**Contents**



The *EntityServer* is composed of several WCF services. DevForce usually handles the WCF configuration for you on both client and server, relieving you of the need to understand the complexities of WCF. But in some circumstances you may want, or need, to take control over this configuration. Below we'll describe the configuration details and how you can customize them for **advanced scenarios**.

# N-tier client

By default, DevForce will use the <objectServer> information in your config file to configure communications from the client to the *EntityServer*. In Silverlight applications you won't usually have an app.config file, so the <*objectServer*> settings are defaulted based on the URL from which the XAP was downloaded.

You can also directly set *objectServer* properties on the *IdeaBladeConfig.Instance* at startup if you don't wish to use a config file.

## Default configuration

Before diving into customization, let's get a better understanding of the default configuration DevForce uses. The rules are fairly simple:

1. Communications will be established for a specific "service name". This is usually either "EntityService" or "EntityServer". When a data source extension or custom composition context is used then the EntityServer name will also contain that information, for example "EntityServer_dsext+ccname".
2. If a WCF <system.serviceModel> section is found in the config file (ServiceReferences.ClientConfig in Silverlight) with an endpoint for the particular service name, then a ChannelFactory is built from this configuration information.
3. If the <*system.serviceModel*> information was not present or invalid then **programmatic** configuration is performed.
    1. The **Address** is built from the information in the <*objectServer*> element, formatting a URI from the *RemoteBaseUrl*, *ServerPort* and the specific service name. In Silverlight, "/sl" is appended to the address if not already present, since the default *EntityServer* endpoints all use this postfix to indicate Silverlight-specific endpoints.
    2. The **Binding** is built, based on the protocol scheme in the *RemoteBaseUrl*. A CustomBinding is built from the following:
        1. The *GZipMessageEncodingBindingElement* to provide compressed binary messages.
        2. A transport binding element appropriate to the protocol:
           *http* - HttpTransportBindingElement
           *https* - HttpsTransportBindingElement
           *net.tcp* - TcpTransportBindingElement. In Silverlight this binding requires the System.ServiceModel.NetTcp assembly, so DevForce does not provide programmatic configuration by default; you can still use tcp via the *serviceModel*.
           *net.pipe* - NamedPipeTransportBindingElement. Not supported in Silverlight.
        3. Whatever the transport, the *MaxReceivedMessageSize* is set to int.MaxValue, and except in Silverlight, the ReaderQuotas are set to int.MaxValue for *MaxArrayLength*, *MaxDepth* and *MaxStringContentLength*. The *ReaderQuotas* are not supported by Silverlight.
    3. The **Contract** is one of the DevForce service contracts, indicating either the *EntityService* or *EntityServer*.
    4. The ChannelFactory is built from this information.
4. Except in Silverlight, a DataContractSerializerOperationBehavior is added to all contract operations to specify the serializer to use (DCS or NDCS).
5. The *ServiceProxyEvents* *OnEndpointCreated* and *OnFactoryCreated* methods are called and the proxy is opened. See below for more information on the *ServiceProxyEvents*.

## Customizing configuration

We saw above that **either** the <*system.serviceModel*> determines the configuration **or** it's built programmatically. There are two ways to to customize this.

1. Use the WCF <*system.serviceModel*> section to **configure** client communications. See the samples on working with the *serviceModel* in client applications.
2. Use a custom *ServiceProxyEvents* to **modify** existing configuration. As we saw above, methods on this class are called after the *ChannelFactory* is built, regardless of how it is built. See the samples for a detailed description and examples.

In some cases you may want to use both approaches at the same time. For example, a <*system.serviceModel*> to configure the communications elements needed, and then a *ServiceProxyEvents* to inject additional runtime-specific behaviors.

In Silverlight, configuration options via the ServiceReferences.ClientConfig are limited, so doing additional modifications, as permitted by Silverlight, through a *ServiceProxyEvents* can be helpful. See the sample on adding Gzip support for an example.

## *EntityServer*

DevForce will create a least two WCF services, one *EntityService* and one or more *EntityServers*. The *EntityService* functions as a gateway to an *EntityServer*: a client application will handshake with it, and then all further communications it makes will be with an *EntityServer*. If you are using either a data source extension and/or custom composition context when you create an *EntityManager* in your client application, then it will communicate with an *EntityServer* service having the same characteristics and whose name reflects these settings.

### Default configuration

The same configuration rules are following for both *EntityService* and *EntityServer* services. Configuration is also independent of hosting type except where noted.

1. Determine the service name. This is usually either "EntityService" or "EntityServer". When a data source extension or custom composition context is used then the EntityServer name will also contain that information, for example "EntityServer_dsext+ccname".
2. Determine the base address(es) of the service.
    1. When hosted in IIS, the IIS application name and web site bindings determine the base addresses. For example, if both http and https bindings are enabled for a web site, then a base address is supplied by IIS for each.
    2. When hosted by either the ServiceService or ServerConsole:
        1. If <baseAddresses> are defined for the service in the config file then they will be used,
        2. Otherwise information in the <objectServer> element is used, formatting a URI from the *RemoteBaseUrl*, *ServerPort* and the specific service name.
3. If a WCF <system.serviceModel> section is found in the config file with a <*service*> element for the particular service name, then it will be used to configure endpoints and behaviors. For the *EntityServer*, a service element named "EntityServer" will be used if present and no element exists for the specific service name (e.g., "EntityServer_ABC"). This allows the single service configuration to serve as a template for all *EntityServer* services which might be used, while still allowing each *EntityServer* to use custom configuration when wanted.
4. Add service behaviors:
    1. A behavior is added for the DevForce error handler to create DevForce-specific message faults.
    2. Set AspNetCompatibilityRequirementsAttribute to *allowed*.
    3. Add a ServiceThrottlingBehavior to set the *MaxConcurrentCalls* and *MaxConcurrentSessions* to 1000 for http/https and 100 for tcp.
    4. Add a DataContractSerializerOperationBehavior to all contract operations to specify the serializer to use (DCS or NDCS).
5. If no endpoints were found for the service in the <*system.serviceModel*> then DevForce will add one or more endpoints for each base address. The *SupportedClientApplicationType* on the <*serverSettings*> determines which endpoints are added. This setting defaults to *UseLicense*, which means that endpoints will be added for whatever your license allows.
    1. The **Address** is the base address.
    2. The **Binding** is built based on the protocol scheme for the address. A CustomBinding is built from the following:
        1. The *GZipMessageEncodingBindingElement* to provide compressed binary messages.
        2. A transport binding element appropriate to the protocol:
           *http* - HttpTransportBindingElement
           *https* - HttpsTransportBindingElement
           *net.tcp* - TcpTransportBindingElement. This is not used for Silverlight endpoints.
           *net.pipe* - NamedPipeTransportBindingElement. This is not used for Silverlight endpoints.
        3. Whatever the transport, the *MaxReceivedMessageSize* is set to int.MaxValue, and the ReaderQuotas are set to int.MaxValue for *MaxArrayLength*, *MaxDepth* and *MaxStringContentLength*.
    3. A ServiceEndpoint is added using the base address, the binding created and the contract for the service. For Silverlight endpoints, a relative address of "/sl" is added.
    4. The *ServiceHostEvents* *OnEndpointCreated* is called to allow the endpoint to be customized.
6. For Silverlight endpoints, an endpoint behavior is added for the *SilverlightFaultBehavior* to allow faults to be sent to the Silverlight client. DevForce considers an endpoint a "Silverlight endpoint" if it contains the "/sl" relative address.

7. After all endpoints are processed *ServiceHostEvents OnServiceHostCreated* is called to allow customization of the service and all endpoints before the *ServiceHost* is opened.

After all these efforts, you'll see messages in the server's log indicating the service name and the addresses it's listening on. For example, the following log entries show that the *EntityService* is listening on four endpoints. Two base addresses were supplied by IIS, http://myhost/BadGolf and https://myhost/BadGolf becasue both http and https bindings are set for the web site; and two endpoints were created for each base address, an endpoint for Silverlight client applications and one for all other applications.

```
EntityService listening on http://[redacted]/BadGolf/EntityService.svc
EntityService listening on http://[redacted]/BadGolf/EntityService.svc/sl
EntityService listening on https://[redacted]/BadGolf/EntityService.svc
EntityService listening on https://[redacted]/BadGolf/EntityService.svc/sl
```

### Customizing configuration

We saw above that **either** the <*system.serviceModel*> determines the configuration **or** it's built programmatically. There are two ways to to customize this.

1. Use the WCF <*system.serviceModel*> section to **configure** the service and its endpoints. See samples on working with the *serviceModel* on the server.
2. Use a custom *ServiceHostEvents* to **modify** existing configuration. As we saw above, methods on this class are called after each programmatic endpoint is added, and again after the *ServiceHost* is built. See the samples for a detailed description and examples.

In some cases you may want to use both approaches at the same time. For example, a <*system.serviceModel*> to configure the communications elements needed, and then a *ServiceHostEvents* to inject additional runtime-specific behaviors.

As we saw above you can also control the endpoints created using the *SupportedClientApplicationType* setting in the <serverSettings>. For example, if your application will only have Silverlight clients yet you have an Enterprise license, you can set the appropriate *SupportedClientApplicationType* setting so that only Silverlight endpoints are created.

## Controlling compression

By default all communications between the n-tier client and *EntityServer* are compressed. You can control the level of compression - to prefer speed over compression factor or vice versa - to fine tune application communications.

To set the compression level, use the *CommunicationSettings* class. For example:

```
IdeaBlade.Core.Wcf.Extensions.CommunicationSettings.Default.CompressionLevel = Ionic.Zlib.CompressionLevel.BestSpeed;
IdeaBlade.Core.Wcf.Extensions.CommunicationSettings.Default.CompressionLevel = Ionic.Zlib.CompressionLevel.BestSpeed
```

The *CompressionLevel* enumeration has a range of possibilities from *BestCompression* to *BestSpeed*. It also supports turning off compression, although if you wish to turn off compression it's more efficient to change the bindings used.

On the *EntityServer* the *CommunicationSettings.Default.CompressionLevel* should be set in the global.asax when hosted in IIS; on the client you can set this in the application startup code. You can actually change the *CompressionLevel* at any time while your application is running and the new level will be used for all further communications, but generally you should set this early in your startup logic. The level can also differ between client and server.

## Additional resources

The ABCs of Endpoints