Contents

A **Remote Server Method** is a server-side method that is defined and deployed in an assembly on the server but which can be called from the client application. This type of method can return any serializable object and can be passed any serializable parameters.

Define a Remote Server Method

It's easy to define an RSM method, just follow a few simple rules:

1) Define a **static** method which conforms to the *ServerMethodDelegate* signature:

```
[AllowRpc]
public delegate object ServerMethodDelegate(IPrincipal principal, EntityManager serverEntityManager,
                                            params object[] args)
```

```
<AllowRpc> _
Public Delegate Function ServerMethodDelegate(ByVal principal As IPrincipal, ByVal serverEntityManager
                                              As EntityManager, ByVal ParamArray args() As Object) As Object
```

In other words, a method which accepts an *IPrincipal*, an *EntityManager* and a parameter array. We'll explain these arguments further below.

2) Apply the *[AllowRpc]* attribute to the method.

3) All types passed as arguments and as the return value must be simple types or primitives or must meet the requirements for known types.

Method arguments

Both the *IPrincipal* and *EntityManager* are provided to the method by DevForce, they are not passed from the client.

IPrincipal - The *IPrincipal* passed to the method is the *IPrincipal* associated with the user of the client application, and is the same *IPrincipal* returned by the *EntityManager.Login* call.

EntityManager - The *EntityManager* passed into the method is an empty server-side *EntityManager* constructed by DevForce specifically for the method call. This *EntityManager* does not need to be logged in, and may be used to issue queries and saves.

params Object[] - To pass any arguments to be used inside the server method.

If you want to pass or return entities from this method, you should serialize them into an *EntityCacheState*. See Take a snapshot to learn more.

Call a Remote Server Method from the client

You can invoke a server method from the client synchronously or asynchronously:

InvokeServerMethod
InvokeServerMethodAsync

Both synchronous and asynchronous flavors have several overloads which allow you to pass either a strongly-typed *ServerMethodDelegate* pointing to the server method, or otherwise the type and method name to be called. When using the weakly-typed overload, be sure that the type name is fully qualified, for example “MyNamespace.MyType, MyAssembly”. If using the strongly-typed overload, make sure that the delegate is available on the client in an assembly with the same name as the server. See the example below for how to use both options.

Specify a communication retry policy

If your server method is idempotent you may want to provide a custom retry policy, or turn off retry for the call altogether. To do so, use the method override which uses an instance of *InvokeServerMethodArgs*.

For example:

```
// Call an RPC method with NoRetry
var args = new InvokeServerMethodArgs {
  FullTypeName = "DomainModel.ServerMethods, DomainModel",
  MethodName = "GetNumberOfOrders",
  ClientArgs = new object[] { 10, new DateTime(1995, 1,1), new DateTime(1999, 1, 1)},
  CommunicationRetryPolicy = new NoRetry()
};
var num = (int)entityManager.InvokeServerMethod(args);
```

Secure a Remote Server Method

You should secure your server methods as you do your data and other service calls. See the authorization topic for details.

Example

1) Define the server method.

```
namespace DomainModel {
 public class ServerMethods {
    [AllowRpc]
   public static int GetNumberOfOrders(IPrincipal principal,
      EntityManager entityManager, params Object[] args) {
      // Gather args
      DateTime startDate = (DateTime) args[0];
       DateTime endDate = (DateTime) args[1];

      // Query and get the count
      int count = entityManager.GetQuery<OrderSummary>()
         .Where(os => os.OrderDate > startDate && os.OrderDate < endDate)
         .Count();
      return count;
    }
  }
}
```

```
Namespace DomainModel
 Public Class ServerMethods
    <AllowRpc> _
   Public Shared Function GetNumberOfOrders(principal As IPrincipal,_
      entityManager As EntityManager, ParamArray args As Object()) As Integer
    ' Gather args
    Dim startDate As DateTime = DirectCast(args(0), DateTime)
    Dim endDate As DateTime = DirectCast(args(1), DateTime)
    ' Query and get the count
    Dim count As Integer = entityManager.GetQuery(Of OrderSummary)()_
       .Where(Function(os) os.OrderDate > startDate AndAlso os.OrderDate < endDate)_
       .Count()
    Return count
   End Function
 End Class
End Namespace
```

2) On the client, call the server method passing its *fullTypeName* and *methodName*:

```
public int GetOrderCount(DateTime startDate, DateTime endDate) {
 string typeName = "DomainModel.ServerMethods, DomainModel";
 string methodName = "GetNumberOfOrders";
 return (int)_entityManager.InvokeServerMethod(typeName, methodName, startDate, endDate);
}
```

```
Public Function GetOrderCount(startDate As DateTime, endDate As DateTime) As Integer
 Dim typeName As String = "DomainModel.ServerMethods, DomainModel"
 Dim methodName As String = "GetNumberOfOrders"
 Return CInt(_entityManager.InvokeServerMethod(typeName, methodName, startDate, endDate))
End Function
```

3) ... or call the server method using a *ServerMethodDelegate*:

```
public int GetOrderCount(DateTime startDate, DateTime endDate) {
```

```
 var serverMethod = new ServerMethodDelegate(ServerMethods.GetNumberOfOrders);
return (int)_entityManager.InvokeServerMethod(serverMethod, startDate, endDate);
}
```

```
Public Function GetOrderCount(ByVal startDate As Date, ByVal endDate As Date) As Integer
 Dim serverMethod = New ServerMethodDelegate(ServerMethods.GetNumberOfOrders)
 Return CInt(_entityManager.InvokeServerMethod(serverMethod, startDate, endDate))
End Function
```

Troubleshooting

So you've followed the setup information above, yet it's just not working. Maybe it looks like the method call is hanging, or maybe an indecipherable error message is returned.

1) Check the debug log for warning and informational messages. DevForce will log an informational message about every known type found, and every remote server method. Logging of the remote server methods is done lazily, as the methods are called, but if DevForce thinks the return type will be a problem it will write a warning to the log.

2) Add a try/catch around your call. We try to intercept the underlying error (which is almost always a *CommunicationException* or *SerializationException*) and provide some more helpful information, but it's not always easy. Check the *InnerException* too. Exceptions with the *InvokeServerMethod* are almost always due to a problem with either the input or output types, so walk through the requirements above to ensure you haven't missed anything.

3) What if you're passing or returning a complex type, over which you have no control? Wrap the type in another simple class or struct and make sure that simple type meets all the serialization/known type requirements.