System.Transactions in .Net 2.0

Auteur
Edwin van Wijk
Info Support
edwinw@infosupport.nl

Publicatie
22-07-2007
SDN Magazine

Samenvatting
Versie 2.0 van het .Net framework biedt ontwikkelaars functionaliteit om op een eenvoudige manier transactionele applicaties te bouwen. Deze functionaliteit is terug te vinden in de namespace *System.Transactions*. In dit artikel wil ik een overzicht geven van een deel van de mogelijkheden die deze namespace ons biedt en aan de hand van voorbeeldcode inzicht verschaffen in de manier waarop deze mogelijkheden kunnen worden benut.

Zoals hoogstwaarschijnlijk bekend is, levert versie 2.0 van het .Net framework een scala aan nieuwe functionaliteiten. Een voorbeeld van een namespace die in versie 2.0 is verschenen is *System.Transactions*. Zoals de naam reeds doet vermoeden bevat deze namespace functionaliteit die gebruikt kan worden bij het ontwikkelen van transactionele applicaties. In dit artikel wil ik een overzicht geven van een deel van de mogelijkheden die deze namespace ons biedt. Daarnaast wil ik aan de hand van voorbeeldcode inzicht verschaffen in de manier waarop de mogelijkheden kunnen worden benut.

Even opfrissen

Elke ontwikkelaar die wel eens een applicatie heeft gebouwd waarbij een database werd gebruikt voor de opslag van gegevens, heeft hoogstwaarschijnlijk te maken gehad met transacties. Maar waarom waren transacties ook al weer handig om te hebben? In het kort komt het er op neer dat wanneer we gegevens opslaan in een database we er zeker van willen zijn dat alle gegevens correct in de database terechtkomen. We willen graag dat ook wanneer een bepaalde handeling uit meerdere stappen bestaat, alle stappen als één geheel worden doorgevoerd of – in het geval van een fout – teruggedraaid. Het is van belang dat halverwege het uitvoeren van een transactie niet een andere gebruiker dezelfde gegevens kan gaan aanpassen. Als een transactie eenmaal is doorgevoerd, moeten we er ook vanuit kunnen gaan dat deze gegevens in de database beschikbaar zijn, zelfs nadat de machine waarop de database draait om wat voor reden dan ook is herstart. Al deze eigenschappen worden samen vaak aangeduid als de “ACID” eigenschappen van een transactie. Dit acroniem staat voor:

- Atomic (atomair)
  Alle stappen als één geheel uitgevoerd of teruggedraaid.
- Consistent
  Voor en na elke transactie is de database consistent.
- Isolated (geïsoleerd)
  Elke transactie gedraagt zich als was hij de enige in het systeem*.
- Durable (duurzaam)
  Als een transactie is doorgevoerd moeten de wijzigingen ook doorgevoerd blijven als het systeem crasht.

* Hierin zijn verschillende niveaus te onderkennen (zogenaamde isolation-levels). Het isolation level van een transactie geeft aan in hoeverre gegevens die binnen die transactie worden geraakt, gedurende de transactie kunnen worden benaderd door andere gebruikers.

Een nieuw programmeermodel

Het .Net 1.x framework biedt een aantal mogelijkheden om met transacties te werken. Zo is het mogelijk om bijvoorbeeld een ADO.Net transactie te starten middels een aanroep van de *BeginTransaction* methode op een *SqlConnection* object (de imperatieve methode). In dit geval is de ontwikkelaar zelf verantwoordelijk voor het coördineren van de transactie. Dit wordt ingewikkelder op het moment dat er bijvoorbeeld als onderdeel van de transactie meerdere type resources moeten worden benaderd. Een voorbeeld hiervan is het doen van een insert in een database en het plaatsen van een message in een transactionele message queue (zie figuur 1). De ontwikkelaar zal er in dit geval zelf voor moeten zorgen dat de MSMQ resourcemanager mee gaat draaien in de lopende transactie. Met andere woorden, er moet een gedistribueerd transactiemanagement systeem gemaakt worden. Om dit wat makkelijker te maken biedt .Net 1.x naast het imperatieve model ook een declaratief model, waarbij een

klasse die afleidt van *ServicedComponent* kan worden voorzien van een *Transaction* attribuut. De methoden die vervolgens worden aangeroepen op deze klasse zullen dan automatisch in een transactie worden uitgevoerd. De transactionele resourcemanagers die worden gebruikt in deze methoden zullen dan ook automatisch worden aangemeld bij de lopende transactie. Ondanks dat deze manier van werken het gebruik van transacties vergemakkelijkt, brengt dit ook enkele nadelen met zich mee. Er zal bijvoorbeeld altijd gebruikgemaakt worden van de MSDTC (ook als een lokale transactie voldoende is), hetgeen een performance degradatie met zich meebrengt. En wat te denken van het feit dat het afleiden van *ServicedComponent* het afleiden van een andere klasse verhindert.

De manier waarop kan worden gewerkt met transacties in .Net 1.x zoals hierboven beschreven, is nog steeds mogelijk in .Net 2.0. Maar om de bovengenoemde nadelen weg te werken, is in .Net 2.0 een nieuw programmeermodel geïntroduceerd. Dit model is gebaseerd op z.g. transactie 'scopes'. Een ontwikkelaar kan een bepaald stuk code binnen een transactie scope plaatsen, waarna de *System.Transactions* infrastructuur er vervolgens voor zorgt dat alle SQL Server, ADO.Net, MSMQ en MSDTC transacties die binnen de scope vallen automatisch als één grote transactie worden doorgevoerd of teruggedraaid. Nadat alle benodigde acties binnen de scope zijn uitgevoerd, geeft de ontwikkelaar aan dat de wijzigingen binnen de scope kunnen worden doorgevoerd. Omdat dit de laatste stap binnen de scope is, zal deze worden overgeslagen als er een exception optreedt met als gevolg dat de transactie wordt teruggedraaid. Een voorbeeld van het gebruik van een transactiescope is te zien in listing 1. Het is raadzaam gebruik te maken van de *using* constructie zoals is te zien in het voorbeeld. Dit zorgt ervoor dat wanneer de scope is afgerond, deze netjes wordt opgeruimd en de transactie eventueel wordt teruggedraaid als de *Complete* methode niet is aangeroepen.

In listing 1 is te zien hoe eenvoudig het is om een insert in de database transactioneel binnen een transactiescope uit te voeren. De ontwikkelaar hoeft zelf geen expliciete transactie te starten (of af te leiden van *ServicedComponent*) zoals dat in .Net 1.x het geval is. Binnen de transactiescope is er altijd een zogenaamde 'ambient' transactie die is op te vragen middels de static *Current* property van de *Transaction* klasse. De kracht van dit impliciete programmeermodel komt echter nog beter naar voren wanneer we aan het voorbeeld een additionele stap toevoegen. We gaan binnen dezelfde transactiescope een bericht in een transactionele message-queue stoppen (zie figuur 1). Om dit mogelijk te maken moet Microsoft Message Queueing geïnstalleerd zijn op de machine (workgroup mode is voldoende) en een private transactionele queue beschikbaar zijn. In het voorbeeld heet de queue 'TxTestQueue'. Volg de volgende stappen om deze queue aan te maken:

1. Open de Server Explorer in Vistual Studio 2005 (*View*, *Server Explorer*).
2. Klik rechts op de node: 'Servers\<*hostnaam*>\Message Queues\Private Queues'.
3. Kies de optie 'Create queue…'. Er zal een dialoog worden getoond.
4. Geef de queue een naam (bijvoorbeeld 'TxTestQueue'), vink de optie 'Make queue transactional' aan en klik op de 'Ok' knop.

De queue zal verschijnen onder de 'Servers\<*hostnaam*>\Message Queues\Private Queues' node (zie figuur 2). Nu is het mogelijk om binnen een transactiescope zowel een database insert te doen als een bericht in de queue te stoppen (zie listing 2). Als deze code is uitgevoerd en er geen fouten zijn opgetreden, zal de gehele transactie zijn doorgevoerd. Het record zal in de database verschijnen en het testbericht zal in de queue verschijnen (zie figuur 3). Indien er ergens in de transactiescope een exception zou optreden, wordt de transactie teruggedraaid en zal zowel de inhoud van de database als de inhoud van de queue ongewijzigd zijn.

Resourcemanagers

De term resourcemanager is al een aantal malen gevallen, maar wat zijn nu precies de verantwoordelijkheden van een resourcemanager? Een resourcemanager beheert een bepaald soort gegevens en zorgt ervoor dat deze gegevens transactioneel kunnen worden benaderd. Een voorbeeld van een resourcemanager dat vaak gebruikt wordt is de SQL Server

resourcemanager. Deze beheert databases, tabellen, enz. en biedt ontwikkelaars onder andere de mogelijkheid om wijzigingen in databaserecords als onderdeel van een transactie door te voeren. Een resourcemanager wordt aangestuurd door een transactiemanager. Dit kan een lokale of gedistribueerde transactiemanager zijn (hier kom ik later op terug). Verder zijn resourcemanagers er in twee smaken: volatile (vluchtig) en durable (duurzaam). Vluchtige resourcemanagers beheren gegevens die alleen in het geheugen worden opgeslagen. Hierbij is de 'durable' eigenschap van de transactie dus niet gewaarborgd. Dit wil echter niet zeggen dat volatile resourcemanagers minder nuttig zijn. Het is vaak prettig om zeker te weten dat een aantal handelingen altijd als een geheel wordt uitgevoerd. Denk bijvoorbeeld aan het aanpassen van een aantal items in een collectie. Als om wat voor reden dan ook de reeds aangebrachte wijzigingen ongedaan gemaakt moeten worden, is het uitvoeren van een rollback makkelijker dan het handmatig terugdraaien van alle wijzigingen. Duurzame resourcemanagers slaan alle wijzigingen op een duurzaam medium (bijvoorbeeld disk) op. Het is op die manier gewaarborgd dat alle wijzigingen bewaard worden, ook al crasht de machine.

Ik heb laten zien dat het nieuwe programmeermodel van *System.Transactions* het ons erg makkelijk maakt om transactionele resourcemanagers (van eventueel verschillende typen) te gebruiken om transactionele applicaties te bouwen. Maar wat nu als we een bepaalde resource transactioneel willen gebruiken, maar er bestaat geen transactionele resourcemanager voor deze resource? Daarvoor biedt *System.Transactions* ook een oplossing. Het is namelijk mogelijk om zelf een resourcemanager te bouwen die kan participeren in een transactiescope. Om dit principe te demonstreren zal ik laten zien hoe men zelf een transactionele resourcemanager kan bouwen.

To commit or not to commit, that's the question

Voordat ik een zelfgemaakte resourcemanager laat zien, eerst het volgende. Afhankelijk van welke resourcemanagers participeren in een transactie, wordt deze lokaal beheerd door de zogenaamde Lightweight Transaction Manager (LTM) van de *System.Transactions* infrastuctuur of gedistribueerd door de MSDTC (ik kom hier nog op terug in de paragraaf over Transaction Management Escalation). Als een transactie door de LTM wordt beheerd, en de resourcemanagers die participeren in de transactie ondersteunen het single-phase-commit (SPC) mechanisme, zal dit worden toegepast. Dit mechanisme bestaat slechts uit 1 fase, namelijk de 'Commit' fase. In deze fase voeren de verschillende resourcemanagers de wijzigingen direct door. Dit in tegenstelling tot het zogenaamde two-phase-commit (2PC) mechanisme. Bij dit mechanisme bestaat het doorvoeren van een transactie uit een tweetal fasen die door een transactionele resourcemanager moeten kunnen worden afgehandeld, te weten: de 'Prepare' fase en de 'Commit' fase. Wanneer men zelf een resourcemanager bouwt, moet deze het 2PC mechanisme ondersteunen en kan additioneel het SPC worden ondersteund. In de onderstaande beschrijving ga ik uit van het 2PC mechanisme.

Als binnen een transactiescope alle acties zijn uitgevoerd en de *Complete* methode wordt aangeroepen, zal de transactiemanager elke resourcemanager die participeert in de transactiescope vragen het doorvoeren van de gegevens voor te bereiden. Dit is de 'Prepare' fase van het 2PC mechanisme. De resourcemanagers bereiden in deze fase het doorvoeren van de wijzigingen voor. Een durable resourcemanager zal in deze fase voldoende informatie opslaan op disk (in een log) om de wijzigingen later door te kunnen voeren. Daarna meldt de resourcemanager of het voorbereiden is gelukt. Dit wordt ook wel het 'stemmen' over de transactie genoemd, waarbij elke resourcemanager een veto stem krijgt in het bepalen van de uitkomst van de transactie. Mocht er tijdens de 'Prepare' fase van de gegevens een fout optreden, dan levert de desbetreffende resourcemanager als resultaat 'niet gelukt' op. Dit is analoog aan het uitspreken van een veto. Als alle resourcemanagers gereed zijn met de 'Prepare' fase, breekt de 'Commit' fase aan. De transactiemanager beoordeelt het resultaat van de stemmen uit de 'Prepare' fase. Als minimaal één resourcemanager tegen heeft gestemd, zal de transactiemanager elke resourcemanager vragen de wijzigingen terug te draaien. Dit is een zogenaamde rollback. Is dit niet het geval, dan zal aan elke resourcemanager gevraagd worden de gegevens door te voeren. Dit is een zogenaamde commit. Na de commit of rollback zal elke resourcemanager zijn log opruimen.

Nu kan het voorkomen dat een durable resourcemanager wel de ‘Prepare’ fase uitvoert maar - om wat voor reden dan ook - niet meer te bereiken is om de ‘Commit’ fase uit te voeren. De beslissing om de commit of rollback uit te voeren is door de transactiemanager echter reeds genomen en kan niet worden teruggedraaid. De overige resourcemanagers zullen dan ook gewoon de commit of rollback uitvoeren en hun log opruimen. De gecrashte resourcemanager kan - nadat hij weer is opgekomen - aan de hand van zijn log bepalen of er nog openstaande transacties zijn. Voor elke openstaande transactie kan hij zich opnieuw aanmelden (*ReEnlist*) bij de transactiemanager. De transactiemanager zal dan nogmaals de uitkomst van de transactie melden (middels *Commit* of *Rollback*). Met behulp van de gegevens die tijdens de ‘Prepare’ fase zijn opgeslagen kan de resourcemanager dan de openstaande wijzigingen voor de transactie alsnog doorvoeren of terugdraaien en zijn log opruimen.

Om met onze eigen resourcemanager te kunnen participeren in dit 2PC mechanisme, levert de *System.Transactions* namespace ons de *IEnlistmentNotification* interface. Deze interface moet geïmplementeerd worden door onze resourcemanager en bevat methoden die corresponderen met de verschillende 2PC fasen: *Prepare*, *Commit*, *Rollback* en *InDoubt*. De eerste drie hebben we reeds gezien. *InDoubt* echter nog niet. Deze methode wordt alleen aangeroepen op volatile resourcemanagers wanneer binnen een transactie een durable resourcemanager uitvalt gedurende de de ‘Commit’ fase van een SPC. De implementatie van de *InDoubt* methode bestaat meestal uit het opruimen van geclaimde resources. Er kan na een *InDoubt* geen *Commit* of *Rollback* meer gedaan worden. Zodra een resourcemanager die *IEnlistmentNotification* implementeert, meedraait in een transactie, worden hierop de methoden corresponderend met de actieve fase van het 2PC mechanisme aangeroepen. De resourcemanager is dan zelf verantwoordelijk voor het leveren van de juiste implementatie om de ACID eigenschappen van de transactie te waarborgen. De implementatie van de resourcemanager en de manier waarop deze wordt aangemeld bij de lopende transactie bepalen samen of een resourcemanager vluchtig of duurzaam is. Beide varianten worden alleen geacht de *IEnlistmentNotification* interface te implementeren. Wanneer men naast het 2PC ook het SPC mechanisme wil ondersteunen, moet de *ISinglePhaseNotification* interface worden geïmplementeerd. Deze interface leidt af van de *IEnlistmentNotification* interface en heeft een additionele methode genaamd *SinglePhaseCommit*. De resourcemanager moet zelf de juiste implementatie leveren om zowel een SPC als een 2PC uit te kunnen voeren.

TxCustomer

Zoals gezegd gaan we zelf een transactionele resourcemanager bouwen. Deze zal als volatile resourcemanager worden geïmplementeerd, waarbij we dus wijzigingen die binnen een transactie worden gedaan alleen opslaan in het geheugen. Een resourcemanager moet ook voor isolation zorgen (afhankelijk van het isolation-level dat gewenst is). Omwille van de duidelijkheid laat ik dit in het voorbeeld buiten beschouwing.

De resourcemanager in het voorbeeld maakt het mogelijk een business-object dat een klant representeert mee te laten draaien in een transactie. Alle wijzigingen die vervolgens binnen de transactie worden gedaan, worden in hun geheel doorgevoerd of teruggedraaid. Allereerst maken we een *Customer* klasse die een klant representeert. Voor elke klant willen we een naam en een adres vastleggen. We voegen daarom een *Name* property en een *Address* property toe aan de klasse. De code voor deze klasse is te vinden in listing 3. Tot zover niets nieuws onder de zon. Daarna leiden we een klasse *TxCustomer* af van deze klasse. De code van deze transactionele variant is te vinden in listing 4. We implementeren *TxCustomer* zodanig dat deze zich automatisch als volatile resourcemanager registreert bij een eventuele lopende transactie (dit gebeurt in de *Enlist* methode). Daarnaast zorgen we ervoor dat wijzigingen die worden gedaan op de properties van deze klasse eventueel kunnen worden teruggedraaid. Dit wordt gerealiseerd door het bijhouden van een private *Customer* instantie waarop de wijzigingen worden gedaan (als er een lopende transactie is). Als laatste implementeren we de *IEnlistmentNotification* interface op de *TxCustomer* klasse. De bijbehorende code is te vinden in listing 5. De twee belangrijkste methoden zijn *Commit* en *Rollback*. Als *Commit* wordt aangeroepen, worden de tijdelijke waarden van de private *Customer* instantie overgenomen als de actieve waarden van de *TxCustomer* instantie. Als

Rollback wordt aangeroepen, blijven de actieve waarden van de *TxCustomer* instantie onveranderd.

Om *TxCustomer* te testen, maken we een console applicatie. De code is te vinden in listing 6. Allereerst maken we een *TxCustomer* instantie aan en initialiseren de waarden van de properties. Deze waarden drukken we af. Daarna starten we een transactie, wijzigen binnen deze transactie de waarden van de properties en drukken de waarden van de properties opnieuw af alvorens de transactie door te voeren. Daarna drukken we nogmaals de waarden van de properties af ter controle. Het resultaat is te zien in figuur 4. De wijzigingen zijn na de transactie netjes doorgevoerd. Als we de code nogmaals uitvoeren maar de transactiescope niet afsluiten (door de aanroep van *txScope.Complete()* achterwege te laten), zal een rollback worden gedaan. Het resultaat is te zien in figuur 5. Zoals te zien is zijn de wijzigingen die zijn gemaakt binnen de transactie teruggedraaid zoals verwacht.

Transaction management escalation

De voorbeeldimplementatie van de resourcemanager in dit artikel is natuurlijk verre van compleet. Er zijn talloze zaken waarmee een resourcemanager rekening moet houden om goed bruikbaar te zijn. Een van die zaken is 'Transaction Management Escalation' (TME). Het loont om hier rekening mee te houden als men zelf serieuze resourcemanagers gaat bouwen.

De *System.Transactions* infrastructuur biedt ons de zogenaamde Lightweight Transaction Manager (LTM) die de verschillende resourcemanagers die in een bepaalde transactie participeren beheert. Zolang binnen een transactie maar maximaal 1 duurzame resourcemanager en eventueel meerdere vluchtige resourcemanagers worden gebruikt, kan de LTM deze transactie beheren. Nu komt het regelmatig voor dat in een transactie meerdere duurzame resourcemanagers moeten participeren. Daarnaast is het soms noodzakelijk dat een transactie moet worden doorgegeven tussen verschillende appdomains, processen of zelfs machines. TME is het mechanisme van de *System.Transactions* infrastructuur dat het beheer van een bepaalde transactie automatisch overdraagt aan de MSDTC (escaleert), als de bovengenoemde situaties zich voordoen. Het is natuurlijk erg praktisch dat dit allemaal automatisch door de infrastructuur wordt geregeld, maar dit is wel iets om in de gaten te houden. De overdracht van het beheer van een transactie naar de MSDTC brengt namelijk wel een noemenswaardige performance penalty met zich mee. Deze penalty is enigszins te beperken middels het zogenaamde 'Promotable single phase enlistment' (PSPE) mechanisme. Dit kan het escaleren van een transactie naar de MSDTC in bepaalde gevallen voorkomen. Een bespreking van dit onderwerp valt buiten de scope van dit artikel, maar is iets wat zeker de moeite waard is om te bestuderen als men zelf een resourcemanager gaat bouwen.

Conclusie

De *System.Transactions* namespace biedt ontwikkelaars een groot aantal mogelijkheden om het bouwen van transactionele applicaties te vergemakkelijken. Vooral het impliciete programmeermodel op basis van transactie scopes, is een grote verbetering ten opzichte van het gebruik van transacties in .Net 1.x. Ondanks dat ik in dit artikel slechts een deel van deze mogelijkheden heb beschreven, hoop ik dat ik mensen enthousiast heb kunnen maken over deze nieuwe functionaliteit. Ik zou iedereen die geïnteresseerd is in het onderwerp aanraden om de MSDN Library er eens op na te slaan. Hierin is de *System.Transactions* namespace uitgebreid gedocumenteerd.

Listings

```
using (TransactionScope scope = new TransactionScope())
{
  // database insert
  SqlConnection conn = new SqlConnection(
    "Data Source=diamondsrv01;" +
    "Initial Catalog=northwind;" +
    "User ID=edwinw;Password=pa$$word");
```

```
  SqlCommand cmd = new SqlCommand(
    "insert into Employees (Lastname) " +
    "values ('Janssen')", conn);

  conn.Open();
  cmd.ExecuteNonQuery();
  conn.Close();

  scope.Complete();
}
```

Listing 1: Gebruik van een transactiescope

```
using (TransactionScope scope = new TransactionScope())
{
  // database insert
  SqlConnection conn = new SqlConnection(
    "Data Source=diamondsrv01;" +
    "Initial Catalog=northwind;" +
    "User ID=edwinw;Password=pa$$word");

  SqlCommand cmd = new SqlCommand(
    "insert into Employees (Lastname) " +
    "values ('Janssen')", conn);

  conn.Open();
  cmd.ExecuteNonQuery();
  conn.Close();

  // stop een bericht in de queue
  MessageQueue q = new MessageQueue("FormatName:" +
    "Direct=OS:localhost\\private$\\TxTestQueue");

  q.Send("Testdata", "TestBericht",
    MessageQueueTransactionType.Automatic);

  scope.Complete();
}
```

Listing 2: Gebruik van verschillende resourcetypen binnen één transactiescope

```
public class Customer
{
  private string _name;
  private string _address;

  public Customer()
  {
  }

  public virtual string Name
  {
    get
    {
      return _name;
    }
    set
    {
      _name = value;
    }
  }

  public virtual string Address
  {
```

```
    get
    {
      return _address;
    }
    set
    {
      _address = value;
    }
  }
}
```

Listing 3: Customer klasse

```
public class TxCustomer : Customer
{
  private Transaction _transaction = null;
  private Customer _tempCustomer = new Customer();

  public TxCustomer()
  {
  }

  public override string Name
  {
    get
    {
      if (_transaction != null)
      {
        return _tempCustomer.Name;
      }
      return base.Name;
    }
    set
    {
      if (Enlist())
      {
        _tempCustomer.Name = value;
      }
      else
      {
        base.Name = value;
      }
    }
  }

  public override string Address
  {
    get
    {
      if (_transaction != null)
      {
        return _tempCustomer.Address;
      }
      return base.Address;
    }
    set
    {
      if (Enlist())
      {
        _tempCustomer.Address = value;
      }
      else
      {
```

```
        base.Address = value;
      }
    }
  }

  private bool Enlist()
  {
    if (Transaction.Current != null)
    {
      if (_transaction == null)
      {
        _transaction = Transaction.Current;
        _transaction.EnlistVolatile(this,
          EnlistmentOptions.None);
      }
      return true;
    }
    return false;
  }
}
```

Listing 4: TxCustomer klasse

```
public class TxCustomer : Customer,
  IEnlistmentNotification
{

  // ...
  // code uit listing 3 weggelaten
  // voor de duidelijkheid
  // ...

  #region IEnlistmentNotification Members

  public void Prepare( PreparingEnlistment
    preparingEnlistment )
  {
    Console.WriteLine("TxCustomer.Prepare ");

    preparingEnlistment.Prepared();
  }

  public void Commit( Enlistment enlistment )
  {
    Console.WriteLine("TxCustomer.Commit ");

    base.Name = _tempCustomer.Name;
    base.Address = _tempCustomer.Address;

    _transaction = null;
    enlistment.Done();
  }

  public void Rollback( Enlistment enlistment )
  {
    Console.WriteLine("TxCustomer.Rollback ");

    _transaction = null;
    enlistment.Done();
  }

  public void InDoubt( Enlistment enlistment )
  {
```

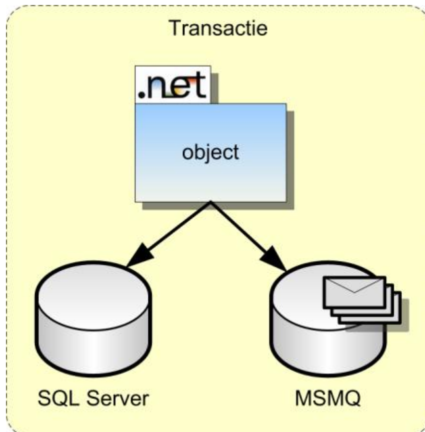
```
    Console.WriteLine("TxCustomer.InDoubt ");

    _transaction = null;
    enlistment.Done();
  }

  #endregion
}
```

Listing 5: TxCustomer klasse met IEnlistmentNotification implementatie

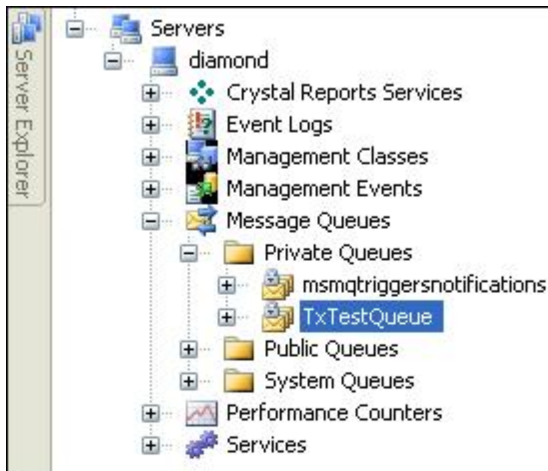
```
static void Main( string[] args )
{
  try
  {
    TxCustomer customer = new TxCustomer();
    customer.Name = "H. de Wit";
    customer.Address = "Leliestraat 12";

    Console.WriteLine(
      "Voor tx: Name = '{0}', Address = '{1}'.",
      customer.Name, customer.Address);

    using (TransactionScope txScope =
      new TransactionScope())
    {
      customer.Name = "J. Pietersen";
      customer.Address = "Beatrixlaan 133";

      Console.WriteLine(
        "Binnen tx: Name = '{0}', Address = '{1}'.",
        customer.Name, customer.Address);

      txScope.Complete();
    }

    Console.WriteLine(
      "Na tx: Name = '{0}', Address = '{1}'.",
      customer.Name, customer.Address);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.ToString());
  }

  Console.ReadKey(true);
}
```

Listing 6: Testapplicatie

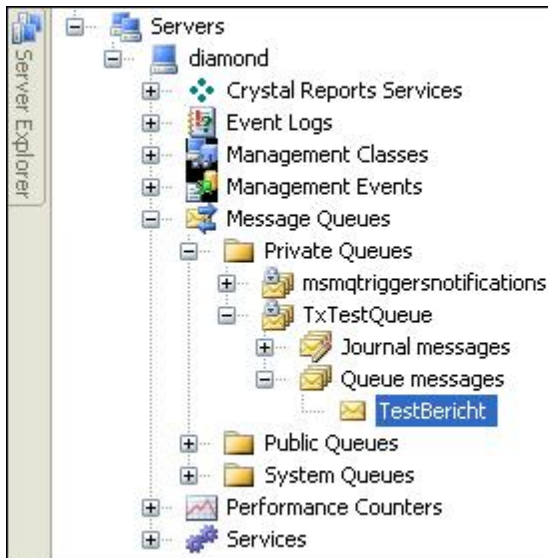
Figuren



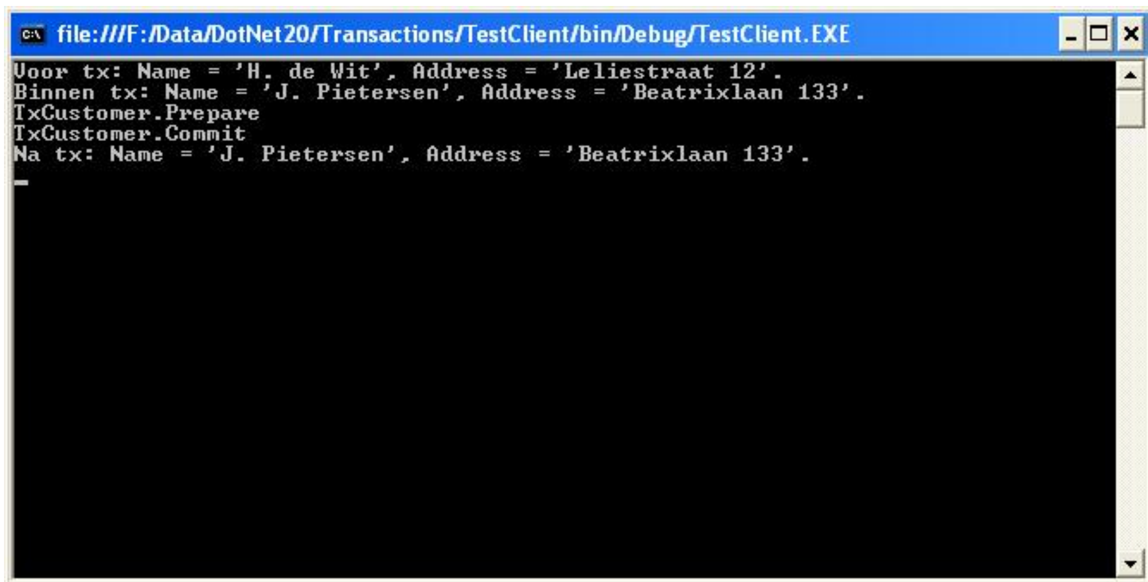
Figuur 1: Het gebruik van verschillende type resources binnen 1 transactie



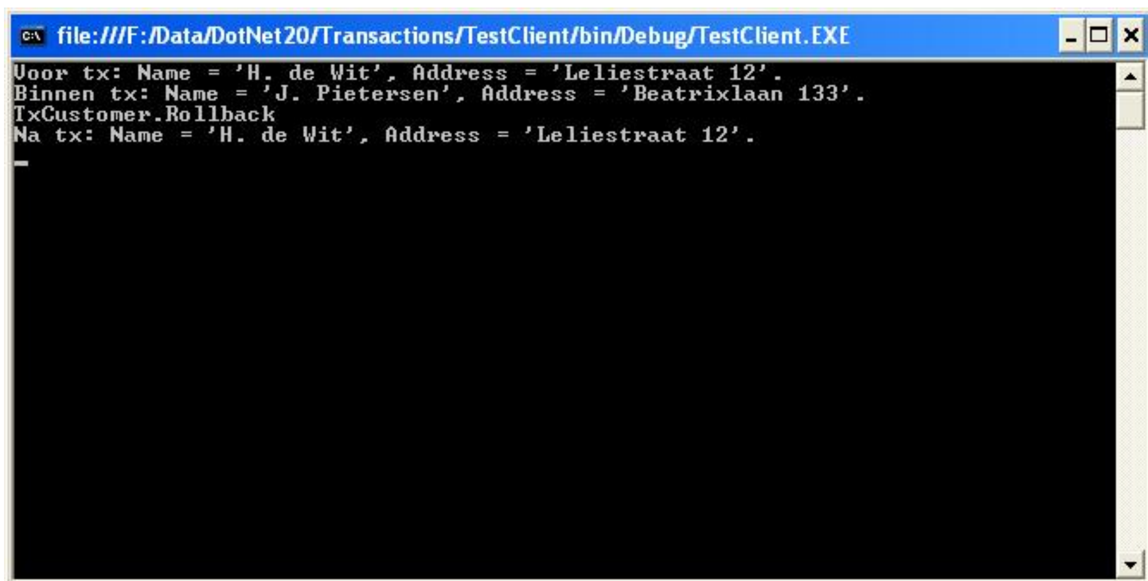
Figuur 2: Messagequeue aangemaakt



Figuur 3: Bericht in de MessageQueue



Figuur 4: Resultaat van de testapplicatie met commit



Figuur 5: Resultaat van de testapplicatie met rollback