Contents

You've landed here because you're trying to use some simple feature of DevForce - say a remote server method or POCO type - and you've gotten a strange error message about "known types". Here we'll try to dispel some of the misconceptions and explain a few of the mysteries of why **known types** are so important in your DevForce application.

What is a known type?

In an n-tier application, such as a DevForce Silverlight application or any application using a remote EntityServer, data must be moved between tiers: it must be serialized for transmission and deserialized by the recipient. In DevForce, this serialization and deserialization is done using the .NET DataContractSerializer (DCS). The serializer must be able to transmit queries, entities, POCOs, RSM arguments and return values - anything you send to or receive from the server.

The DCS uses contracts to tell it how to process the data it receives; whenever the contract is imprecise, for example when a property is typed as Object, or as an interface or abstract type, the DCS needs to be told what types are known to be in the object graph, hence **known types**. In DevForce it's not only properties which might be vague, but method arguments and return values too may not be strongly typed by the underlying DevForce contracts, and must instead be made known at run time.

It should be obvious that a known type must also be serializable. A type is generally serializable if it has a parameterless public constructor, or it's marked up with DataContract and DataMember attributes. Here's more information from Microsoft on what can be serialized with the DCS.

When should I care?

Login - You may have noticed two things about the *Login* method: it takes an *ILoginCredential*, and from that credential produces an *IPrincipal*. Two interfaces, whose concrete implementations must be known at run time. Therefore, any custom *ILoginCredential* or *IPrincipal* (or *IIdentity*) must be made known to DevForce as a known type. Fortunately, DevForce will specifically look for custom implementations of these interfaces and add them to a list of known types it builds for the serializer so you do not have to do anything special, but it's still worth remembering that these are known types.

Remote Server Methods - If you've created any remote server methods (sometimes called either RSM or RPC), you've also likely passed data both as arguments to the methods and as return values. Remember the signature for the RSM:

```
public delegate Object ServerMethodDelegate(IPrincipal principal, EntityManager serverEntityManager, params Object[] args);
```

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

A method which takes zero or more user-specified objects, and returns an object (which might be a list or graph of objects). Since the signature is not strongly typed, everything passed into or out of the method must be designated as a known type. In this case, DevForce won't do this discovery for you - you must tell DevForce, or the serializer, the known types involved.

POCO - POCO types which will be passed between tiers must also be identified as known types. If your POCO type has a Key attribute DevForce will automatically add your POCO type as a known type, but it's good to remember that POCO types must be known types.

Projection into type - When you project query results into a custom type, that custom type must be designated as a known type.

Query Contains clause - In LINQ a *Contains* clause is used in place of the SQL *IN* operator. The list on which the *Contains* is defined must be a known type if the query will be sent to the server. DevForce automatically defines a known type for all List<T> where T is a numeric type, a string, a DateTime or a GUID. If your list contains some other type the list for this type must be designated as a known type.

Note that primitive types are already known to the DCS and if you are using these, for example as arguments to or a return value from an RSM, you do not need to designate the type as a known type. Here's a list from Microsoft on the primitives it automatically recognizes.

What does DevForce provide automatically?

At start up, and upon recomposition when an on-demand XAP is downloaded, DevForce will automatically build a list of known types to be provided to the serializer. This list is actually provided to any DCS which might be used: the serializer used to transmit data between tiers, a serializer used when creating or restoring an EntityCacheState, and serializers used for specialized cloning.

This list contains:

- All entity and complex types found in any entity models, and List<T> and *RelatedEntityList<T>* for each.
- List<T> where T is a numeric, string, DateTime or GUID.
- All DevForce types expected to be transmitted between tiers.
- DateTimeOffset and StringComparison.
- Any discovered *ILoginCredential*, *IDataSourceKeyResolver*, *IIdGenerator*, *IPrincipal*, and *IIdentity* implementations.
- Types discovered via *IKnownType*, *DiscoverableTypeMode.KnownType* and *IKnownTypeProvider*.

You may notice that any types marked with the .NET KnownType attribute are not included in this list. The serializer has two ways of obtaining known types: one is the list provided to it upon construction, and second is via the *KnownType* attributes it finds within the object graph being serialized/deserialized. The list DevForce builds is the list passed to the serializer constructor; but your object graph can include *KnownType* attributes to directly tell the serializer what to expect within the object graph. Note that the *KnownType* attribute is not used to indicate that the type itself is a known type, but instead to indicate other types are known types. This is a subtle but important distinction: you can tell DevForce about known types (via DevForce interfaces) and DevForce will in turn tell the serializer; or you can directly tell the serializer (via the .NET *KnownType* attribute). The end result is the same: the serializer will use the known type information whenever the contract definition is imprecise.

How to indicate a known type?

Have the type implement the *IdeaBlade.EntityModel.IKnownType* marker interface

When you have control over the type (i.e., it's not a .NET type or one defined in a third-party assembly), using the marker interface is often the simplest implementation.

```
public class OrderRequestInfo  : IKnownType {
public int Top { get; set; }
public DateTime StartDate { get; set; }
public DateTime EndDate { get; set; }
}
```

```
Public Class OrderRequestInfo _
 Implements IKnownType
Public Property Top() As Integer
Public Property StartDate() As Date
Public Property EndDate() As Date
End Class
```

Decorate the type with the *IdeaBlade.EntityModel.DiscoverableType* attribute

This attribute is functionally equivalent to the *IKnownType* marker interface.

```
[DiscoverableType(DiscoverableTypeMode.KnownType)]
public class OrderRequestInfo  {
public int Top { get; set; }
public DateTime StartDate { get; set; }
public DateTime EndDate { get; set; }
}
```

```
<DiscoverableType(DiscoverableTypeMode.KnownType)> _
Public Class OrderRequestInfo
Public Property Top() As Integer
Public Property StartDate() As Date
Public Property EndDate() As Date
End Class
```

Return the type from an *IdeaBlade.EntityModel.IKnownTypeProvider*

The *IKnownTypeProvider* allows you to specify any type - your own, .NET or third party types - so is useful either when you don't want to modify type declarations or cannot. Any number of *IKnownTypeProviders* may be defined.

```
public class KnownTypeProvider : IKnownTypeProvider {
```

```
public IEnumerable<Type> AddKnownTypes() {
    var list = new Type[] { typeof(OrderRequestInfo),
    typeof(OrderResponseInfo),
    typeof(List<DateTimeOffset>)
    };
  return list;
  }
}
```

```
Public Class KnownTypeProvider
 Implements IKnownTypeProvider
 Public Function AddKnownTypes() As IEnumerable(Of Type)
  Dim list = New Type() {GetType(OrderRequestInfo), _
    GetType(OrderResponseInfo), GetType(List(Of DateTimeOffset))}
  Return list
 End Function
End Class
```

Add the *System.Runtime.SerializationKnownTypeAttribute* to another type (such as an entity)

Here you're marking up your object graph to indicate to the serializer what other types to expect. DevForce is unaware of types marked with the *KnownType* attribute, so this cannot be used when defining POCO types.

```
[KnownType(typeof(OrderRequestInfo))]
public partial class OrderSummary { ... }
```

```
<KnownType(GetType(OrderRequestInfo))> _
Partial Public Class OrderSummary
...
End Class
```

What are the current KnownTypes?

DevForce can tell you what types are "KnownTypes" from its perspective:

```
IEnumerable<Type> knownTypes = IdeaBlade.EntityModel.KnownTypeHelper.GetServiceKnownTypes(null);
```

```
IEnumerable(Of Type) knownTypes = IdeaBlade.EntityModel.KnownTypeHelper.GetServiceKnownTypes(Nothing)
```

The *IdeaBlade.EntityModel.**KnownTypeHelper*** helper class sports other methods that reveal useful information about the "KnownTypes" known to DevForce at the current moment.

Troubleshooting

So you've received the dreaded formatter exception, something like "The formatter threw an exception while trying to deserialize the message ..." and it goes on for 3 or 4 lines. Somewhere in there it mentions a type (or contract name) and says to add the type to the list of known types. What? How? Didn't I already do that?

This message is coming straight from the serializer, and does tell you exactly what it found to be wrong, albeit in somewhat confusing language.

1. Make sure that the type is defined on both client and server. This is easy to overlook, but should be intuitively obvious. Both the client and server should be aware of any types to be passed between them. With Silverlight applications, DevForce ensures this of entities in the model by including a link to the generated code in the Silverlight application. You can do the same with your own types: define the type in a file in a Desktop/Server project and create a link to the file in a Silverlight project. DevForce will perform type mapping between tiers, so as long as the namespace and type name are the same, you may place the type definition in any assembly.

2. Make sure the type is serializable. If your type has a non-default constructor (in other words a constructor which accepts arguments) then it is not serializable. You can easily remedy this by either adding a default constructor or marking up the class with *DataContract* and *DataMember* attributes so that you can specifically control which members are serialized.

If a type which cannot be serialized is included as a known type the EntityServer service will not start. You'll see a message in the server's debug log which states the service "cannot be activated due to an exception during compilation." The message will also include the type at fault.

3. Make sure the type is somehow marked as a known type, given the information above on how to indicate a known type.

Note that a type, T, and List<T> are not the same thing. If you return a List<T> from an RSM, then the known type is List<T>, not T.

Also note that to the serializer, List<T> and T[] are different things, but they may unfortunately have the same contract. Since DevForce automatically adds List<T> for most primitive types to the known type list, you will receive an error if you add T[] for a primitive type as a known type.

This also means that in queries with a **contains** clause you should use a List<T> and not T[].