
        static void Main(string[] args)
        {
            Uri address = new Uri("http://localhost:4000/IWCFServer");
            BasicHttpBinding binding = new BasicHttpBinding();

            ChannelFactory<IWCFServer> factory =
            new ChannelFactory<IWCFServer>(binding,
                new EndpointAddress(address));
            IWCFServer proxy = factory.CreateChannel();

            int id = 1; double writeValue = 100.4; double readValue;
            proxy.Write(id, writeValue);
            Console.WriteLine("Value written {0} = {1}", id.ToString(),
                writeValue.ToString());
            readValue = proxy.Read(id);
            Console.WriteLine("Value read {0} = {1}", id.ToString(),
                readValue.ToString());

            Console.ReadKey();
        }
    }

    [ServiceContract]
    public interface IWCFServer
    {
        [OperationContract]
        void Write(int id, double input);

        [OperationContract]
        double Read(int id);
    }
}
```

Upotreba metapodataka

U prethodnom primeru smo krenuli od činjenice da i server i klijent poznaju ugovor **IWCFServer**. Mada ovo u opštem slučaju nije nemoguće, uvodi ograničenje da se i klijenti moraju menjati svaki put kada se desi minimalna promena ugovora.

WS-MetadataExchange specifikacija opisuje kako server može da objavi format podataka koji se razmenjuje. Uobičajeno je da **WCF** ne objavljuje metapodatke. Ako je to potrebno, moguće je formirati krajnju tačku koja služi isključivo za razmenu metapodataka.

Prvi korak za pravljenje krajnje tačke za metapodatke je da se izmeni **ServiceHost**, kako bi pružao metapodake. To se radi tako što se objekat **ServiceMetadataBehavior** doda u **Behaviors** kolekciju **ServiceHost** objekta. **Behaviors** predstavlja specifičnu informaciju koju **WCF** koristi kako bi izmenio procesiranje lokalnih poruka. U sledećem koraku je neophodno definisati povezivanje krajnje tačke za metapodatke i odrediti adresu ove krajnje tačke, a zatim se krajnja tačka dodaje već postojećem **ServiceHost** objektu. Primer ovoga je dat u nastavku:

```
using System.ServiceModel.Description;

namespace WCFServerApp
{
     public class Program
     {
          static void Main(string[] args)
          {
               ServiceHost svc = new ServiceHost(typeof(WCFServer));
               ServiceMetadataBehavior metadata = new ServiceMetadataBehavior();
               svc.Description.Behaviors.Add(metadata);
               Binding mexBinding =
                   MetadataExchangeBindings.CreateMexHttpBinding();
               Uri mexAddress = new Uri("http://localhost:4000/IWCFServer/Mex");
               svc.AddServiceEndpoint(typeof(IMetadataExchange), mexBinding,
                   mexAddress);
               //More code...
          }
     }
}
```

Dodavanje krajnje tačke za metapodatke se može izvršiti i putem konfiguracionog fajla. Za servis iz prethodnog primera, kao i za **IIS/WAS** hostovane servise, to bi izgledalo ovako:

```
<system.serviceModel>
      <services>
            <service name="WCFServerApp.WCFServer">
                  <endpoint
                  contract="IWCFServer"
                  binding="basicHttpBinding"
                  address="http://localhost:4000/IWCFServer"/>
                  <endpoint
                  contract="IMetadataExchange"
                  binding="mexHttpBinding"
                  address="http://localhost:4000/IWCFServer/Mex"/>
            </service>
      </services>
</system.serviceModel>
```

Komunikacioni obrasci klijenata

U okviru ove sekcije će biti opisani različiti tipovi komunikacije koji se mogu odvijati između klijenta i krajnje tačke servisa.

One-Way

Jednosmerna(*eng. one-way*) komunikacija, kao što i samo ime kaže, predstavlja komunikaciju u jednom smeru. Taj smer teče od klijenta ka servisu. Sa servisa se ne šalje nikakav odgovor i klijent ne očekuje nikakav odgovor. U ovom scenariju, klijent šalje poruku i nastavlja izvršavanje. Sledi ilustracija jednosmerne komunikacje.

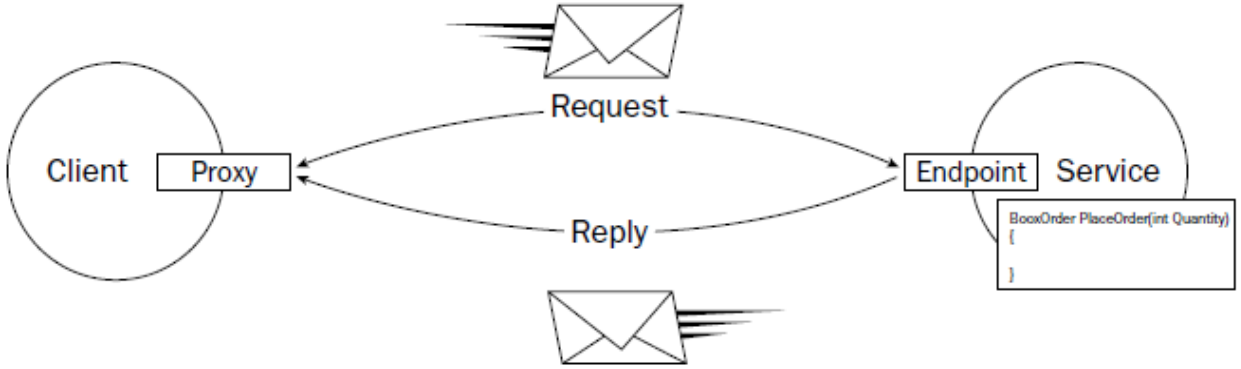
S obzirom da u jednosmernoj komunikaciji nema odgovora od strane servisa, klijentu neće biti javljeno ukoliko se na servisnoj strani dogodi greška, odnosno, klijent ne može znati da li je njegov zahtev bio uspešan ili ne.
Za jednosmernu komunikaciju se vrednost parametra **IsOneWay**, unutar **OperationContract** atributa, postavlja na t**rue**. Ovo govori servisu da nije potreban nikakav odgovor. Sledi primer podešavanja jednosmerne komunikacije.

```
[ServiceContract]
public interface IServiceClass
{
      [OperationContract(IsOneWay=true)]
      string InitiateOrder();

      [OperationContract]
      BookOrder PlaceOrder(BookOrder request);

      [OperationContract(IsOneWay=true)]
      string FinalizeOrder();
}
```

U ovom primeru, servis sadrži tri dostupne operacije. Operacije **InitiateOrder** i **FinalizeOrder** su definisane kao jednosmerne operacije, dok **PlaceOrder** nije. Kada klijent pozove **InitiateOrder** servisnu operaciju, odmah će nastaviti sa radom, bez čekanja na odgovor od strane servisa. Međutim, ukoliko klijent pozove **PlaceOrder** servisnu operaciju, čekaće na odgovor servisa pre nastavka izvršavanja.

Request-reply

Komunikacija bazirana na zahtevu i odgovoru(*eng. request-reply*) predstavlja komunikaciju gde klijent prilikom slanja poruke servisu, očekuje odgovor od strane servisa. Korišćenje ovog tipa komunikacije znači da klijent ne nastavlja sa izvršavanjem dok god ne dobije odgovor od servisa. U nastavku je data ilustracija ovog tipa komunikacije.



U **WCF**-u, pstoje dva načina za specificiranje **request-reply** komunikacije. Prva metoda je setovanem vrednosti **IsOneWay** parametra, unutar **OperationContract** atributa, na f**alse**. Ovo govori servisu da je odgovor neophodan.

Uobičajena vrednost za **IsOneWay** parametar je **false**, tako da druga metoda podrazumeva izostavljanje ovog atributa, prilikom čega će komunikacija svakako biti bazirana na zahtevu i odgovoru. Sledi primer podešavanja ovog tipa komunikacije.

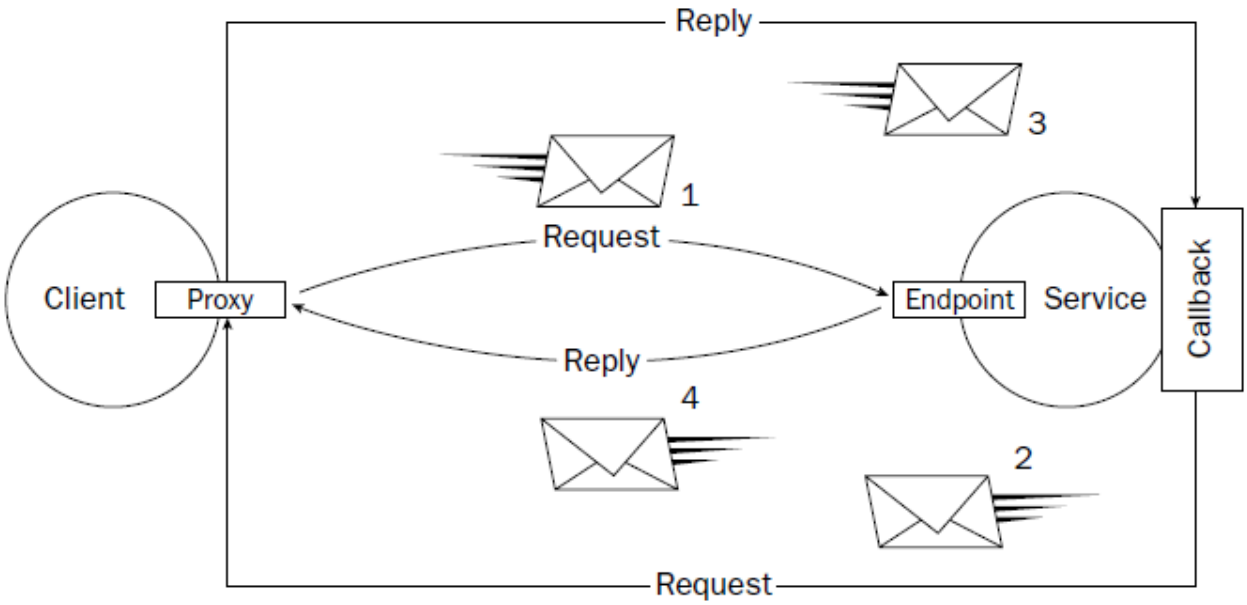
```
[ServiceContract]
public interface IServiceClass
{
	[OperationContract(IsOneWay=false)]
	string InitiateOrder();

	[OperationContract]
	BookOrder PlaceOrder(BookOrder request);

	[OperationContract(IsOneWay=true)]
	string FinalizeOrder();
}
```

U ovom primeru, servis sadrži tri dostupne operacije. Operacije **InitiateOrder** i **PlaceOrder** su definisane kao **request-reply** operacije, dok je **FinalizeOrder** definisana kao jednosmerna operacija. Operacija **InitiateOrder** je eksplicitno definisana kao **request-reply** operacija, postavljanjem vrednosti parametra **IsOneWay** na **false**, a metoda **PlaceOrder** je **request-reply** tipa, jer nije specificirana metoda komunikacije, što znači da ona uzima uobičajenu(**request-reply**) vrednost. Ukoliko se pozovu **InitiateOrder** ili **PlaceOrder** operacije, klijent će čekati na odgovor, dok će prilikom poziva **FinalizeOrder** operacije samo nastaviti sa izvršavanjem.

Duplex

Dupleks(*eng. duplex*) ili dvosmerna komunikacija predstavlja sposobnost, kako klijenta, tako i servisa, da inicira komunikaciju, kao i da odgovori na dolazeće poruke. Koristeći dupleks komunikacju, servis ne samo da može da odgovori na dolazeće poruke, već može i da inicira komunikaciju sa klijentom, slanjem zahteva i traženjem odgovora od strane klijenta.

Klijentska komunikacija sa servisom se prilikom dupleks komunikacje ne menja(i dalje se u tu svrhu koristi **proxy**). Međutim, servis komunicira sa klijentom putem povratnog poziva(*eng. callback*), kao što je prikazano u narednoj ilustraciji.



Kako bi povratni pozivi bili omogućeni, tipovi povezivanja moraju podržavati dvosmernu komunikaciju, što znači da se za ove vrste operacija ne može upotrebiti bilo koji tip povezivanja. Na primer, **BasicHttpBinding** ili **WsHttpBinding** ne podržavaju povratne pozive, dok ih **WsDualHttpBinding** podržava, jer formira dva **HTTP** kanala za komunikaciju, jedan za komunikaciju od klijenta ka servisu i drugi za komunikaciju od servisa ka klijentu.

Callback servis se definiše upotrebom **CallbackContract** parametra, unutar **ServiceContract** atributa. Dozvoljen je samo jedan **callback** ugovor po servisnom ugovoru. Nakon definisanja **callback** ugovora, potrebno je podesiti klijenta, kako bi komunikacija od servisa ka klijentu bila moguća.

Self-hosted duplex

U narednom primeru, **callback** ugovor definiše operacije koje servis može da pozove putem izložene krajnje tačke klijenta.

```
public interface INotificationServiceCallBack
{
    [OperationContract(IsOneWay = true)]
    void OnNotificationSend(string message);
}
```

U nastavku je data implementacija servisnog ugovora, koji podržava povratne pozive.

```
[ServiceContract(CallbackContract = typeof(INotificationServiceCallBack))]
public interface INotificationService
{
    [OperationContract]
    void SendNotification(string message);
}

public class NotificationService : INotificationService
{
    public INotificationServiceCallBack Proxy
    {
        get
        {
            return OperationContext.Current.GetCallbackChannel
            <INotificationServiceCallBack>();
        }
    }

    public void SendNotification(string message)
    {
        Console.WriteLine("\nClient says :" + message);
        Proxy.OnNotificationSend("Yes");
    }
}
```

U prethodnom primeru se može videti da servisna klasa ima povratni kanal **INotificationServiceCallBack Proxy**, i da poziva njegove operacije, kojima će se pozabaviti klijent. **Host** proces i njegova konfiguracija su dati u nastavku.

```
class Program
{
    static void Main(string[] args)
    {
        var svcHost = new ServiceHost(typeof(NotificationService));
        svcHost.Open();

        Console.ReadKey();
    }
}
```

```
<system.serviceModel>
    <services>
      <service name="DuplexWcf.NotificationService">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8888"/>
          </baseAddresses>
        </host>
        <endpoint address="CallbackService" binding="wsDualHttpBinding"
            contract="DuplexWcf.INotificationService"/>
        <endpoint address="mex" binding="mexHttpBinding"
            contract="IMetadataExchange" />
      </service>
    </services>
</system.serviceModel>
```

Kako bi se stvorio način putem kojeg servis može da obavi povratni poziv, klijent mora da izloži krajnju tačku ka servisu. Ovo je odrađeno na sledeći način, koristeći klasu **InstanceContext**:

```
public class NotificationServiceCallBack : INotificationServiceCallBack
{
    public void OnNotificationSend(string message)
    {
        Console.WriteLine(message);
    }
}

public class Program
{
    public static INotificationServices Proxy
    {
        get
        {   var ctx = new InstanceContext(new NotificationServiceCallBack());
            return new DuplexChannelFactory<INotificationServices>
               (ctx,"WSDualHttpBinding_INotificationServices").CreateChannel();
        }
    }

    static void Main(string[] args)
    {
        Proxy.SendNotification("Are u ready?");
        Console.ReadKey();
    }
}
```

Konfiguracija klijentske aplikacije:

```
<system.serviceModel>
   <client>
      <endpoint address="http://localhost:8888/CallbackService"
          binding="wsDualHttpBinding"
          contract="DuplexWcf.INotificationServices"
          name="WSDualHttpBinding_InotificationServices"/>
    </client>
</system.serviceModel>
```

IIS-hosted duplex

Server:

```
public interface INotificationServiceCallBack
{
    [OperationContract(IsOneWay = true)]
    void OnNotificationSend(string message);
}

[ServiceContract(CallbackContract = typeof(INotificationServiceCallBack))]
public interface INotificationServices
{
    [OperationContract]
    void SendNotification(string message);
}
--------------------------------------------------------------------------------

public class NotificationService : INotificationService
{
    public INotificationServiceCallBack Proxy
    {
        get
        {
            return OperationContext.Current.GetCallbackChannel
            <INotificationServiceCallBack>();
        }
    }

    public void SendNotification(string message)
    {
        Console.WriteLine("\nClient says :" + message);
        Proxy.OnNotificationSend("Yes");
    }
}
--------------------------------------------------------------------------------

<system.serviceModel>
      <services>
            <service name="DuplexWcf.NotificationService">
                  <endpoint
                  contract="INotificationService"
                  binding="wsDualHttpBinding"
                  address="CallbackService"/>
                  <endpoint
                  contract="IMetadataExchange"
                  binding="mexHttpBinding"
                  address="CallbackService/Mex"/>
            </service>
      </services>
</system.serviceModel>
```

Klijent:

```
public class ClientCallback : INotificationServiceCallBack
{
      public void OnNotificationSend(string message)
      {
            Console.WriteLine(message);
      }
}
-------------------------------------------------------------------------------

public class Program
{
    static void Main(string[] args)
    {
        InstanceContext ic = new InstanceContext(new ClientCallback());
        ServiceReference1.NotificationServiceClient proxy =
            new  ServiceReference1.NotificationServiceClient(ic);
        proxy.SendNotification("Are u ready?");
        Console.ReadLine();
    }
}
```

Uz nekoliko izmena, prethodni primer se može upotrebiti za vršenje komunikacije između dve konzolne aplikacije, pri čemu prva aplikacija konstantno šalje poruke, a druga ih prima i obrađuje. U ovom slučaju, prva aplikacija predstavlja izdavača (*eng. publisher*), a druga pretplatnika (*eng. subscriber*). Zbog jednostavnijeg shvatanja ovog koncepta komunikacije, svaka aplikacija(klijent) će imati svoj servisni ugovor. Primer ovog koda se nalazi u dodatnom pdf materijalu.

Sesije

U WCF aplikacijama, sesije uvezuju grupu poruka u konverzaciju, odnosno kreiraju deljeno stanje, koje omogućava korišćenje mnogih korisnih opcija, poput povratnih poziva. WCF sesije se umnogome razlikuju od ASP.NET sesija. Osnovna uloga WCF sesija jeste da održavaju servisnu instancu živom, kako bi se serija poruka uvek slala i obrađivala u okviru iste servisne instance. Osnovni koncepti iza WCF sesija su:

- Sesije se eksplicitno iniciraju i završavaju od strane klijentkih aplikacija.
- Poruke koje dospevaju na servis za vreme trajanja sesije se obrađuju u redosledu u kom su pristigle na servis.
- Uvezivanje poruka je apstraktan pojam i vrši se na različite načine. Neki kanali bazirani na sesijama (*eng. session-based channels*) uvezuju poruke na osnovu konekcije ka servisu, dok drugi kanali mogu da uvezuju poruke na osnovu deljenog taga u telu poruka.
- Ne postoji nikakvo skladište podataka koje se vezuje za sesiju.

Kada se unutar servisnog ugovora specificira da on zahteva postojanje sesije (koristeći **ServiceContractAttribute.SessionMode** svojstvo), taj ugovor zapravo specificira da svi pozivi ka servisu moraju biti deo iste konverzacije. Ukoliko se unutar ugovora specificira da su sesije dozvoljene, ali ne i obavezne, klijenti mogu da se povežu na servis i odaberu da li će uspostaviti sesiju ili ne. Pored ove dve opcije, unutar servisnog ugovora može biti specificirana i izričita zabrana postojanja sesije. Uobičajena opcija je **SessionMode.Allowed**, odnosno o uspostavljanju sesije odlučuju klijenti.

Kada servis prihvati klijentsku sesiju, svi pozivi između klijentskog objekta i servisa se obrađuju u okviru iste servisne instance. U slučaju završetka sesije i slanja nove poruke preko istog kanala (baziranog na sesijama) dolazi do bacanja izuzetka.

Povezivanje koje pokušava da inicira sesiju se naziva povezivanje bazirano na sesijama (*eng. session-based binding*). Ove vrste povezivanja omogućavaju vezivanje servisne instance za određenu sesiju. Uz to, različiti tipovi povezivanja bazirnih na sesijama podržavaju i različite dodatne mogućnosti, u zavisnosti od tipa sesije koji uspostavljaju. WCF nudi sledeće tipove sesija:

- **Zaštićene sesije** - Enkriptuju i digitalno potpisuju poruke.
- **TCP/IP (transportne) sesije** - Staraju se o tome da sve poruke budu uvezane na osnovu konekcije na nivou socket-a.
- **Pouzdane sesije** - Staraju se o tome da poruke koje se šalju kroz više čvorova stignu tačno jedanput i, opciono, u istom redosledu u kojem su i poslate.
- **MSMQ datagram sesije** - Omogućavaju upotrebu **MMQ** (Microsoft Message Queuing) transporta.

Postavljanjem **SessionMode** svojstva se ne specificira koji tip sesije se koristi. Ovo svojstvo samo govori WCF-u da iskonfigurisani tip povezivanja mora, ne sme, ili može da uspostavi sesiju prilikom komunikacije sa servisom. Ukoliko se zahteva uspostavljanje sesije, povezivanje može da zadovolji ovaj zahtev uspostavljajući bilo koji tip sesije - zaštićeni, transportni, pouzdani ili neku od kombinacija navedenih tipova. Ukoliko tip povezivanja nije u skladu sa vrednošću **SessionMode** svojstva, dolazi do bacanja izuzetka.

Naredni servisni ugovor specificira da sve operacije iz **ICalculatorSession** interfejsa moraju da se izvrše u okviru iste sesije. Nijedna od operacija ne vraća vrednost pozivaocu osim metode **Equals**. Međutim, **Equals** metoda ne prihvata nikakve parametre i zbog toga može da vrati vrednost različitu od nule samo u okviru sesije u okviru koje su parametri već prosleđeni drugim operacijama. Ovaj ugovor zahteva da sesija funkcioniše ispravno. Bez sesije vezane za odgovarajućeg klijenta, servisna instanca nema načina da dođe do podataka koje je klijent prethodno poslao.

```
[ServiceContract(SessionMode=SessionMode.Allowed)]
public interface ICalculatorSession
{
    [OperationContract(IsOneWay=true)]
    void Clear();

    [OperationContract(IsOneWay = true)]
    void AddTo(double n);

    [OperationContract(IsOneWay = true)]
    void SubtractFrom(double n);

    [OperationContract(IsOneWay = true)]
    void MultiplyBy(double n);

    [OperationContract(IsOneWay = true)]
    void DivideBy(double n);

    [OperationContract]
    double Equals();
}
```

Ukoliko je upotreba sesija dozvoljena, sesija se uspostavlja samo ukoliko je klijent inicira. U suprotnom, sesija se ne uspostavlja.

U nastavku je data servisna klasa, koja implementira prethodno navedeni servisni ugovor, kao i klijent koji koristi usluge ovog servisa. Klijent će uspostaviti sesiju samo ukoliko se koristi povezivanje koje podržava uspostavljanje sesije. U suprotnom, sesija neće biti uspostavljena i poziv Equals operacije će uvek vraćati nulu.

```
public class CalculatorSession : ICalculatorSession
{
    double result;

    public void Clear()
    {
        result = 0;
    }

    public void AddTo(double n)
    {
        result += n;
    }

    public void SubtractFrom(double n)
    {
        result -= n;
    }

    public void MultiplyBy(double n)
    {
        result *= n;
    }

    public void DivideBy(double n)
    {
        result /= n;
    }

    public double Equals()
    {
        return result;
    }
}
```

Klijent:

```
class Program
{
    static void Main(string[] args)
    {
        ServiceReference.CalculatorSessionClient client =
             new ServiceReference.CalculatorSessionClient();

        client.AddTo(8);
        client.MultiplyBy(5);
        client.DivideBy(2);
```

```
        double result = client.Equals();

        Console.WriteLine($"Result is: {result}");

        Console.ReadKey();
    }
}
```

Sesije i servisne instance

U slučajevima kada se za instanciranje koristi uobičajeno ponašanje, svi pozivi između WCF klijentskog objekta i servisa se obrađuju od strane iste servisne instance. Ovo ponašanje liči na pozivanje metoda nad lokalnom instancom objekta, koje bi podrazumevalo sledeće korake:

- Poziva se konstruktor
- Svi pozivi se obrađuju od strane iste instance objekta
- Nakon brisanja reference na objekat, poziva se destruktor

Sesije omogućavaju slično ponašanje između klijenata i servisa, dok god se koristi uobičajeno ponašanje za servisne instance.

Ukoliko servisni ugovor zahteva ili podržava upotrebu sesija, jedna ili više operacija unutar ugovora mogu biti obeležene kao inicijalne (eng. initiating) ili završne (eng. terminating) operacije za sesiju, postavljanjem **IsInitiating** i **IsTerminating** svojstava. Uobičajene vrednosti za ova dva svojstva su **true** i **false**, respektivno.

Inicijalne operacije su one operacije koje moraju biti pozvane kao prve operacije unutar nove sesije. Neinicijalne operacije mogu biti pozvane tek nakon što je pozvana barem jedna inicijalna operacija. Definisanjem inicijalnih operacija, koje od klijenta preuzimaju određeni skup podataka za servisnu instancu, može da se kreira neka vrsta konstruktora za servis.

Završne operacije su one operacije koje moraju biti pozvane kao poslednje operacije unutar postojeće sesije. U uobičajenom slučaju, WCF uništava servisni objekat i njegov kontekst nakon zatvaranja sesije sa kojom je objekat bio asociran. Definisanjem završnih operacija, koje izvršavaju odgovarajuće operacije na kraju životnog veka servisne instance, može da se kreira neka vrsta destruktora za servis.

Iako postoje sličnosti sa upotrebom lokalne instance objekta, postoje i neke razlike. Svaka WCF servisna operacija može biti inicijalna ili završna, ili i jedno i drugo u isto vreme. Pored toga, inicijalne operacije mogu da budu pozvane bilo koji broj puta i u bilo kom redosledu. Naknadni pozivi inicijalnih operacija ne dovode do uspostavljanja nove sesije, nakon što je već ostvarena asocijacija sesije sa klijentskom instancom (osim u slučajevima kada se životni vek servisne instance kontroliše eksplicitno, upotrebom **InstanceContext** objekta).

Na primer, izmenjeni servisni ugovor **ICalculatorSession** dat u nastavku zahteva pozivanje **Clear** operacije od strane klijenta, pre bilo koje druge operacije. Takođe, u okviru ugovora je navedeno i da sesija sa trenutnim WCF klijentom treba da se završi ukoliko klijent pozove **Equals** operaciju.

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface ICalculatorSession
{
    [OperationContract(IsOneWay=true, IsInitiating=true, IsTerminating=false)]
    void Clear();

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void AddTo(double n);

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void SubtractFrom(double n);

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void MultiplyBy(double n);

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void DivideBy(double n);

    [OperationContract(IsInitiating = false, IsTerminating = true)]
    double Equals();
}
```

Servisi ne započinju sesije sa klijentima. U WCF klijentskim aplikacijama postoji direktna povezanost između životnog veka kanala baziranog na sesijama i životnog veka same sesije. Zbog toga, klijenti mogu da kreiraju novu sesiju kreiranjem novog kanala baziranog na sesijama i unište postojeću sesiju zatvaranjem kanala baziranog na sesijama. Klijent započinje sesiju sa krajnjom tačkom servisa pozivajući jednu od sledećih metoda:

- **ICommunicationObject.Open** nad novim kanalom vraćenim od strane **ChannelFactory<TChannel>.CreateChannel** metode.
- **ClientBase<TChannel>.Open** nad WCF klijentskim objektom izgenerisanim od strane ServiceModel Metadata Utility Tool-a (Svcutil.exe).
- Iniciajlne metode nad bilo kojim tipom WCF klijentskog objekta. Kada se pozove prva metoda, WCF klijentski objekat automatski otavara kanal i inicijalizuje sesiju.

Klijent zatvara sesiju sa servisnom tačkom pozivajući jednu od sledećih metoda:

- **ICommunicationObject.Close** nad novim kanalom vraćenim od strane **ChannelFactory<TChannel>.CreateChannel** metode.
- **ClientBase<TChannel>.Close** nad WCF klijentskim objektom izgenerisanim od strane ServiceModel Metadata Utility Tool-a (Svcutil.exe).
- Završne metode nad bilo kojim tipom WCF klijentskog objekta.

U nastavku je dat primer klijenta koji uspostavlja sesiju prilikom komunikacije sa servisom (koji implementira novi **ICalculatorSession** servisni ugovor).

```
class Program
{
    static void Main(string[] args)
    {
        ServiceReference.CalculatorSessionClient client =
            new ServiceReference.CalculatorSessionClient();

        // ((0 + 8)*5)/2 = 20
        client.Clear(); // OBAVEZNO U OVOM SLUČAJU
```

```
            client.AddTo(8);
            client.MultiplyBy(5);
            client.DivideBy(2);

            double result = client.Equals();

            Console.WriteLine($"Result is: {result}");

            Console.ReadKey();
        }
}
```

Ukoliko se pre inicijalne **Clear** operacije pozove bilo koja druga operacija, doći će do bacanja izuzetka. Ukoliko se nakon poziva **Equals** operacije, koja je označena kao završna, preko istog servisnog klijenta pozove neka nova operacija,takođe će doći do bacanja izuzetka.
SessionMode enumeracija je usko povezana sa **ServiceBehaviorAttribute.InstanceContextMode** svojstvom, koje je opisano u nastavku.

Instanciranje

Ponašanje servisa prilikom instanciranja (koje se podešava postavljanjem **ServiceBehaviorAttribute.InstanceContextMode** svojstva nad samom servisnom klasom) kontroliše kako se **InstanceContext** kreira prilikom odgovora na dolazeće poruke. Uobičajeno je da jedan **InstanceContext** objekat bude vezan za jedan servisni objekat, tako da se postavljanjem **InstanceContextMode** svojstva zapravo kontroliše intanciranje servisnih objekata. **InstanceContextMode** enumeracija definiše dostupne modove instanciranja. Dostupni su sledeći modovi:

- **PerCall** - Novi **InstanceContext** (a samim tim i servisni objekat) se kreira za svaki klijentski zahtev.
- **PerSession** - Novi **InstanceContext** (a samim tim i servisni objekat) se kreira za svaku klijentsku sesiju i održava se tokom životnog veka sesije (ovo zahteva povezivanje koje podržava sesije). Ovaj mod je uobičajen.
- **Single** - Jedan **InstanceContext** (a samim tim i servisni objekat) rukuje svim klijentskim zahtevima tokom životnog veka aplikacije.

Jedna od varijanti servisa sa samo jednim servisnim objektom (**InstanceContextMode.Single**) podrazumeva ručno kreiranje instance servisnog objekta i prosleđivanje te instance **ServiceHost(Object, Uri[])** konstruktoru (alternativni pristup zahteva implementaciju proizvoljnog **System.ServiceModel.Dispatcher.IInstanceContextInitializer-a**). Na ovaj način je moguće kreirati servise za servisne klase koje je teško instancirati (na primer, ne mogu da implementiraju konstruktor bez parametara, koji je neophodan za kreiranje servisne instance).

Interakcija između sesija i instanciranja zavisi od kombinacije vrednosti SessionMode enumarcije i **InstanceContextMode** svojstva, koje kontroliše vezu između kanala i određenog servisnog objekta. Naredna tabela prikazuje ponašanje servisa za različite kombinacije ova dva parametra, pri čemu **1)** označava ponašanje sa kanalom koji podržava sesije, a **2)** ponašanje sa kanalom koji ne podržava sesije:

InstanceContextMode	SessionMode.Required	SessionMode.Allowed	SessionMode.NotAllowed
Per Call	1) Sesija i IntanceContext za svaki poziv 2) Izuzetak	1) Sesija i IntanceContext za svaki poziv 2) InstanceContext za svaki poziv	1) Izuzetak 2) IntanceContext za svaki poziv
Per Session	1) Sesija i InstanceContext za svaki kanal 2) Izuzetak	1) Sesija i InstanceContext za svaki kanal 2) IntanceContext za svaki poziv	1) Izuzetak 2) InstanceContext za svaki poziv
Single	1) Sesija i jedan InstanceContext za sve pozive 2) Izuzetak	1) Sesija i jedan InstanceContext za sve pozive 2) Jedan IntanceContext za sve pozive	1) Izuzetak 2) Jedan IntanceContext za sve pozive

U nastavku je dat primer servisne klase sa postavljenom vrednošću **InstanceContextMode** svojstva. Kako je vrednost ovog svojstva postavljena na **InstanceContextMode.PerCall**, svaki poziv ka servisu dovodi do kreiranja nove servisne instance. Data servisna klasa implementira servisni ugovor sa jednom operacijom, koji koristi uobičajenu vrednost **SessionMode** svojstva, tako da će uspostavljanje sesije zavisiti od tipa povezivanja koje se koristi (**basicHttpBinding** ili **wsHttpBinding**). Menjanje vrednosti **InstanceContextMode** i **SessionMode** svojstava, kao i tipa povezivanja, dovodi do različitih vidova ponašanja servisa, navedenih u prethodnoj tabeli.

```
[ServiceContract(SessionMode=SessionMode.Allowed)]
public interface IInstanceModeInterface
{
    [OperationContract(IsOneWay = true)]
    void CallMethod();
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall]
public class InstanceModeClass : IInstanceModeInterface
{
    static int instanceCount;
    static readonly object instanceCountlocker = new object();
    int instanceId;
    int callCount;

    public InstanceModeClass()
    {
        lock (instanceCountlocker)
        {
            instanceCount++;
            instanceId = instanceCount;
        }
    }
    public void CallMethod()
    {
        callCount++;
        localCount = callCount;
```

```
            Console.WriteLine($"Instance with ID {instanceId}: " +
                $"current callCount value is {localCount} {DateTime.Now}");

            Thread.Sleep(1000);
        }
    }
```

U nastavku je dat i primer konzolne aplikacije, koja kreira više klijentskih instanci i šalje pozive ka prethodno implementiranom servisu.

```
class Program
{
    static void Main(string[] args)
    {
        ServiceReference.InstanceModeInterfaceClient client1 =
            new ServiceReference.InstanceModeInterfaceClient();

        ServiceReference.InstanceModeInterfaceClient client2 =
            new ServiceReference.InstanceModeInterfaceClient();

        CallMethodHandler(client1);
        CallMethodHandler(client2);

        Console.ReadKey();
    }

    static void CallMethodHandler(ServiceReference.InstanceModeInterfaceClient client)
    {
        for (int i = 0; i < 5; i++)
        {
            client.CallMethod();
        }
    }
}
```

Konkuretnost

Konkurentnost predstavlja kontrolu broja niti koje su aktivne unutar servisnog objekta u svakom trenutku. Kontrola se vrši postavljanjem **ServiceBehaviorAttribute.ConcurrencyMode** svojstva. **ConcurrencyMode** enumeracija definiše dostupne modove konkurentnosti. Dostupni su sledeći modovi:

- **Single** - Svaki servisni objekat može da ima samo jednu nit koja obrađuje pristigle poruke u svakom trenutku. Ostale niti koje žele da pristupe istom servisnom objektu moraju da budu blokirane dok originalna nit ne napusti servisni objekat.
- **Multiple** - Svaki servisni objekat može da ima više niti koje konkurentno obrađuju pristigle poruke. Implementacija servisne klase mora biti thread-safe kako bi se koristio ovaj mod.
- **Reentrant** - Svaki servisni objekat može da obrađuje samo jednu poruku u svakom trenutku. Dok se poruka obrađuje, nijedna druga poruka ne može da pristupi servisnom objektu. Ukoliko se tokom izvršavanja pozvane operacije desi poziv ka drugom servisu (preko WCF kanala), trenutna poruka gubi ekskluzivno pravo upotrebe servisnog objekta i njegova nit se otključava za druge poruke. Kada se završi pozvana operacija na drugom servisu, nit se ponovo zaključava i originalna poruka može da nastavi sa obradom do svog završetka ili novog poziva ka drugom servisu.

Upotreba konkurentnosti je povezana sa modom instanciranja. U InstanceContextMode.PerCall modu konkuretnost nije relevantna, jer se svaka poruka obrađuje unutar zasebne servisne instance. U InstanceContextMode.Single modu je relevantan ili Single ili Multiple mod konkurentnosti, u zavisnosti od toga da li jedna servisna instanca poruke obrađuje sekvencijalno ili konkurentno. U InstanceContextMode.PerSession modu bilo koji od modova konkurentnosti može biti relevantan, ali tačno ponašanje zavisi od toga da li se uspostavlja sesija ili se za svaki poziv pravi nova servisna instanca.

Primer servisne klase iz prethodne sekcije može biti dopunjen, tako da posluži za prikazivanje ponašanja servisa u zavisnosti od postavljene kombinacije moda instanciranja i konkurentnosti.

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
      ConcurrencyMode = ConcurrencyMode.Multiple)]
public class InstanceModeClass : IInstanceModeInterface
{
    static int instanceCount;
    static readonly object instanceCountlocker = new object();
    int instanceId;
    int callCount;
    object countlocker = new object();


    public InstanceModeClass()
    {
        lock (instanceCountlocker)
        {
            instanceCount++;
            instanceId = instanceCount;
        }
    }

    public void CallMethod()
    {
        int localCount;

        lock (countlocker)
        {
            callCount++;
            localCount = callCount;
        }

        Console.WriteLine($"Thread with Id {Thread.CurrentThread.ManagedThreadId} " +
             $"Instance with ID {instanceId}: " +
             $"current callCount value is {localCount} {DateTime.Now}");

        Thread.Sleep(1000);
    }
}
```

Servisni ugovor i klijentska aplikacija su nepromenjeni u odnosu na prethodni primer.