BEA Tuxedo®

Using the Tuxedo .NET Workstation Client

Version 9.1
Document Revised: May 16, 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Creating Tuxedo .NET Workstation Client Applications

Creating Tuxedo .NET Workstation Client Applications

This topic includes the following sections:

Overview

For Microsoft .NET programmers, Tuxedo .NET Workstation Client is a facilitating tool that will help to efficiently develop Tuxedo .NET Workstation Client applications. Besides providing a set of Object Oriented (OO) interfaces to .NET programmers, this tool allows you to design and write code in OO styles.

For Tuxedo programmers, the Tuxedo .NET Workstation Client inherits most ATMI function invocation behavior which makes it easier to understand and use .NET Client classes to write applications. Because the Tuxedo .NET Workstation Client is published as a .NET assembly, it also leverages the benefit of .NET Framework. It can be used with many .NET programming languages (for example, C#, J#, VB .NET, and ASP.NET).

Note: The Tuxedo Workstation Client has been tested with and officially supports Microsoft Framework 1.1

The Tuxedo .NET Workstation Client enables you to write Tuxedo client applications using .NET programming languages to access Tuxedo services. It also provides connectivity between .NET workstation applications and Tuxedo services.

The Tuxedo .NET Workstation Client contains the following components:

- A wrapper assembly: `libwsclient.dll`

  This Microsoft .NET Framework `.dll` assembly wraps Tuxedo ATMI and FML functions for developing Tuxedo .NET workstation clients.

- A set of utilities: `viewcs`, `viewcs32`; `mkfldcs`, `mkfldcs32`; and `buildnetclient`

  These executable utilities help to develop C# code using Tuxedo VIEW/VIEW32 and FML/FML32 typed buffer and compile C# code to Tuxedo .NET Workstation Client executable assemblies. For more information, see `viewcs, viewcs32(1)`, `mkfldcs, mkfldcs32(1)`, `buildnetclient(1)`.

- Sample applications: callapp, fmlviewapp, and unsolapp

  These three samples explain how to create Tuxedo .NET Workstation Client application using C#. See Tuxedo .NET Workstation Client Samples.

Limitations

The Tuxedo .NET Workstation Client has the following limitations:

- It exclusively supports developing Tuxedo workstation clients with .NET languages. It does not support developing Tuxedo native clients or Tuxedo servers in a .NET environment.
- New Tuxedo 9.0 ATMI functions are not included in the Tuxedo .NET Workstation Client package (for example, `tpxmltofml32(3)` and `tpfmltoxml32(3)`).
- Tx Transaction interfaces (for example: `tx_open()`, `tx_close()`, and `tx_begin()`), are not supported by Tuxedo .NET Workstation Client. Only Tuxedo TP transaction functions can be used.
- Tuxedo .NET workstation Client implementation requires ATMI client functionality. During the Tuxedo installation, the *Full Install Set* and *.NET Client Install Set* automatically include the *ATMI Client Install Set*.

  For more Tuxedo .NET Client installation and Tuxedo install set information, see *Installing the BEA Tuxedo System*.

- Microsoft .NET Framework 1.1 must be installed in order to use the Tuxedo .NET Workstation Client. Microsoft .NET Framework is not bundled with the Tuxedo .NET Workstation Client package.

  To download Microsoft .NET Framework 1.1 and for more Microsoft .NET Framework information, see Microsoft's .NET Developer Center.

How Tuxedo .NET Workstation Client Works

The Tuxedo .NET Workstation Client works as an intermediate layer or wrapper between your .NET applications and underlying Tuxedo workstation shared libraries (`libwsc.dll`, `libengine.dll`, and `libfml.dll`). Figure 1 illustrates how the Tuxedo .NET Workstation client works.

The . NET assembly `libwscdnet.dll` contains the wrapper API classes for Tuxedo .NET Workstation Client and implements a set of object-oriented-styled interfaces that wrap around Tuxedo ATMI functions and FML functions.

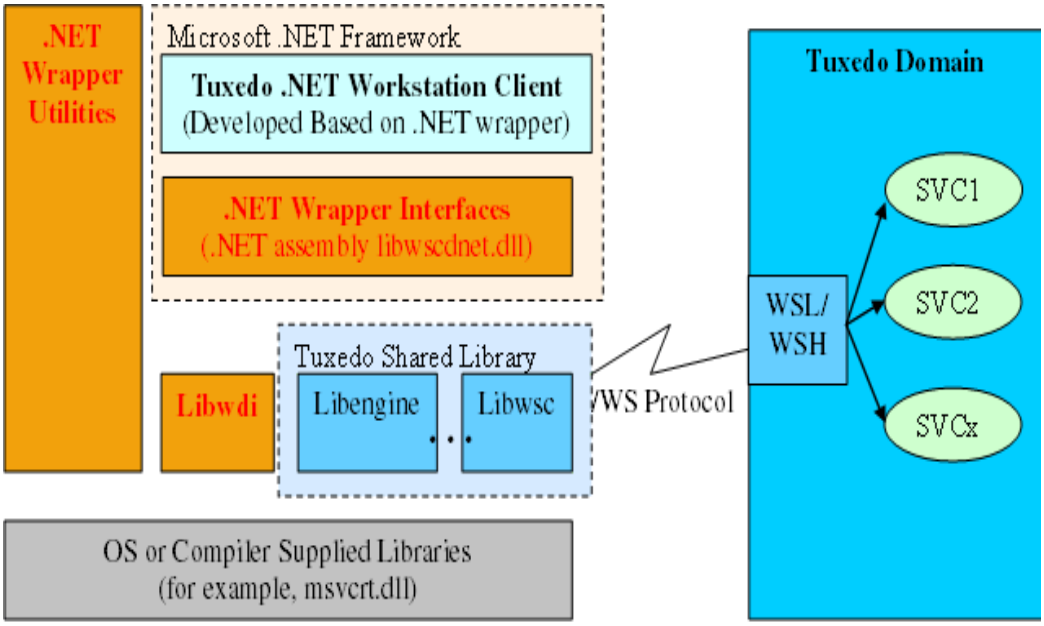
`buildnetclient` references `libwscdnet.dll` in order to build Tuxedo workstation clients written in . NET programming languages. It targets the Common Language Runtime (CLR) environment, and is invoked by the assemblies (for example, client executables, libraries) depending on it at runtime.

The win32 shared library, `libwdi.dll`, implements platform specific functions that `libwscdnet.dll` uses.

Note: `libwdi.dll` is only required by `libwscdnet.dll` at runtime and is not necessary when `buildnetclient` builds the application source files into .NET executables.

This library is reserved for future use if Microsoft's .NET Framework is ported to platforms other than the Microsoft Windows family.

Figure 1 How Tuxedo .NET Workstation Client Works



Microsoft .NET Framework

The Tuxedo .NET Workstation Client requires Microsoft .NET Framework 1.1 SDK installation on your system. The BEA Tuxedo installer program automatically checks to see if .NET Framework is installed or not. If installed, `libwscdnet.dll` is automatically registered in the .NET Framework global assembly cache.

If .NET Framework is not installed, you must install it. You can download the .NET Framework from Microsoft's .NET Developer Center. After you have installed .NET Framework, manually registering `libwscdnet.dll` in the global assembly cache is highly recommended.

To manually register/unregister `libwscdnet.dll` in the global assembly cache, you must do the following steps:

Register libwscdnet.dll Using GUI Configuration

1. Launch the . NET GUI configuration program. On your Windows desktop, click *Start/All Programs/Administrative Tools/Microsoft .NET Framework*

2. Right click *Assembly Cache*. Click *Add*
3. Select `libwscdnet.dll` from the `%TUXDIR%\bin` directory. Click *Open*. `libwscdnet.dll` is added to the Assembly Cache list.

Unregister libwscdnet.dll Using GUI Configuration

1. Launch the . NET GUI configuration program. On your Windows desktop, click **Start→ All Programs→ Administrative Tools→ Microsoft .NET Framework** and select **Assembly Cache**.
2. In the **Assembly Cache** list, right click `libwscdnet.dll` and click **Delete**.

Register/Unregister Using Command Line

You can also register/un-register `libwscdnet.dll` from the command line.

To register enter: `%WINDIR%\Microsoft.NET\Framework\v1.1.4322\gacutil.exe /i %TUXDIR%\bin\libwscdnet.dll.`

To unregister enter: `%WINDIR%\Microsoft.NET\Framework\v1.1.4322\gacutil.exe /u libwscdnet.dll.`

Developing Tuxedo .NET Workstation Client Applications

Programmers developing Tuxedo .NET Workstation Client applications must:

1. Use the .NET wrapper classes/interfaces
2. Set up a Tuxedo workstation client environment
3. Access Tuxedo services via Tuxedo /WS protocol.

The Tuxedo .NET Workstation Client provides development utilities that can aid programmers using Tuxedo FML/VIEW typed buffer and building .NET executable files. See Using FML/FML32 Typed Buffers and Using VIEW/VIEW32 Typed Buffers.

Programming Tuxedo .NET Workstation Clients

Main changes in Tuxedo .NET Workstation Client interface (compared to Tuxedo ATMI and FML C functions), are as follows:

- `Class AppContext` is used to organize almost all ATMI C functions.

- A Tuxedo transaction is encapsulated as a class. All transaction related functions are defined as methods of `Class Transaction`, (for example, `tpbegin()`, `tpcommit()`, and so on).
- Exception classes control error handling
- Tuxedo typed buffer encapsulation is handled using class `TypedBuffer` and its derived classes. See Using Typed Buffers.

Tuxedo .NET Workstation Client Namespaces

Tuxedo .NET Workstation Client namespaces are divided into two categories. The first category includes two namespaces, `Bea.Tuxedo.ATMI` and `Bea.Tuxedo.FML` that bundles ATMI and FML wrapper classes.

The second category uses the `Bea.Tuxedo.Autogen namespace` to bundle all auto-generated .NET classes using .NET Client utilities.

These namespaces include *all the classes and structures* related to the functions listed in the Tuxedo .NET Workstation Client API Reference.

Using AppContext Class

The `AppContext` class is a key class used to perform Tuxedo service access functions. `AppContext` leverages the OO programming style in a multi-contexted client application.

Note: For more *multi-context* information, see Programming a Multithreaded and Multicontexted ATMI Application.

Most Tuxedo ATMI C functions (for example, `tpcall()`, and `tpnotify()`), are defined as AppContext class methods. Creating an `AppContext` class instance is a key component in connecting to a Tuxedo domain and call services provided by that Tuxedo domain.

In a multi-contexted application written in C or COBOL, programmers typically have to switch between different Tuxedo context using two ATMI functions, `tpgetctxt()` and `tpsetctxt()`. This is not required using the Tuxedo .NET Workstation Client. Creating a class `AppContext` instance also creates specific Tuxedo context instance.

Operations on a particular `AppContext` will not impact other `AppContext` instances. You can develop multi-context applications and easily switch between them.

To create a Tuxedo context instance you need to invoke the static class method, `AppContext.tpinit(TPINIT)`, instead of the class constructor.

Note: Tuxedo context instances are not destroyed automatically. You must invoke `AppContext.tpterm()` before a Tuxedo context instance is destroyed, otherwise you may encounter the following:

- The garbage collector (gc) may destroy `AppContext` class instances without terminating the Tuxedo context session.
- The client and WSH connection remains live until it times-out.

The following sample shows how to connect to a *single context* Tuxedo domain.

Listing 1 C# Code Sample: Connecting to a Single Context Client

```
……

TypedTPINIT  tpinfo = new TypedTPINIT();

AppContext ctx1 = AppContext.tpinit(tpinfo);   // connect to Tuxedo domain

……

ctx1.tpterm();          // disconnect from Tuxedo domain
```

The following sample shows how to connect to a *multi-context* Tuxedo domain .

Listing 2 C# Code Sample: Connecting to a Multi-Context Client

```
……

TypedTPINIT  tpinfo = new TypedTPINIT();

tpinfo.flags = TypedTPINIT.TPMULTICONTEXTS;   // set multi context flag

// connect to the first Tuxedo domain

AppContext ctx1 = AppContext.tpinit(tpinfo);
```

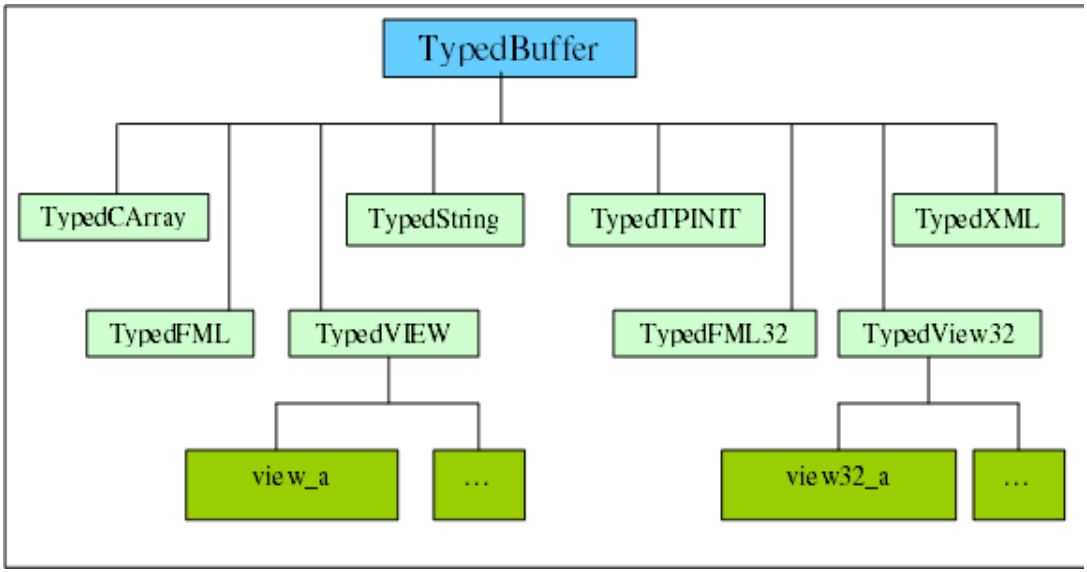
```
Utils.tuxputenv("WSNADDR=//10.1.1.5:1001");

// connect to the second Tuxedo domain

AppContext ctx2 = AppContext.tpinit(tpinfo);

……

ctx1.tpterm();            // disconnect from the first Tuxedo domain

ctx2.tpterm();            // disconnect from the second Tuxedo domain
```

Using Typed Buffers

The Tuxedo .NET Workstation Client supports the following built-in BEA Tuxedo buffer types: `FML`, `FML32`, `VIEW`, `VIEW32`, `CARRAY`, and `STRING`. Figure 2 provides an illustration of the Tuxedo .NET Workstation Client typed buffer class hierarchy.

The Tuxedo .NET Workstation Client class `TypedBuffer` is the base class of all concrete Tuxedo buffer types and provides some low level functions to all derived classes. Class `TypedBuffer` is an *abstract* class and cannot be used to create instances.

Figure 2 Tuxedo .NET Workstation Client Typed Buffer Class Hierarchy



Using STRING Typed Buffers

The Tuxedo .NET Workstation Client uses class `TypedString` to define `STRING` typed buffer characters. `TypedString` instances can be used directly to communicate with `AppContext` methods such as `tpcall()`. See the following example.

Listing 3 Using TypedString Class (C# code example)

```
.....

TypedString  snd_str = new TypedString ("Hello World");

TypedString rcv_str = new TypedString(1000);

AppContext ctx = AppContext.tpinit(null);

.....

ctx.tpcall("TOUPPER", snd_str, rcv_str, 0);
```

```
.....

ctx.tpterm();

.....
```

Using FML/FML32 Typed Buffers

The Tuxedo .NET Workstation Client uses class `TypedFML`/`TypedFML32` to define most FML C functions. You should do the following steps to develop Tuxedo .NET applications using FML typed buffers:

1. Define FML field table files.

   Compile field table files into C# source files using the `mkfldcs` Tuxedo .NET Workstation Client utility. The generated C# files contain public classes including definitions of every FML field ID defined in the field table files. See also `mkfldcs(1)`

2. Write your .NET application.

   Use `TypeFML` class methods to create and access FML data.

For more FML typed buffer programming information, see *Programming a Tuxedo ATMI Application Using FML*.

Listing 4 FML Auto-Generated Code Using mkfldcs (C# code example)

```
using Bea.Tuxedo.FML;

namespace Bea.Tuxedo.Autogen {

public class fnext_flds {

public static readonly FLDID F_short = 110;// number: 110 type: short

public static readonly FLDID F_view32 = 369098863;// number: 111 type:
view32
```

```
    public static readonly FLDID F_double = 134217840;// number: 112 type:
double
    public static readonly FLDID F_ptr = 301990001;// number: 113 type:
ptr

}

}      //     namespace Bea.Tuxedo.Autogen
```

Listing 5 Using TypedFML Class (C# code example)

```
……

TypedFML  fmlbuf = new TypedFML(2048);

short s = 123;

fmlbuf.Fadd(fnext_flds.F_short,  s);

……

fmlbuf.Resize(3000);

……

fmlbuf.Dispose();

……
```

Using VIEW/VIEW32 Typed Buffers

The Tuxedo .NET Workstation Client uses class `TypedVIEW` to create and access VIEW/VIEW32 data. You should do the following steps to develop Tuxedo .NET Workstation Client applications using VIEW/VIEW32 typed buffers:

1. Define the VIEW definition file (.v).
2. Use the Tuxedo .NET Workstation Client `viewcs` utility to compile the VIEW definition file into a VIEW binary file (.VV). For more information, see `viewc(1)`, `viewcs(1)`.
3. Use the Tuxedo .NET Workstation Client `viewcs` utility to generate class `TypedVIEW` derived definition C# code and corresponding `.dll` library (if necessary) from the View binary file.
4. Use class `TypedVIEW` to write your .NET application.

   Using class `TypedVIEW` provides you with two options:

   – **Option 1: No Environment Variables**

     This is the most common usage of `TypedVIEW`.

     Use the `viewcs` utility to generate derived class `TypedVIEW` definition C# code from the xxx.VV file, then compile the C# code into an `.exe` file. No additional environment variables are required.

     See the following example:

     ```
     viewcs(32) view1.VV view2.VV
     buildnetclient -o simpapp.exe simpapp.cs view1.cs view2.cs
     ```

   – **Option 2: Use .NET Assembly Environment Variables**

     You can use the `viewcs` utility along with .NET assembly environment variables to generate `.dll` libraries. The .NET assembly environment variables `ASSFILES`, `ASSDIR` (`ASSFILES32`, `ASSDIR32` for view32) must be set accordingly in order to view `viewcs`-generated `.dll` libraries.

     Note: `TypedView` must link to `.dll` libraries instead of C# code in the .NET environment. This is because it compiles the class type into `.dll` libraries or `.exe` files. If the definition is compiled into both `.dll` libraries and `.exe` files, the output binaries for these two files are not the same.

     Using these environment variables, `.dll` libraries can be generated automatically or manually:

     Automatic `viewcs`-generated `.dll` libraries

     This method may be used when many xxx.VV files exist. To simplify management of `TypedVIEW` C# code, these xxx.VV files can be compiled into a `.dll` library.

     Use the `viewcs` utility to generate derived class `TypedVIEW` definition C# code and corresponding `.dll` library from the xxx.VV files. Manually register the

libwscdnet.dll assembly, and then compile your client application using the .dll library.

See the following example:

```
viewcs(32) view.dll view1.VV view2.VV
gacutil.exe /i view.dll
buildnetclient -o simpapp.exe simpapp.cs view.dll
set ASSFILES(32)=view.dll
set ASSDIR(32)=%APDIR%
```

Manual-generated .dll Libraries

In certain integrated programming environments (for example, VB .NET, and ASP.NET). the framework provides the executing environment. Client applications are integrated as .dll files. In this case it is best to manually generate .dll libraries.

Use the viewcs utility to generate derived class TypedVIEW definition C# code from the xxx.VV file, then compile the C# code into an application .dll.

The .NET assembly environment variables ASSFILES, ASSDIR (ASSFILES32, ASSDIR32 for view32) must be set to application .dll libraries and directories that have TypedVIEW defined.

See the following example:

```
viewcs(32) view1.VV view2.VV
csc /t:library /out:simpapp.dll /r:%TUXDIR%\bin\libwscdnet.dll
simpapp.cs
view1.cs view2.cs
set ASSFILES(32)=simpapp.dll
set ASSDIR(32)=%APDIR%
```

The Typed Buffer Samples file (included in the Tuxedo . NET Workstation Client package) demonstrates how to use FML and VIEW typed buffers.

Programming with Visual Basic (VB)

One benefit of the .NET Framework environment is *language integration*. Once a .NET assembly is generated, you can use any .NET supported language to develop applications using that assembly. Accordingly, you can also use J#, VB, C++ or other .NET supported languages to develop Tuxedo .NET Workstation Client applications.

The following is a VB language code example:

Listing 6 Visual Basic .NET Code Example

```
Imports System
Imports Bea.Tuxedo.ATMI

Module Main
       Sub Main()

              Dim sndstr, rcvstr As TypedString
              Dim ac As AppContext
              Dim info As TypedTPINIT

              info = New TypedTPINIT()
              info.cltname = "vb client"

              Try
                  ac = AppContext.tpinit(info)

                  sndstr = New TypedString("hello world")
                  rcvstr = new TypedString(1000)

                  ac.tpcall("TOUPPER", sndstr, rcvstr, 0)

                  Console.WriteLine("rcvstr = {0}"
                  rcvstr.GetString(0,1000))

                  ac.tpterm()

              Catch e as ApplicationException

                 Console.WriteLine("Got Exception = {0}", e)

              End Try

End Sub
```

End Module

Building .NET Workstation Clients

The `buildnetclient` utility is provided to help compile C# source files into a .NET executable files. See also `buildnetclient(1)`. The following is a `buildnetclient` syntax example:

```
buildnetclient -v -o simpapp.exe simpapp.cs
```

Managing Errors

The error code return mechanism used with Tuxedo ATMI C and FML C functions is replaced with an exception mechanism in the Tuxedo .NET Workstation Client. You can use the `try` statement to handle errors using the Tuxedo .NET Workstation Client. Errors are defined into two categories: `TPException` and `FException`.

Listing 7 Exception Handling (C# Code Example)

```
……

try {

……

      TypedTPINIT  tpinfo = new TypedTPINIT();

      AppContext ctx1 = AppContext.tpinit(tpinfo);   // connect to Tuxedo
domain

……

      ctx1.tpterm();          // disconnect from Tuxedo domain

       ……
```

```
} catch (ApplicationException e) {

      Console.WriteLine("******Error******, e = {0}", e);

}
.....
```

Tuxedo .NET Workstation Client Samples

Three sample applications are bundled with Tuxedo .NET Workstation Client package:

- Basic Sample

  Describes how to develop Tuxedo .NET Workstation Client applications

- Typed Buffer Sample

  Demonstrates `FML/VIEW` typed buffer usage in Tuxedo .NET Workstation Client applications

- Unsolicited Message Sample

  Demonstrates how to register unsolicited message handler in Tuxedo .NET Workstation Client applications

You must do the following steps to access the sample applications:

1. Read the `readme.nt` file in each sample application directory.
2. Run `setenv.cmd` to set Tuxedo environment variable.
3. Run `nmake -f xxx.nt` to build the Tuxedo .NET Workstation Client application, Tuxedo server program and Tuxedo `TUXCONFIG` file.
4. Run `tmboot -y` to start Tuxedo application.
5. Run Tuxedo .NET Workstation Client application.

See Also

- viewc, viewc32(1); viewcs, viewcs32(1)
- mkfldhdr, mkfldhdr32(1); mkfldcs, mkfldcs32(1)

- Programming BEA Tuxedo ATMI Applications Using C
- BEA Tuxedo ATMI C Function Reference
- BEA Tuxedo ATMI FML Function Reference
- File Formats, Data Descriptions, MIBs, and System Processes Reference