Transactions

Estimated time for completion: **30 Minutes**

Overview:

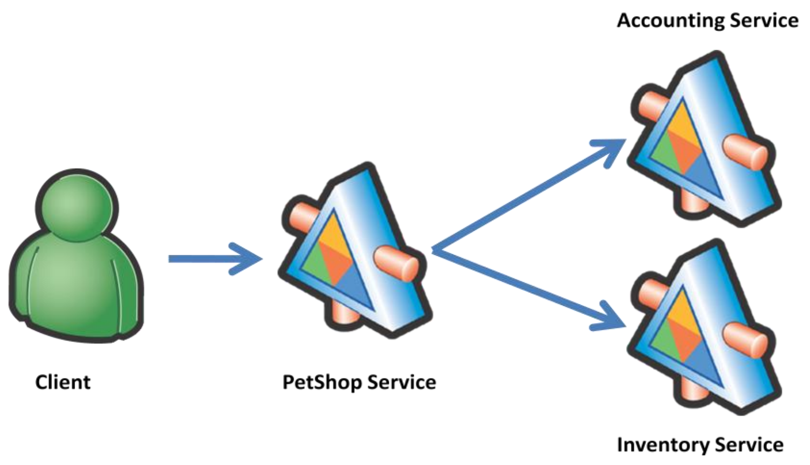
In service oriented architecture, services are building blocks of functionality. Sometimes it is necessary to span a transaction over multiple services and operations. The WS-Atomic Transactions protocol allows doing that. In this lab you will modify an existing application to use transactions for back-end processing.

Goals:

- Configure WCF to use transactions
- Add distributed transactions to the application

Part 1-1: The PetShop starter application

Before you start with the lab, take a second to look at the starter solution which you can find here. This time the infrastructure is a little bit more complex than in the previous labs.



The client is a WinForms application that allows buying animals. You can see which animals and how much of them are in stock. You can also see the balance of your account with the pet shop. When you order animals, the PetShop service will first talk to the accounting service to debit your account and after that update your inventory using the inventory service.

These two operations should happen in an atomic fashion meaning either both operations succeed or both fail. The current implementation does not take this into account.

Try to buy four elephants. The charging of your account will succeed, but updating the inventory will fail and the two pieces of data will be out of sync.

Part 1-2: Adding transaction support

In this part you will add transaction support to the PetShop, accounting and inventory service.

1. Configure the transport to flow transactions. This has to be done in the inventory, accounting and petshop service. Create a binding configuration for the *wsHttpBinding* and set the *transactionFlow* attribute to *true*.

```
<bindings>
  <wsHttpBinding>
    <binding name="Tx"
             transactionFlow="true" />
  </wsHttpBinding>
</bindings>
```

2. Add this binding configuration to all three configuration files and link to it from the corresponding *service* or *client* elements.

3. Annotate the service contracts with transaction attributes. The *ChargeAccount* operation in the accounting service and the *UpdateInventory* operation in the inventory service need to be called in a transaction. Express this fact by adding a *[TransactionFlow(TransactionFlowOption.Mandatory)]* to the service contract for these operations.

4. An operation behavior controls if the ambient transaction is available in an operation. Add *[OperationBehavior(TransactionScopeRequired = true)]* to the implementations of operations that you have configured to require transaction in the previous step.

5. Inspect the source for *AccountStore* and *InventoryStore*. They use transactional data structures that know how to enlist in an ambient transaction. You can download these useful classes here.

6. The last step is to initiate the transaction before calling into the two backend services. Wrap the calling code in the *PlaceOrder* operation in the PetShop service in a *TransactionScope* and call *Complete* on that scope at the end.

```
public void PlaceOrder(OrderMessage request)
{
  using (TransactionScope scope = new TransactionScope())
  using (ChannelWrapper<IPetShopInventoryService> inventory = new
    ChannelWrapper<IPetShopInventoryService>("Inventory"))
  using (ChannelWrapper<IPetShopAccountingService> accounting = new
    ChannelWrapper<IPetShopAccountingService>("Accounting"))
  {
    // update account
    // update inventory

    scope.Complete();
  }
}
```

7. Now try again to buy four elephants. This time the first call to the service take a significant amount longer. That's the overhead for setting up the distributed transaction. This time the inventory and the account will be in sync and both backend operations were executed as a single atomic operation.

Solutions:

A solution for this part is located at here.