Coherence for .NET User Guide

Table Of Contents

1 Overview

Coherence for .NET allows .NET applications to access Coherence clustered services, including data, data events, and data processing from outside the Coherence cluster. Typical uses of Coherence for .NET include desktop and web applications that require access to Coherence caches.

Coherence for .NET consists of a lightweight .NET library that connects to a Coherence*Extend clustered service instance running within the Coherence cluster using a high performance TCP/IP-based communication layer. This library sends all client requests to the Coherence*Extend clustered service which, in turn, responds to client requests by delegating to an actual Coherence clustered service (for example, a Partitioned or Replicated cache service).

An `INamedCache` instance is retrieved via the `CacheFactory.GetCache()` API call. Once it is obtained, a client accesses the `INamedCache` in the same way as it would if it were part of the Coherence cluster. The fact that `INamedCache` operations are being sent to a remote cluster node (over TCP/IP) is completely transparent to the client application.

2 Getting Started

Configuring and using Coherence for .NET requires five basic steps:

1. **Configure Coherence*Extend (Section 3)** on both the client and on one or more servers within the cluster.
2. **Configure a POF context (Section 4)** on the client and on all of the servers within a cluster that are running Coherence*Extend service.
3. Implement the .NET client application **using the Coherence for .NET API (Section 5)**.
4. Make sure the **Coherence cluster is up and running (Section 6)**.
5. Launch the .NET client application.

The following sections describe each of these steps in detail.

3 Configuring Coherence*Extend

To configure Coherence*Extend, you need to add the appropriate configuration elements to both the cluster and client-side cache configuration descriptors. The cluster-side cache configuration elements instruct a Coherence `DefaultCacheServer` to start a Coherence*Extend clustered service that will listen for incoming TCP/IP requests from Coherence*Extend clients (including Coherence for .NET clients). The client-side cache configuration elements are used by the client library to determine the IP address and port of one or more servers in the cluster that are running the Coherence*Extend clustered service so that it can connect to the cluster. It also contains various connection-related parameters, such as connection and request timeouts.

Configuring Coherence*Extend in the Cluster

In order for a Coherence*Extend client to connect to a Coherence cluster, one or more `DefaultCacheServer` JVMs within the cluster must run a TCP/IP Coherence*Extend clustered service. To configure a `DefaultCacheServer` to run this service, a `proxy-scheme` element with a child `tcp-acceptor` element must be added to the cache configuration descriptor used by the `DefaultCacheServer`. For example:

```
<?xml version="1.0"?>
<!DOCTYPE cache-config SYSTEM "cache-config.dtd">

<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>dist-*</cache-name>
      <scheme-name>dist-default</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>

  <caching-schemes>
    <distributed-scheme>
      <scheme-name>dist-default</scheme-name>
      <lease-granularity>member</lease-granularity>
      <backing-map-scheme>
        <local-scheme/>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>

    <proxy-scheme>
      <service-name>ExtendTcpProxyService</service-name>
      <thread-count>5</thread-count>
      <acceptor-config>
        <tcp-acceptor>
          <local-address>
            <address>localhost</address>
            <port>9099</port>
          </local-address>
        </tcp-acceptor>
      </acceptor-config>
      <autostart>true</autostart>
    </proxy-scheme>
  </caching-schemes>
</cache-config>
```

This cache configuration descriptor defines two clustered services, one that allows remote Coherence*Extend clients to connect to the Coherence cluster over TCP/IP and a standard Partitioned cache service. Since this descriptor is used by a `DefaultCacheServer`, it is important that the `autostart` configuration element for each service is set to true so that clustered services are automatically restarted upon termination. The `proxy-scheme` element has a `tcp-acceptor` child element which includes all TCP/IP-specific information needed to accept client connection requests over TCP/IP.

The Coherence*Extend clustered service configured above will listen for incoming requests on the `localhost` address and port 9099. When, for example, a client attempts to connect to a Coherence named cache called "dist-extend", the Coherence*Extend clustered service will proxy subsequent requests to the `NamedCache`

with the same name which, in this example, will be a Partitioned cache.

Configuring Coherence*Extend on the Client

A Coherence*Extend client uses the information within an `initiator-config` cache configuration descriptor element to connect to and communicate with a Coherence*Extend clustered service running within a Coherence cluster. For example:

```
<?xml version="1.0"?>

<cache-config xmlns="http://schemas.tangosol.com/cache">
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>dist-extend</cache-name>
      <scheme-name>extend-dist</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>

  <caching-schemes>
    <remote-cache-scheme>
      <scheme-name>extend-dist</scheme-name>
      <service-name>ExtendTcpCacheService</service-name>
      <initiator-config>
        <tcp-initiator>
          <remote-addresses>
            <socket-address>
              <address>localhost</address>
              <port>9099</port>
            </socket-address>
          </remote-addresses>
        </tcp-initiator>
        <outgoing-message-handler>
          <request-timeout>5s</request-timeout>
        </outgoing-message-handler>
      </initiator-config>
    </remote-cache-scheme>
  </caching-schemes>
</cache-config>
```

This cache configuration descriptor defines a caching scheme that connects to a remote Coherence cluster. The `remote-cache-scheme` element has a `tcp-initiator` child element which includes all TCP/IP-specific information needed to connect the client with the Coherence*Extend clustered service running within the remote Coherence cluster.

When the client application retrieves a named cache via the `CacheFactory` using, for example, the name "dist-extend", the Coherence*Extend client will connect to the Coherence cluster via TCP/IP (using the address "localhost" and port 9099) and return a `INamedCache` implementation that routes requests to the `NamedCache` with the same name running within the remote cluster. Note that the `remote-addresses` configuration element can contain multiple `socket-address` child elements. The Coherence*Extend client will attempt to connect to the addresses in a random order, until either the list is exhausted or a TCP/IP connection is established.

Connection Error Detection and Failover

When a Coherence*Extend client service detects that the connection between the client and cluster has been severed (for example, due to a network, software, or hardware failure), the Coherence*Extend client service implementation (i.e. `ICacheService` or `IInvocationService`) will raise a `MemberEventType.Left` event (via the `MemberEventHandler` delegate) and the service will be stopped. If the client application attempts to subsequently use the service, the service will automatically restart itself and attempt to reconnect to the cluster. If the connection is successful, the service will raise a `MemberEventType.Joined` event; otherwise, a fatal exception will be thrown to the client application.

A Coherence*Extend service has several mechanisms for detecting dropped connections. Some mechanisms are inherit to the underlying protocol (i.e. TCP/IP in Extend-TCP), whereas others are implemented by the service itself. The latter mechanisms are configured via the `outgoing-message-handler` configuration element.

The primary configurable mechanism used by a Coherence*Extend client service to detect dropped connections is a request timeout. When the service sends a request to the remote cluster and does not receive a response within the request timeout interval (see `<request-timeout>`), the service assumes that the connection has been dropped. The Coherence*Extend client and clustered services can also be configured to send a periodic heartbeat over the connection (see `<heartbeat-interval>` and `<heartbeat-timeout>`). If the service does not receive a response within the configured heartbeat timeout interval, the service assumes that the connection has been dropped.

4 Configuring a POF Context

Coherence caches are used to cache value objects. Enabling .NET clients to successfully communicate with a Coherence JVM requires a platform-independent serialization format that allows both .NET clients and and Coherence JVMs (including Coherence*Extend Java clients) to properly serialize and deserialize value objects stored in Coherence caches. The Coherence for .NET client library and Coherence*Extend cluster service use a serialization format known as Portable Object Format (POF). POF allows value objects to be encoded into a binary stream in such a way that the platform and language origin of the object is irrelevant.

POF supports all common .NET and Java types out-of-the-box. Any custom .NET and Java class can also be serialized to a POF stream; however, there are additional steps required to do so:

1. Create a .NET class that implements the `IPortableObject` interface.
2. Create a matching Java class that implements the `PortableObject` interface in the same way.
3. Register your custom .NET class on the client.
4. Register your custom Java class on each of the servers running the Coherence*Extend clustered service.

Once these steps are complete, you can cache your custom .NET classes in a Coherence cache in the same way as a built-in data type. Additionally, you will be able to retrieve, manipulate, and store these types from a Coherence or Coherence*Extend JVM using the matching Java classes.

Creating an IPortableObject Implementation (.NET)

Each class that implements `IPortableObject` is capable of self-serializing and deserializing its state to and from a POF data stream. This is achieved in the `ReadExternal` (deserialize) and `WriteExternal` (serialize) methods. Conceptually, all user types are composed of zero or more indexed values (properties) which are read from and written to a POF data stream one by one. The only requirement for a portable class, other than the need to implement the `IPortableObject` interface, is that it must have a default constructor which will allow the POF deserializer to create an instance of the class during deserialization.

Here is an example of a user-defined portable class:

```
public class ContactInfo : IPortableObject
{
    private string name;
    private string street;
    private string city;
    private string state;
    private string zip;

    public ContactInfo()
    {}

    public ContactInfo(string name, string street, string city, string state, string zip)
    {
        Name   = name;
        Street = street;
        City   = city;
        State  = state;
        Zip    = zip;
    }

    public void ReadExternal(IPofReader reader)
    {
        Name   = reader.ReadString(0);
        Street = reader.ReadString(1);
        City   = reader.ReadString(2);
        State  = reader.ReadString(3);
        Zip    = reader.ReadString(4);
    }

    public void WriteExternal(IPofWriter writer)
    {
        writer.WriteString(0, Name);
        writer.WriteString(1, Street);
```

```
        writer.WriteString(2, City);
        writer.WriteString(3, State);
        writer.WriteString(4, Zip);
    }

    // property definitions ommitted for brevity
}
```

Creating a PortableObject Implementation (Java)

An implementation of the portable class in Java is very similar to the one in .NET from the example above:

```
public class ContactInfo
        implements PortableObject
    {
    private String m_sName;
    private String m_sStreet;
    private String m_sCity;
    private String m_sState;
    private String m_sZip;

    public ContactInfo()
        {}

    public ContactInfo(String sName, String sStreet, String sCity,
            String sState, String sZip)
        {
        setName(sName);
        setStreet(sStreet);
        setCity(sCity);
        setState(sState);
        setZip(sZip);
        }

    public void readExternal(PofReader reader)
            throws IOException
        {
        setName(reader.readString(0));
        setStreet(reader.readString(1));
        setCity(reader.readString(2));
        setState(reader.readString(3));
        setZip(reader.readString(4));
        }

    public void writeExternal(PofWriter writer)
            throws IOException
        {
        writer.writeString(0, getName());
        writer.writeString(1, getStreet());
        writer.writeString(2, getCity());
        writer.writeString(3, getState());
        writer.writeString(4, getZip());
        }

    // accessor methods omitted for brevity
    }
```

Registering Custom Types on the .NET Client

Each POF user type is represented within the POF stream as an integer value. As such, POF requires an external mechanism that allows a user type to be mapped to its encoded type identifier (and visa versa). This mechanism uses an XML configuration file to store the mapping information. For example:

```
<?xml version="1.0"?>

<pof-config xmlns="http://schemas.tangosol.com/pof">
  <user-type-list>
```

```
    <!-- include all "standard" Coherence POF user types -->
    <include>assembly://Coherence/Tangosol.Config/coherence-pof-config.xml</include>

    <!-- include all application POF user types -->
    <user-type>
      <type-id>1001</type-id>
      <class-name>My.Example.ContactInfo, MyAssembly</class-name>
    </user-type>
      ...
  </user-type-list>
</pof-config>
```

There are few things to note:

1. Type identifiers for your custom types should start from 1001 or higher, as the numbers below 1000 are reserved for internal use.
2. You need not specify a fully-qualified type name within the `class-name` element. The class and assembly name is enough.

Once you have configured mappings between type identifiers and your custom types, you must configure Coherence for .NET to use them by adding a `serializer` element to your cache configuration descriptor. Assuming that user type mappings from the example above are saved into `my-dotnet-pof-config.xml`, you need to specify a `serializer` element as follows:

```
<remote-cache-scheme>
  <scheme-name>extend-dist</scheme-name>
  <service-name>ExtendTcpCacheService</service-name>
  <initiator-config>
    ...
    <serializer>
      <class-name>Tangosol.IO.Pof.ConfigurablePofContext, Coherence</class-name>
      <init-params>
        <init-param>
          <param-type>string</param-type>
          <param-value>my-dotnet-pof-config.xml</param-value>
        </init-param>
      </init-params>
    </serializer>
  </initiator-config>
</remote-cache-scheme>
```

The `ConfigurablePofContext` type will be used for the POF serializer if one is not explicitly specified. It uses a default configuration file ($AppRoot/coherence-pof-config.xml) if it exists, or a specific file determined by the contents of the `pof-config` element in the Coherence for .NET application configuration file. For example:

```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <section name="coherence" type="Tangosol.Config.CoherenceConfigHandler, Coherence"/>
  </configSections>
  <coherence>
    <pof-config>my-dotnet-pof-config.xml</pof-config>
  </coherence>
</configuration>
```

See **Configuring and Using the Coherence for .NET Client Library (Section 5)** for additional details.

Registering Custom Type in the Cluster

Each Coherence node running the TCP/IP Coherence*Extend clustered service requires a similar POF configuration in order to be able to send and receive objects of these types.

The cluster-side POF configuration file looks similar to the one we created on the client. The only difference is that instead of .NET class names, you must specify the fully qualified Java class names within the `class-`

`name` element. For example, consider the following file called `my-java-pof-config.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE pof-config SYSTEM "pof-config.dtd">

<pof-config>
  <user-type-list>
    <!-- include all "standard" Coherence POF user types -->
    <include>example-pof-config.xml</include>

    <!-- include all application POF user types -->
    <user-type>
      <type-id>1001</type-id>
      <class-name>com.mycompany.example.ContactInfo</class-name>
    </user-type>
    ...
  </user-type-list>
</pof-config>
```

Once your custom types have been added, you must configure the server to use your POF configuration when serializing objects:

```
<proxy-scheme>
  <service-name>ExtendTcpProxyService</service-name>
  <acceptor-config>
    ...
    <serializer>
      <class-name>com.tangosol.io.pof.ConfigurablePofContext</class-name>
      <init-params>
        <init-param>
          <param-type>string</param-type>
          <param-value>my-java-pof-config.xml</param-value>
        </init-param>
      </init-params>
    </serializer>
  </acceptor-config>
  ...
</proxy-scheme>
```

Evolvable Portable User Types

PIF-POF includes native support for both forwards- and backwards-compatibility of the serialized form of portable user types. In .NET, this is accomplished by making user types implement the `IEvolvablePortableObject` interface instead of the `IPortableObject` interface. The `IEvolvablePortableObject` interface is a marker interface that extends bot the `IPortableObject` and `IEvolvable` interfaces. The `IEvolvable` inteface adds three properties to support type versioning.

An `IEvolvable` class has an integer version identifier `n`, where `n >= 0`. When the contents and/or semantics of the serialized form of the `IEvolvable` class changes, the version identifier is increased. Two versions identifiers, `n1` and `n2`, indicate the same version if `n1 == n2`; the version indicated by `n2` is newer than the version indicated by `n1` if `n2 > n1`.

The `IEvolvable` interface is designed to support the evolution of types by the addition of data. Removal of data cannot be safely accomplished as long as a previous version of the type exists that relies on that data. Modifications to the structure or semantics of data from previous versions likewise cannot be safely accomplished as long as a previous version of the type exists that relies on the previous structure or semantics of the data.

When an `IEvolvable` object is deserialized, it retains any unknown data that has been added to newer versions of the type, and the version identifier for that data format. When the `IEvolvable` object is subsequently serialized, it includes both that version identifier and the unknown future data.

When an `IEvolvable` object is deserialized from a data stream whose version identifier indicates an older version, it must default and/or calculate the values for any data fields and properties that have been added since that older version. When the `IEvolvable` object is subsequently serialized, it includes its own version identifier and all of its data. Note that there will be no unknown future data in this case; future data can only

exist when the version of the data stream is newer than the version of the `IEvolvable` type.

The following example demonstrates how the `ContactInfo` .NET type can be modified to support class evolution:

```
public class ContactInfo : IEvolvablePortableObject
{
    private string name;
    private string street;
    private string city;
    private string state;
    private string zip;

    // IEvolvable members
    private int    version;
    private byte[] data;

    public ContactInfo()
    {}

    public ContactInfo(string name, string street, string city, string state, string zip)
    {
        Name   = name;
        Street = street;
        City   = city;
        State  = state;
        Zip    = zip;
    }

    public void ReadExternal(IPofReader reader)
    {
        Name   = reader.ReadString(0);
        Street = reader.ReadString(1);
        City   = reader.ReadString(2);
        State  = reader.ReadString(3);
        Zip    = reader.ReadString(4);
    }

    public void WriteExternal(IPofWriter writer)
    {
        writer.WriteString(0, Name);
        writer.WriteString(1, Street);
        writer.WriteString(2, City);
        writer.WriteString(3, State);
        writer.WriteString(4, Zip);
    }

    public int DataVersion
    {
        get { return version; }
        set { version = value; }
    }

    public byte[] FutureData
    {
        get { return data; }
        set { data = value; }
    }

    public int ImplVersion
    {
        get { return 0; }
    }

    // property definitions ommitted for brevity
}
```

Likewise, the `ContactInfo` Java type can also be modified to support class evolution by implementing the `EvolvablePortableObject` interface:

```
public class ContactInfo
        implements EvolvablePortableObject
    {
    private String m_sName;
    private String m_sStreet;
    private String m_sCity;
    private String m_sState;
    private String m_sZip;

    // Evolvable members
    private int    m_nVersion;
    private byte[] m_abData;

    public ContactInfo()
        {}

    public ContactInfo(String sName, String sStreet, String sCity,
            String sState, String sZip)
        {
        setName(sName);
        setStreet(sStreet);
        setCity(sCity);
        setState(sState);
        setZip(sZip);
        }

    public void readExternal(PofReader reader)
            throws IOException
        {
        setName(reader.readString(0));
        setStreet(reader.readString(1));
        setCity(reader.readString(2));
        setState(reader.readString(3));
        setZip(reader.readString(4));
        }

    public void writeExternal(PofWriter writer)
            throws IOException
        {
        writer.writeString(0, getName());
        writer.writeString(1, getStreet());
        writer.writeString(2, getCity());
        writer.writeString(3, getState());
        writer.writeString(4, getZip());
        }

    public int getDataVersion()
        {
        return m_nVersion;
        }

    public void setDataVersion(int nVersion)
        {
        m_nVersion = nVersion;
        }

    public Binary getFutureData()
        {
        return m_binData;
        }

    public void setFutureData(Binary binFuture)
        {
        m_binData = binFuture;
```

```
        }

    public int getImplVersion()
        {
        return 0;
        }

    // accessor methods omitted for brevity
    }
```

Making Types Portable Without Modification

In some cases, it may be undesirable or impossible to modify an existing user type to make it portable. In this case, you can externalize the portable serialization of a user type by creating an implementation of the `IPofSerializer` in .NET and/or an implementation of the `PofSerializer` interface in Java. For example, an implementation of the `IPofSerializer` interface for the `ContactInfo` type would look like:

```
public class ContactInfoSerializer : IPofSerializer
{
    public object Deserialize(IPofReader reader)
    {
        string name   = reader.ReadString(0);
        string street = reader.ReadString(1);
        string city   = reader.ReadString(2);
        string state  = reader.ReadString(3);
        string zip    = reader.ReadString(4);

        ContactInfo info = new ContactInfo(name, street, city, state, zip);

        info.DataVersion = reader.VersionId;
        info.FutureData  = reader.ReadRemainder();

        return info;
    }

    public void Serialize(IPofWriter writer, object o)
    {
        ContactInfo info = (ContactInfo) o;

        writer.VersionId = Math.Max(info.DataVersion, info.ImplVersion);
        writer.WriteString(0, info.Name);
        writer.WriteString(1, info.Street);
        writer.WriteString(2, info.City);
        writer.WriteString(3, info.State);
        writer.WriteString(4, info.Zip);
        writer.WriteRemainder(info.FutureData);
    }
}
```

An implementation of the `PofSerializer` interface for the `ContactInfo` Java type would look similar:

```
public class ContactInfoSerializer
        implements PofSerializer
    {
    public Object deserialize(PofReader in)
            throws IOException
        {
        String sName   = in.readString(0);
        String sStreet = in.readString(1);
        String sCity   = in.readString(2);
        String sState  = in.readString(3);
        String sZip    = in.readString(4);

        ContactInfo info = new ContactInfo(sName, sStreet, sCity, sState, sZip);

        info.setDataVersion(in.getVersionId());
```

```
        info.setFutureData(in.readRemainder());

        return info;
        }

    public void serialize(PofWriter out, Object o)
            throws IOException
        {
        ContactInfo info = (ContactInfo) o;

        out.setVersionId(Math.max(info.getDataVersion(), info.getImplVersion()));
        out.writeString(0, info.getName());
        out.writeString(1, info.getStreet());
        out.writeString(2, info.getCity());
        out.writeString(3, info.getState());
        out.writeString(4, info.getZip());
        out.writeRemainder(info.getFutureData());
        }
    }
```

To register the `IPofSerializer` implementation for the `ContactInfo` .NET type, specify the class name of the `IPofSerializer` within a `serializer` element under the `user-type` element for the `ContactInfo` user type in the POF configuration file:

```
<?xml version="1.0"?>

<pof-config xmlns="http://schemas.tangosol.com/pof">
  <user-type-list>
    <!-- include all "standard" Coherence POF user types -->
    <include>assembly://Coherence/Tangosol.Config/coherence-pof-config.xml</include>

    <!-- include all application POF user types -->
    <user-type>
      <type-id>1001</type-id>
      <class-name>My.Example.ContactInfo, MyAssembly</class-name>
      <serializer>
        <class-name>My.Example.ContactInfoSerializer, MyAssembly</class-name>
      </serializer>
    </user-type>
      ...
  </user-type-list>
</pof-config>
```

Similarly, you can register the `PofSerializer` implementation for the `ContactInfo` Java type like so:

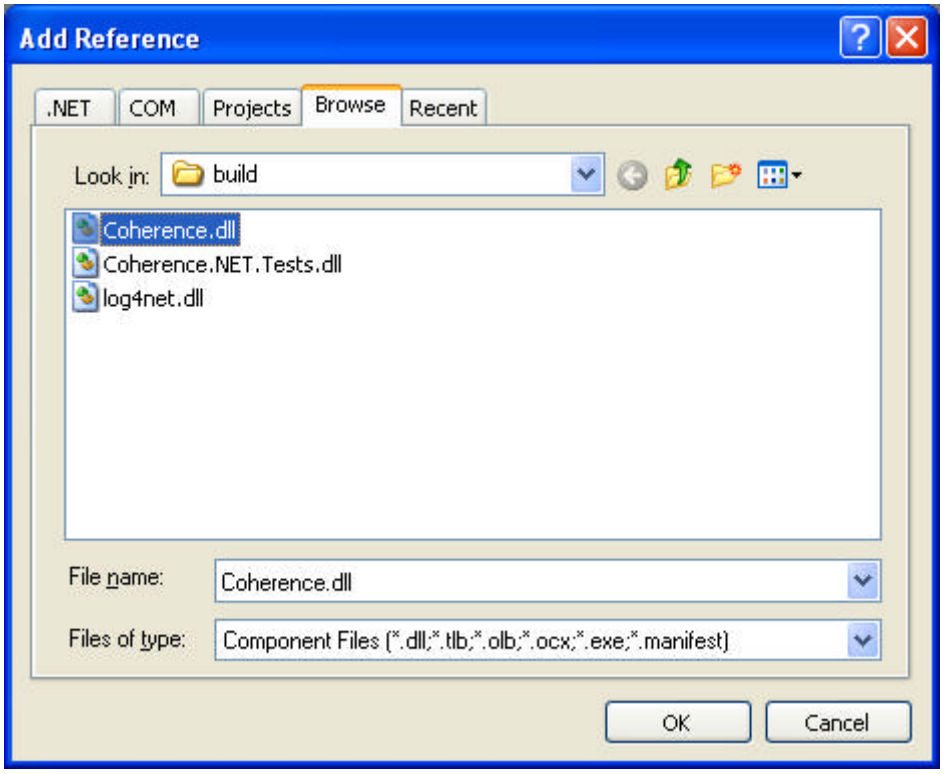
```
<?xml version="1.0"?>
<!DOCTYPE pof-config SYSTEM "pof-config.dtd">

<pof-config>
  <user-type-list>
    <!-- include all "standard" Coherence POF user types -->
    <include>example-pof-config.xml</include>

    <!-- include all application POF user types -->
    <user-type>
      <type-id>1001</type-id>
      <class-name>com.mycompany.example.ContactInfo</class-name>
      <serializer>
        <class-name>com.mycompany.example.ContactInfoSerializer</class-name>
      </serializer>
    </user-type>
    ...
  </user-type-list>
</pof-config>
```

5 Configuring and Using the Coherence for .NET Client Library

To use the Coherence for .NET library in your .NET applications, you must add a reference to the `Coherence.dll` library in your project and create the necessary configuration files.

Creating a reference to the `Coherence.dll`:

1. In your project go to `Project->Add Reference...` or right click on `References` in the Solution Explorer and choose `Add Reference...`
2. In the `Add Reference` window that appears, choose the `Browse` tab and find the `Coherence.dll` library on your file system.



3. Click OK.

Next, you must create the necessary configuration files and specify their paths in the application configuration settings. This is done by adding an application configuration file to your project (if one was not already created) and adding a Coherence for .NET configuration section (i.e. `<coherence/>`) to it.

```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <section name="coherence" type="Tangosol.Config.CoherenceConfigHandler, Coherence"/>
  </configSections>
  <coherence>
    <cache-factory-config>my-coherence.xml</cache-factory-config>
    <cache-config>my-cache-config.xml</cache-config>
    <pof-config>my-pof-config.xml</pof-config>
  </coherence>
</configuration>
```

Elements within the Coherence for .NET configuration section are:

- `cache-factory-config` - contains the path to a configuration descriptor used by the `CacheFactory` to

configure the ConfigurableCacheFactory and Logger used by the `CacheFactory`.

- `cache-config` - contains the path to a cache configuration descriptor which contains the cache configuration described earlier (see **Configuring Coherence*Extend on the Client (Section 3)**). This cache configuration descriptor is used by the DefaultConfigurableCacheFactory.
- `pof-config` - contains the path to a configuration descriptor used by the `ConfigurablePofContext` to register custom types used by the application.

Having added these configuration files, your solution should look like:

5.1 CacheFactory

The `CacheFactory` is the entry point for Coherence for .NET client applications. The `CacheFactory` is a factory for `INamedCache` instances and provides various methods for logging. If not configured explicitly, it uses the default configuration file "coherence.xml" which is an assembly embedded resource. It is possible to override the default configuration file by adding a `cache-factory-config` element to the Coherence for .NET configuration section in the application configuration file and setting its value to the path of the desired configuration file.

```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <section name="coherence" type="Tangosol.Config.CoherenceConfigHandler, Coherence"/>
  </configSections>
  <coherence>
    <cache-factory-config>my-coherence.xml</cache-factory-config>
    ...
  </coherence>
</configuration>
```

This file contains the configuration of two components exposed by the `CacheFactory` via static properties:

- `CacheFactory.ConfigurableCacheFactory` - the `IConfigurableCacheFactory` implementation used by the `CacheFactory` to retrieve, release, and destroy `INamedCache` instances.
- `CacheFactory.Logger` - the `Logger` instance used to log messages and exceptions.

When you are finished using the `CacheFactory` (for example, during application shutdown), the `CacheFactory` should be shutdown via the `Shutdown()` method. This method terminates all services and the `Logger` instance.

5.2 IConfigurableCacheFactory

The `IConfigurableCacheFactory` implementation is specified by the contents of the `configurable-cache-factory-config` element:

- `class-name` - specifies the implementation type by it's assembly qualified name.
- `init-params` - defines parameters used to instantiate the `IConfigurableCacheFactory`. Each parameter is specified via a corresponding `param-type` and `param-value` child element.

Example:

```
<coherence>
  <configurable-cache-factory-config>
    <class-name>Tangosol.Net.DefaultConfigurableCacheFactory, Coherence</class-name>
    <init-params>
      <init-param>
        <param-type>string</param-type>
        <param-value>simple-cache-config.xml</param-value>
      </init-param>
    </init-params>
  </configurable-cache-factory-config>
</coherence>
```

If an `IConfigurableCacheFactory` implementation is not defined in the configuration, the a default

implementation is used (`DefaultConfigurableCacheFactory`).

5.3 DefaultConfigurableCacheFactory

The `DefaultConfigurableCacheFactory` provides a facility to access caches declared in the cache configuration descriptor described earlier (see the **Client-side Cache Configuration Descriptor (Section 3)** section). The default configuration file used by the `DefaultConfigurableCacheFactory` is $AppRoot/coherence-cache-config.xml, where $AppRoot is the working directory (in the case of a Windows Forms application) or the root of the application (in the case of a Web application).
If you wish to specify another cache configuration descriptor file, you can do so by adding a `cache-config` element to the Coherence for .NET configuration section in the application configuration file with its value set to the path of the configuration file.

Example:

```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <section name="coherence" type="Tangosol.Config.CoherenceConfigHandler, Coherence"/>
  </configSections>
  <coherence>
    <cache-config>my-cache-config.xml</cache-config>
    ...
  </coherence>
</configuration>
```

5.4 Logger

The `Logger` is configured using the `logging-config` element:

- `destination` - determines the type of `LogOutput` used by the `Logger`. Valid values are:
  - "common-logger" for `Common.Logging`
  - "stderr" for `Console.Error`
  - "stdout" for `Console.Out`
  - file path if messages should be directed to a file
- `severity-level` - determines the log level that a message must meet or exceed in order to be logged.
- `message-format` - determines the log message format.
- `character-limit` - determines the the maximum number of characters that the logger daemon will process from the message queue before discarding all remaining messages in the queue.

Example:

```
<coherence>
  <logging-config>
    <destination>common-logger</destination>
    <severity-level>5</severity-level>
    <message-format>(thread={thread}): {text}</message-format>
    <character-limit>8192</character-limit>
  </logging-config>
</coherence>
```

The `CacheFactory` provides several static methods for retrieving and releasing `INamedCache` instances:

- `GetCache(String cacheName)` - retrieves an `INamedCache` implementation that corresponds to the `NamedCache` with the specified "cacheName" running within the remote Coherence cluster.
- `ReleaseCache(INamedCache cache)` - releases all local resources associated with the specified instance of the cache. After a cache is release, it can no longer be used.
- `DestroyCache(INamedCache cache)` - destroys the specified cache across the Coherence cluster.

Methods used to log messages and exceptions are:

- `IsLogEnabled(int level)` - determines if the `Logger` would log a message with the given severity level.

- `Log(Exception e, int severity)` - logs an exception with the specified severity level.
- `Log(String message, int severity)` - logs a text message with the specified severity level.
- `Log(String message, Exception e, int severity)` - logs a text message and an exception with the specified severity level.

Logging levels are defined by the values of the `CacheFactory.LogLevel` enum values (in ascending order):

- `Always`
- `Error`
- `Warn`
- `Info`
- `Debug` - (default log level)
- `Quiet`
- `Max`

Common.Logging

Common.Logging (http://netcommon.sourceforge.net/) is an open source library that allows you to plug in various popular open source logging libraries behind a well-defined set of interfaces. The libraries currently supported are Log4Net (versions 1.2.9 and 1.2.10) and NLog. Common.Logging is currently used by the Spring.NET framework and will likely be used in the future releases of IBatis.NET and NHibernate, so you might want to consider it if you are using one or more of these frameworks in combination with Coherence for .NET, as it will allow you to configure logging consistently throughout the application layers.

Coherence for .NET does not include the Common.Logging library. If you would like to use the "common-logger" Logger configuration, you must download the Common.Logging assembly and include a reference to it in your project. You can download the Common.Logging assemblies for both .NET 1.1 and 2.0 from the following location:

http://netcommon.sourceforge.net/

The Coherence for .NET Common.Logging Logger implementation was compiled against the signed release version of these assemblies.

5.5 INamedCache

The `INamedCache` interface extends `IDictionary`, so it can be manipulated in ways similar to a dictionary. Once obtained, `INamedCache` instances expose several properties:

- `CacheName` - the cache name.
- `Count` - the cache size.
- `IsActive` - determines if the cache is active (that is, it has not been released or destroyed).
- `Keys` - collection of all keys in the cache mappings.
- `Values` - collection of all values in the cache mappings.

The value for the specified key can be retreived via `cache[key]`. Similarly, a new value can be added, or an old value can be modified by setting this property to the new value: `cache[key] = value`.
The collection of cache entries can be accessed via `GetEnumerator()` which can be used to iterate over the mappings in the cache.

The `INamedCache` interface provides a number of methods used to manipulate the contents of the cache:

- `Clear()` - removes all the mappings from the cache.
- `Contains(Object key)` - determines if the cache has a mapping for the specified key.
- `GetAll(ICollection keys)` - returns all mappings for the specified keys collection.
- `Insert(Object key, Object value)` - places a new mapping into the cache. If a mapping for the specified key already exists, it's value will be overwritten by the specified value and the old value will be

returned.

- `Insert(Object key, Object value, long millis)` - places a new mapping into the cache, but with an expiry period specified by a number of milliseconds.
- `InsertAll(IDictionary dictionary)` - copies all the mappings from the specified dictionary to the cache.
- `Remove(Object key)` - Removes the mapping for the specified key if it is present and returns the value it was mapped to.

`INamedCache` interface also extends the following three interfaces: IQueryCache, IObservableCache, IInvocableCache.

5.6 IQueryCache

The `IQueryCache` interface exposes the ability to query a cache using various filters.

- `GetKeys(IFilter filter)` - returns a collection of the keys contained in this cache for entries that satisfy the criteria expressed by the filter.
- `GetEntries(IFilter filter)` - returns a collection of the entries contained in this cache that satisfy the criteria expressed by the filter.
- `GetEntries(IFilter filter, IComparer comparer)` - returns a collection of the entries contained in this cache that satisfy the criteria expressed by the filter. It is guaranteed that the enumerator will traverse the collection in the order of ascending entry values, sorted by the specified comparer or according to the natural ordering if the "comparer" is null.

Additionally, the `IQueryCache` interface includes the ability to add and remove indexes. Indexes are used to correlate values stored in the cache to their corresponding keys and can dramatically increase the performance of the `GetKeys` and `GetEntries` methods.

- `AddIndex(IValueExtractor extractor, bool isOrdered, IComparer comparator)` - adds an index to this cache that correlates the values extracted by the given `IValueExtractor` to the keys to the corresponding entries. Additionally, the index information can be optionally ordered.
- `RemoveIndex(IValueExtractor extractor)` - removes an index from this cache.

For example:

The following code would perform an efficient query of the keys of all entries that have an age property value greater or equal to 55.

```
IValueExtractor extractor = new ReflectionExtractor("getAge");
cache.AddIndex(extractor, true, null);

ICollection keys = cache.GetKeys(new GreaterEqualsFilter(extractor, 55));
```

5.7 IObservableCache

`IObservableCache` interface enables an application to receive events when the contents of a cache changes. To register interest in change events, an application adds a `Listener` implementation to the cache that will receive events that include information about the event type (inserted, updated, deleted), the key of the modified entry, and the old and new values of the entry.

- `AddCacheListener(ICacheListener listener)` - adds a standard cache listener that will receive all events (inserts, updates, deletes) emitted from the cache, including their keys, old, and new values.
- `RemoveCacheListener(ICacheListener listener)` - removes a standard cache listener that was previously registered.
- `AddCacheListener(ICacheListener listener, object key, bool isLite)` - adds a cache listener for a specific key. If `isLite` is true, the events may not contain the old and new values.
- `RemoveCacheListener(ICacheListener listener, object key)` - removes a cache listener that was previously registered using the specified key.
- `AddCacheListener(ICacheListener listener, IFilter filter, bool isLite)` - adds a cache listener that receives events based on a filter evaluation. If `isLite` is true, the events may not contain the old and new values.
- `RemoveCacheListener(ICacheListener listener, IFilter filter)` - removes a cache listener that

previously registered using the specified filter.

Listener registered using the filter-based method will receive all event types (inserted, updated, and deleted). To further filter the events, wrap the filter in a `CacheEventFilter` using a `CacheEventMask` enumeration value to specify which type of events should be monitored.

For example:

A filter that evaluates to true if an `Employee` object is inserted into a cache with an `IsMarried` property value set to true.

```
new CacheEventFilter(CacheEventMask.Inserted, new EqualsFilter("IsMarried", true));
```

A filter that evaluates to true if any object is removed from a cache.

```
new CacheEventFilter(CacheEventMask.Deleted);
```

A filter that evaluates to true if when an `Employee` object `LastName` property is changed from "Smith".

```
new CacheEventFilter(CacheEventMask.UpdatedLeft, new EqualsFilter("LastName", "Smith"));
```

5.8 IInvocableCache

An `IInvocableCache` is a cache against which both entry-targeted processing and aggregating operations can be invoked. The operations against the cache contents are executed by (and thus within the localized context of) a cache. This is particularly useful in a distributed environment, because it enables the processing to be moved to the location at which the entries-to-be-processed are being managed, thus providing efficiency by localization of processing.

- `Invoke(object key, IEntryProcessor agent)` - invokes the passed processor against the entry specified by the passed key, returning the result of the invocation.
- `InvokeAll(ICollection keys, IEntryProcessor agent)` - invokes the passed processor against the entries specified by the passed keys, returning the result of the invocation for each.
- `InvokeAll(IFilter filter, IEntryProcessor agent)` - invokes the passed processor against the entries that are selected by the given filter, returning the result of the invocation for each.
- `Aggregate(ICollection keys, IEntryAggregator agent)` - performs an aggregating operation against the entries specified by the passed keys.
- `Aggregate(IFilter filter, IEntryAggregator agent)` - performs an aggregating operation against the entries that are selected by the given filter.

Filters

The `IQueryCache` interface provides the ability to search for cache entries that meet a given set of criteria, expressed using a `IFilter` implementation.

All filters must implement the `IFilter` interface:

- `Evaluate(object o)` - apply a test to the specified object and return true if the test passes, false otherwise.

Coherence for .NET includes several `IFilter` implementations in the `Tangosol.Util.Filter` namespace.

The following code would retrieve the keys of all entries that have a value equal to 5.

```
EqualsFilter equalsFilter = new EqualsFilter(IdentityExtractor.Instance, 5);
ICollection  keys         = cache.GetKeys(equalsFilter);
```

The following code would retrieve the keys of all entries that have a value greater or equal to 55.

```
GreaterEqualsFilter greaterEquals = new GreaterEqualsFilter(IdentityExtractor.Instance, 55);
ICollection         keys          = cache.GetKeys(greaterEquals);
```

The following code would retrieve all cache entries that have a value that begins with "Belg".

```
LikeFilter  likeFilter = new LikeFilter(IdentityExtractor.Instance, "Belg%", '\\', true);
ICollection entries    = cache.GetEntries(likeFilter);
```

The following code would retrieve all cache entries that have a value that ends with "an" (case sensitive) **or** begins with "An" (case insensitive).

```
OrFilter    orFilter = new OrFilter(new LikeFilter(IdentityExtractor.Instance, "%an", '\\', f
ICollection entries  = cache.GetEntries(orFilter);
```

Extractors

Extractors are used to extract values from an object.

All extractors must implement the `IValueExtractor` interface:

- `Extract(object target)` - extract the value from the passed object.

Coherence for .NET includes the following extractors:

- `IdentityExtractor` is a trivial implementation that does not actually extract anything from the passed value, but returns the value itself.
- `KeyExtractor` is a special purpose implementation that serves as an indicator that a query should be run against the key objects rather than the values.
- `ReflectionExtractor` extracts a value from a specified object property.
- `MultiExtractor` is composite `IValueExtractor` implementation based on an array of extractors. All extractors in the array are applied to the same target object and the result of the extraction is a `IList` of extracted values.
- `ChainedExtractor` is composite `IValueExtractor` implementation based on an array of extractors. The extractors in the array are applied sequentially left-to-right, so a result of a previous extractor serves as a target object for a next one.

The following code would retrieve all cache entries with keys greater than 5:

```
IValueExtractor extractor = new KeyExtractor(IdentityExtractor.Instance);
IFilter         filter    = new GreaterFilter(extractor, 5);
ICollection     entries   = cache.GetEntries(filter);
```

The following code would retrieve all cache entries with values containing a `City` property equal to "city1":

```
IValueExtractor extractor = new ReflectionExtractor("City");
IFilter         filter    = new EqualsFilter(extractor, "city1");
ICollection     entries   = cache.GetEntries(filter);
```

Processors

A processor is an invocable agent that operates against the entry objects within a cache.

All processors must implement the `IEntryProcessor` interface:

- `Process(IInvocableCacheEntry entry)` - process the specified entry.
- `ProcessAll(ICollection entries)` - process a collection of entries.

Coherence for .NET includes several `IEntryProcessor` implementations in the `Tangosol.Util.Processor` namespace.

The following code demonstrates a conditional put. The value mapped to "key1" is set to 680 only if the current mapped value is greater than 600.

```
IFilter         greaterThen600 = new GreaterFilter(IdentityExtractor.Instance, 600);
IEntryProcessor processor      = new ConditionalPut(greaterThen600, 680);
cache.Invoke("key1", processor);
```

The following code uses the `UpdaterProcessor` to update the value of the `Degree` property on a `Temperature` object with key "BGD" to the new value 26.

```
cache.Insert("BGD", new Temperature(25, 'c', 12));
IValueUpdater   updater   = new ReflectionUpdater("setDegree");
IEntryProcessor processor = new UpdaterProcessor(updater, 26);
object          result    = cache.Invoke("BGD", processor);
```

Aggregators

An aggregator represents processing that can be directed to occur against some subset of the entries in an `IInvocableCache`, resulting in an aggregated result. Common examples of aggregation include functions such as minimum, maximum, sum and average. However, the concept of aggregation applies to any process that needs to evaluate a group of entries to come up with a single answer. Aggregation is explicitly capable of being run in parallel, for example in a distributed environment.

All aggregators must implement the `IEntryAggregator` interface:

- `Aggregate(ICollection entries)` - process a collection of entries in order to produce an aggregate result.

Coherence for .NET includes several `IEntryAggregator` implementations in the `Tangosol.Util.Aggregator` namespace.

The following code returns the size of the cache:

```
IEntryAggregator aggregator = new Count();
object           result     = cache.Aggregate(cache.Keys, aggregator);
```

In following code returns an `IDictionary` with keys equal to the unique values in the cache and values equal to the number of instances of the corresponding value in the cache:

```
IEntryAggregator aggregator = GroupAggregator.CreateInstance(IdentityExtractor.Instance, new
object           result     = cache.Aggregate(cache.Keys, aggregator);
```

Like cached value objects, all custom `IFilter`, `IExtractor`, `IProcessor` and `IAggregator` implementation classes must be correctly registered in the POF context of the .NET application and cluster-side node to which the client is connected. As such, corresponding Java implementations of the custom .NET types must be created, compiled, and deployed on the cluster-side node. Note that the actual execution of the these custom types is performed by the Java implementation and not the .NET implementation.

See **Configuring a POF Context (Section 4)** for additional details.

6 Launching a Coherence DefaultCacheServer Process

To start a `DefaultCacheServer` that uses the cluster-side Coherence cache configuration described earlier to allow Coherence for .NET clients to connect to the Coherence cluster via TCP/IP, you need to do the following:

- Change the current directory to the Oracle Coherence library directory (`%COHERENCE_HOME%\lib` on Windows and `$COHERENCE_HOME/lib` on Unix).
- Make sure that the paths are configured so that the Java command will run.
- Start the `DefaultCacheServer` command line application with the `-Dtangosol.coherence.cacheconfig` system property set to the location of the cluster-side Coherence cache configuration descriptor described earlier.

For example (note that the following command is broken up into multiple lines here only for formatting purposes; this is a single command typed on one line):

```
java -cp coherence.jar -Dtangosol.coherence.cacheconfig=file://<path to the server-side cache
     com.tangosol.net.DefaultCacheServer
```

7 Local Cache

Overview

A **Local Cache** is just that: A cache that is local to (completely contained within) a particular .NET application. There are several attributes of the Local Cache that are particularly interesting:

- The Local Cache implements the same standard cache interfaces that a remote cache implements (`ICache`, `IObservableCache`, `IConcurrentCache`, `IQueryCache`, and `IInvocableCache`), meaning that there is no programming difference between using a local and a remote cache.
- The Local Cache can be size-limited. This means that the Local Cache can restrict the number of entries that it caches, and automatically evict entries when the cache becomes full. Furthermore, both the sizing of entries and the eviction policies are customizable, for example allowing the cache to be size-limited based on the memory utilized by the cached entries. The default eviction policy uses a combination of Most Frequently Used (MFU) and Most Recently Used (MRU) information, scaled on a logarithmic curve, to determine what cache items to evict. This algorithm is the best general-purpose eviction algorithm because it works well for short duration and long duration caches, and it balances frequency versus recentness to avoid cache thrashing. The pure LRU and pure LFU algorithms are also supported, as well as the ability to plug in custom eviction policies.
- The Local Cache supports automatic expiration of cached entries, meaning that each cache entry can be assigned a time-to-live value in the cache. Furthermore, the entire cache can be configured to flush itself on a periodic basis or at a preset time.
- The Local Cache is thread safe and highly concurrent.
- The Local Cache provides cache "get" statistics. It maintains hit and miss statistics. These runtime statistics can be used to accurately project the effectiveness of the cache, and adjust its size-limiting and auto-expiring settings accordingly while the cache is running.

Configuring and Using a Local Cache

The Coherence for .NET Local Cache functionality is implemented by the `Tangosol.Net.Cache.LocalCache` class. As such, it can be programatically instantiated and configured; however, it is recommended that a `LocalCache` be configured via a cache configuration descriptor, just like any other Coherence for .NET cache. For example:

```
<?xml version="1.0"?>

<cache-config xmlns="http://schemas.tangosol.com/cache">
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>example-local-cache</cache-name>
      <scheme-name>example-local</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <local-scheme>
      <scheme-name>example-local</scheme-name>
      <eviction-policy>LRU</eviction-policy>
      <high-units>32000</high-units>
      <low-units>10</low-units>
      <unit-calculator>FIXED</unit-calculator>
      <expiry-delay>10ms</expiry-delay>
      <flush-delay>1000ms</flush-delay>
      <cachestore-scheme>
        <class-scheme>
          <class-name>ExampleCacheStore</class-name>
        </class-scheme>
      </cachestore-scheme>
      <pre-load>true</pre-load>
    </local-scheme>
  </caching-schemes>
</cache-config>
```

A reference to a configured Local Cache can then be obtained by name via the `CacheFactory` class:

```
INamedCache cache = CacheFactory.GetCache("example-local-cache");
```

Once you obtain a reference to a Local Cache, you can simply use it as any other type of cache provided by Coherence.

Cleaning up the resources associated with a LocalCache

Instances of all `INamedCache` implementations, including `LocalCache`, should be explicitly released by calling the `INamedCache.Release()` method when they are no longer needed, in order to free up any resources they might hold.

If the particular `INamedCache` is used for the duration of the application, then the resources will be cleaned up when the application is shut down or otherwise stops. However, if it is only used for a period of time, the application should call its `Release()` method when finished using it.

Alternatively, you can leverage the fact that `INamedCache` extends `IDisposable` and that all cache implementations delegate a call to `IDisposable.Dispose()` to `INamedCache.Release()`. This means that if you need to obtain and release a cache instance within a single method, you can do so via a `using` block:

```
using (INamedCache cache = CacheFactory.GetCache("my-cache"))
{
   // use cache as usual
}
```

After the `using` block terminates, `IDisposable.Dispose()` will be call on the `INamedCache` instance, and all resources associated with it will be released.

8 Near Cache

Overview

A **Near Cache** provides local cache access to recently- and/or often-used data, backed by a centralized or multi-tiered cache that is used to load-on-demand for local cache misses. Near Caches have configurable levels of cache coherency, from the most basic expiry-based caches and invalidation-based caches, up to advanced data-versioning caches that can provide guaranteed coherency. The result is a tunable balance between the preservation of local memory resources and the performance benefits of truly local caches.

A Near Cache is actually an `INamedCache` implementation that wraps two caches - a front cache (assumed to be "inexpensive" and probably "incomplete") and a back cache (assumed to be "complete" and "correct", but more "expensive") - using a read-through/write-through approach. If the back cache implements the `IObservableCache` interface, the Near Cache provides four different strategies of invalidating the front cache entries that have changed by other processes in the back cache:

- **Listen None** strategy instructs the cache not to listen for invalidation events at all. This is the best choice for raw performance and scalability when business requirements permit the use of data which might not be absolutely current. Freshness of data can be guaranteed by use of a sufficiently brief eviction policy for the front cache.
- **Listen Present** strategy instructs the Near Cache to listen to the back cache events related only to the items currently present in the front cache. This strategy works best when each instance of a front cache contains distinct subset of data relative to the other front cache instances (e.g. sticky data access patterns).
- **Listen All** strategy instructs the Near Cache to listen to all back cache events. This strategy is optimal for read-heavy tiered access patterns where there is significant overlap between the different instances of front caches.
- **Listen Auto** strategy instructs the Near Cache to switch automatically between Listen Present and Listen All strategies based on the cache statistics.

Configuring and Using a Near Cache

The Coherence for .NET Near Cache functionality is implemented by the `Tangosol.Net.Cache.NearCache` class. As such, it can be programatically instantiated and configured; however, it is recommended that a `NearCache` be configured via a cache configuration descriptor, just like any other Coherence for .NET cache.

A typical Near Cache is configured to use a **Local Cache (Section 7)** (thread safe, highly concurrent, size-limited and/or auto-expiring local cache) as the front cache and a remote cache as a back cache. A Near Cache is configured via the `near-scheme` which has two child elements - a `front-scheme` for configuring a local (front) cache and a `back-scheme` for defining a remote (back) cache. For example:

```
<?xml version="1.0"?>

<cache-config xmlns="http://schemas.tangosol.com/cache">
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>example-near-cache</cache-name>
      <scheme-name>example-near</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <local-scheme>
      <scheme-name>example-local</scheme-name>
    </local-scheme>

    <near-scheme>
      <scheme-name>example-near</scheme-name>
      <front-scheme>
        <local-scheme>
          <scheme-ref>example-local</scheme-ref>
        </local-scheme>
      </front-scheme>
      <back-scheme>
```

```
        <remote-cache-scheme>
          <scheme-ref>example-remote</scheme-ref>
        </remote-cache-scheme>
      </back-scheme>
    </near-scheme>

    <remote-cache-scheme>
      <scheme-name>example-remote</scheme-name>
      <service-name>ExtendTcpCacheService</service-name>
      <initiator-config>
        <tcp-initiator>
          <remote-addresses>
            <socket-address>
              <address>localhost</address>
              <port>9099</port>
            </socket-address>
          </remote-addresses>
        </tcp-initiator>

        <outgoing-message-handler>
          <request-timeout>30s</request-timeout>
        </outgoing-message-handler>
      </initiator-config>
    </remote-cache-scheme>
  </caching-schemes>
</cache-config>
```

A reference to a configured Near Cache can then be obtained by name via the `CacheFactory` class:

```
INamedCache cache = CacheFactory.GetCache("example-near-cache");
```

Once you obtain a reference to a Near Cache, you can simply use it as any other type of cache provided by Coherence.

Cleaning up the resources associated with a NearCache

Instances of all `INamedCache` implementations, including `NearCache`, should be explicitly released by calling the `INamedCache.Release()` method when they are no longer needed, in order to free up any resources they might hold.

If the particular `INamedCache` is used for the duration of the application, then the resources will be cleaned up when the application is shut down or otherwise stops. However, if it is only used for a period of time, the application should call its `Release()` method when finished using it.

Alternatively, you can leverage the fact that `INamedCache` extends `IDisposable` and that all cache implementations delegate a call to `IDisposable.Dispose()` to `INamedCache.Release()`. This means that if you need to obtain and release a cache instance within a single method, you can do so via a `using` block:

```
using (INamedCache cache = CacheFactory.GetCache("my-cache"))
{
   // use cache as usual
}
```

After the `using` block terminates, `IDisposable.Dispose()` will be call on the `INamedCache` instance, and all resources associated with it will be released.

9 Continuous Query Cache

Overview

While it is possible to obtain a point in time query result from a Coherence for .NET cache, and it is possible to receive events that would change the result of that query, Coherence for .NET provides a feature that combines a query result with a continuous stream of related events in order to maintain an up-to-date query result in a real-time fashion. This capability is called **Continuous Query**, because it has the same effect as if the desired query had zero latency *and* the query were being executed several times every millisecond!

Coherence for .NET implements the Continuous Query functionality by materializing the results of the query into a Continuous Query Cache, and then keeping that cache up-to-date in real-time using event listeners on the query. In other words, a Coherence for .NET Continuous Query is a cached query result that never gets out-of-date.

Uses of Continuous Query Caching

There are several different general use categories for Continuous Query Caching:

- It is an ideal building block for Complex Event Processing (CEP) systems and event correlation engines.
- It is ideal for situations in which an application repeats a particular query, and would benefit from always having instant access to the up-to-date result of that query.
- A Continuous Query Cache is analogous to a *materialized view*, and is useful for accessing and manipulating the results of a query using the standard `INamedCache` API, as well as receiving an ongoing stream of events related to that query.
- A Continuous Query Cache can be used in a manner similar to a **Near Cache (Section 8)**, because it maintains an up-to-date set of data locally *where it is being used*, for example on a particular server node or on a client desktop; note that a Near Cache is invalidation-based, but the Continuous Query Cache actually maintains its data in an up-to-date manner.

An example use case is a trading system desktop, in which a trader's open orders and all related information needs to be maintained in an up-to-date manner at all times. **By combining the Coherence*Extend functionality with Continuous Query Caching, an application can support literally tens of thousands of concurrent users.**

Continuous Query Caches are useful in almost every type of application, including both client-based and server-based applications, because they provide the ability to very easily and efficiently maintain an up-to-date local copy of a specified sub-set of a much larger and potentially distributed cached data set.

The Continuous Query Cache

The Coherence for .NET implementation of Continuous Query is found in the `Tangosol.Net.Cache.ContinuousQueryCache` class. This class, like all Coherence for .NET caches, implements the standard `INamedCache` interface, which includes the following capabilities:

- Cache access and manipulation using the `IDictionary` interface: `INamedCache` extends the standard `IDictionary` interface from the .NET Collections Framework, which is the same interface implemented by the .NET `Hashtable` class.
- Events for all objects modifications that occur within the cache: `INamedCache` extends the `IObservableCache` interface.
- Identity-based cluster-wide locking of objects in the cache: `INamedCache` extends the `IConcurrentCache` interface.
- Querying the objects in the cache: `INamedCache` extends the `IQueryCache` interface.
- Distributed Parallel Processing and Aggregation of objects in the cache: `INamedCache` extends the `IInvocableCache` interface.

Since the `ContinuousQueryCache` implements the `INamedCache` interface, which is the same API provided by all Coherence for .NET caches, it is extremely simple to use, and it can be easily substituted for another cache when its functionality is called for.

Constructing a Continuous Query Cache

There are two items that define a Continuous Query Cache:

1. The underlying cache that it is based on;
2. A query of that underlying cache that produces the sub-set that the Continuous Query Cache will cache.

The underlying cache is any Coherence for .NET cache, including another Continuous Query Cache. A cache is usually obtained from a `CacheFactory`, which allows the developer to simply specify the name of the cache and have it automatically configured based on the application's cache configuration information; for example:

```
INamedCache cache = CacheFactory.GetCache("orders");
```

The query is the same type of query that would be used to query any other cache; for example:

```
Filter filter = new AndFilter(new EqualsFilter("getTrader", traderId),
new EqualsFilter("getStatus", Status.OPEN));
```

Similarly, the Continuous Query Cache is constructed from those same two pieces:

```
ContinuousQueryCache cacheOpenTrades = new ContinuousQueryCache(cache, filter);
```

Once you obtain a reference to a Continuous Query Cache, you can simply use it as any other type of cache provided by Coherence.

Cleaning up the resources associated with a ContinuousQueryCache

Instances of all `INamedCache` implementations, including `ContinuousQueryCache`, should be explicitly released by calling the `INamedCache.Release()` method when they are no longer needed, in order to free up any resources they might hold.

If the particular `INamedCache` is used for the duration of the application, then the resources will be cleaned up when the application is shut down or otherwise stops. However, if it is only used for a period of time, the application should call its `Release()` method when finished using it.

Alternatively, you can leverage the fact that `INamedCache` extends `IDisposable` and that all cache implementations delegate a call to `IDisposable.Dispose()` to `INamedCache.Release()`. This means that if you need to obtain and release a cache instance within a single method, you can do so via a `using` block:

```
using (INamedCache cache = CacheFactory.GetCache("my-cache"))
{
   // use cache as usual
}
```

After the `using` block terminates, `IDisposable.Dispose()` will be call on the `INamedCache` instance, and all resources associated with it will be released.

Semi- and Fully-Materialized Views

When constructing a `ContinuousQueryCache`, it is possible to specify that the cache should only keep track of the keys that result from the query, and obtain the values from the underlying cache only when they are asked for. This feature may be useful for creating a `ContinuousQueryCache` that represents a very large query result set, or if the values are never or rarely requested. To specify that only the keys should be cached, use the constructor that allows the Is`CacheValues` property to be configured; for example:

```
ContinuousQueryCache cacheOpenTrades = new ContinuousQueryCache(cache, filter, false);
```

If necessary, the IsCacheValues property can also be modified after the cache has been instantiated; for example:

```
cacheOpenTrades.IsCacheValues = true;
```

IsCacheValues property and Event Listeners

If the `ContinuousQueryCache` has any standard (non-lite) event listeners, or if any of the event listeners are filtered, then the `CacheValues` property will automatically be set to true, because the `ContinuousQueryCache` uses the locally cached values to filter events and to supply the old and new values for the events that it raises.

9.1 Listening to a Continuous Query Cache

Since the `ContinuousQueryCache` is itself observable, it is possible for the client to place one or more event listeners onto it. For example:

```
ContinuousQueryCache cacheOpenTrades = new ContinuousQueryCache(cache, filter);
cacheOpenTrades.AddCacheListener(listener);
```

Assuming some processing has to occur against every item that is already in the cache **and** every item added to the cache, there are two approaches. First, the processing could occur then a listener could be added to handle any later additions:

```
ContinuousQueryCache cacheOpenTrades = new ContinuousQueryCache(cache, filter);
foreach (ICacheEntry entry in cacheOpenTrades.Entries)
    {
// .. process the cache entry
    }
cacheOpenTrades.AddCacheListener(listener);
```

However, **that code is incorrect** because it allows events that occur in the split second after the iteration and before the listener is added to be missed! The alternative is to add a listener first, so no events are missed, and then do the processing:

```
ContinuousQueryCache cacheOpenTrades = new ContinuousQueryCache(cache, filter);
cacheOpenTrades.AddCacheListener(listener);
foreach (ICacheEntry entry in cacheOpenTrades.Entries)
    {
    // .. process the cache entry
    }
```

However, it is possible that the same entry will show up in both an event an in the `IEnumerator`, and the events can be asynchronous, so the sequence of operations cannot be guaranteed.

The solution is to provide the listener during construction, and it will receive one event for each item that is in the Continuous Query Cache, whether it was there to begin with (because it was in the query) or if it got added during or after the construction of the cache:

```
ContinuousQueryCache cacheOpenTrades = new ContinuousQueryCache(cache, filter, listener);
```

Achieving a stable materialized view

The `ContinuousQueryCache` implementation faced the same challenge: How to assemble an exact point-in-time snapshot of an underlying cache *while receiving a stream of modification events from that same cache*. The solution has several parts. First, Coherence for .NET supports an option for synchronous events, which provides a set of ordering guarantees. Secondly, the `ContinuousQueryCache` has a two-phase implementation of its initial population that allows it to first query the underlying cache and then subsequently resolve all of the events that came in during the first phase. Since achieving these guarantees of data visibility without any missing or repeated events is fairly complex, the `ContinuousQueryCache` allows a developer to pass a listener during construction, thus avoiding exposing these same complexities to the application developer.

Support for synchronous and asynchronous listeners

By default, listeners to the `ContinuousQueryCache` will have their events delivered asynchronously. However, the `ContinuousQueryCache` does respect the option for synchronous events as provided by the `CacheListenerSupport.ISynchronousListener` interface.

9.2 Making a Continuous Query Cache Read-Only

The `ContinuousQueryCache` can be made into a read-only cache; for example:

```
cacheOpenTrades.IsReadOnly = true;
```

A read-only `ContinuousQueryCache` will not allow objects to be added to, changed in, removed from or locked in the cache.

Once a `ContinuousQueryCache` has been set to read-only, it cannot be changed back to read/write.

10 Remote Invocation Service

Overview

Coherence for .NET provides a **Remote Invocation Service** which allows execution of single-pass agents (called `IInvocable` objects) within the cluster-side JVM to which the client is connected. Agents are simply runnable application classes that implement the `IInvocable` interface. Agents can execute any arbitrary action and can use any cluster-side services (cache services, grid services, etc.) necessary to perform their work. The agent operations can also be stateful, which means that their state is serialized and transmitted to the grid nodes on which the agent is run.

Configuring and Using the Remote Invocation Service

A Remote Invocation Service is configured using the `remote-invocation-scheme` element in the cache configuration descriptor. For example:

```
<remote-invocation-scheme>
  <scheme-name>example-invocation</scheme-name>
  <service-name>ExtendTcpInvocationService</service-name>
  <initiator-config>
    <tcp-initiator>
      <remote-addresses>
        <socket-address>
          <address>localhost</address>
          <port>9099</port>
        </socket-address>
      </remote-addresses>
    </tcp-initiator>

    <outgoing-message-handler>
      <request-timeout>30s</request-timeout>
    </outgoing-message-handler>
  </initiator-config>
</remote-invocation-scheme>
```

A reference to a configured Remote Invocation Service can then be obtained by name via the `CacheFactory` class:

```
IInvocationService service = CacheFactory.GetInvocationService("ExtendTcpInvocationService");
```

To execute an agent on the grid node to which the client is connected requires **only one line of code**:

```
IDictionary result = service.Query(new MyTask(), null);
```

The single result of the execution will be keyed by the local `Member`, which can be retrieved by calling `CacheFactory.GetConfigurableCacheFactory().GetLocalMember()`.

Like cached value objects, all result `IInvocable` implementation classes must be correctly registered in the POF context of the .NET application and cluster-side node to which the client is connected. As such, a Java implementation of the `IInvocable` task (a `com.tangosol.net.Invocable` implementation)must be created, compiled, and deployed on the cluster-side node. Note that the actual execution of the task is performed by the Java `Invocable` implementation and not the .NET `IInvocable` implementation.

See **Configuring a POF Context (Section 4)** for additional details.

11 Special Considerations Regarding Windows Forms Applications

One of the features of the `INamedCache` interface is the ability to add cache listeners that receive events emitted by a cache as its contents change. These events are sent from the server and dispatched to registered listeners by a background thread.

The .NET Single-Threaded Apartment model prohibits windows form controls created by one thread from being updated by another thread. If one or more controls should be updated as a result of an event notification, you must ensure that any event handling code that needs to run as a response to a cache event is executed on the UI thread. The `WindowsFormsCacheListener` helper class allows end users to ignore this fact and to handle Coherence cache events (which are always raised by a background thread) as if they were raised by the UI thread. This class will ensure that the call is properly marshalled and executed on the UI thread.

Here is the sample of using this class:

```
public partial class ContactInfoForm : Form
{
    ...
    listener = new WindowsFormsCacheListener(this);
    listener.EntryInserted += new CacheEventHandler(AddRow);
    listener.EntryUpdated  += new CacheEventHandler(UpdateRow);
    listener.EntryDeleted  += new CacheEventHandler(DeleteRow);
    ...
    cache.AddCacheListener(listener);
    ...
}
```

`The` AddRow, `UpdateRow` and `DeleteRow` methods are called in response to a cache event:

```
private void AddRow(object sender, CacheEventArgs args)
{
...
}

private void UpdateRow(object sender, CacheEventArgs args)
{
...
}

private void DeleteRow(object sender, CacheEventArgs args)
{
...
}
```

The `CacheEventArgs` parameter encapsulates the `IObservableCache` instance that raised the cache event; the `CacheEventType` that occurred; and the `Key`, `NewValue` and `OldValue` of the cached entry.

12 Special Considerations Regarding Web Applications

By default, session-state values and information are stored in memory within the ASP.NET process. ASP.NET also provides session-state providers that allow you to use a session-state server that keeps session data in a separate process, or you can persist session state data to a SQL database. However, with ASP.NET 2.0, you can create custom session-state providers that allow you to customize how session-state data is stored in your ASP.NET applications.

Coherence for .NET includes a custom `SessionStateStoreProvider` implementation that uses a Coherence cache to store session state. This makes Coherence for .NET the best solution for any large ASP.NET application running within a web farm. Other options in this scenario are to use the `StateServer`, which introduces a single point of failure for the whole web farm, or to use the `SqlServerStateProvider`, which theoretically can be clustered, but is extremely slow and scales only to a certain point. Also, unlike both `StateServer` and `SqlServerStateProvider`, the `CoherenceSessionProvider` supports `Session.End` event through cache events - only the `InProc` one supports this, but it cannot be used in a web farm environment.

The only requirement of the `CoherenceSessionStore` is that all objects stored in the session must be serializable (.NET serializable, not POF). This same requirement applies to both out-of-proc session stores provided by Microsoft, so modifying any existing ASP.NET 2.0 application that uses `StateServer` or `SqlServerStateProvider` to use the `CoherenceSessionStore` is as simple as adding the following to the Web.config file:

```
<sessionState mode="Custom" customProvider="CoherenceSessionProvider" timeout="20">
  <providers>
    <add name="CoherenceSessionProvider" type="Tangosol.Web.CoherenceSessionStore, Coherence"
  </providers>
</sessionState>
```

Note that no code changes are required within the application.

If your web application uses Coherence for .NET (either directly, via the `CoherenceSessionProvider`, or both), you must remember to call `CacheFactory.shutdown()` when your application terminates. To make this easier, Coherence for .NET includes an HTTP module that will automatically call `CacheFactory.shutdown()` when your web application exits. To use the `CoherenceShutdownModule` simply include it in your Web.config file:

```
<httpModules>
  <add name="CoherenceShutdown" type="Tangosol.Web.CoherenceShutdownModule, Coherence"/>
</httpModules>
```