CA Plex C# Best Practices

Example Document

Creating packages and code libraries and managing C# source code and application artifacts

CA Plex version 6.1

Created by:

In collaboration with IIDEA Solutions and CCH, and assistance from CA.

Published with the kind permission of the South Carolina Judicial Department

.

Contents

1. Introduction 3
2. Use of assemblies and modules when creating code libraries 3
3. Creating and using assemblies 4
4. Considerations when packaging your model 5
5. Software required on the build server 6
6. Modelling C# Code Libraries in Plex 7
7. Setting up the build server and installing Cruise Control 10
8. Development life cycle 20
Appendix 23

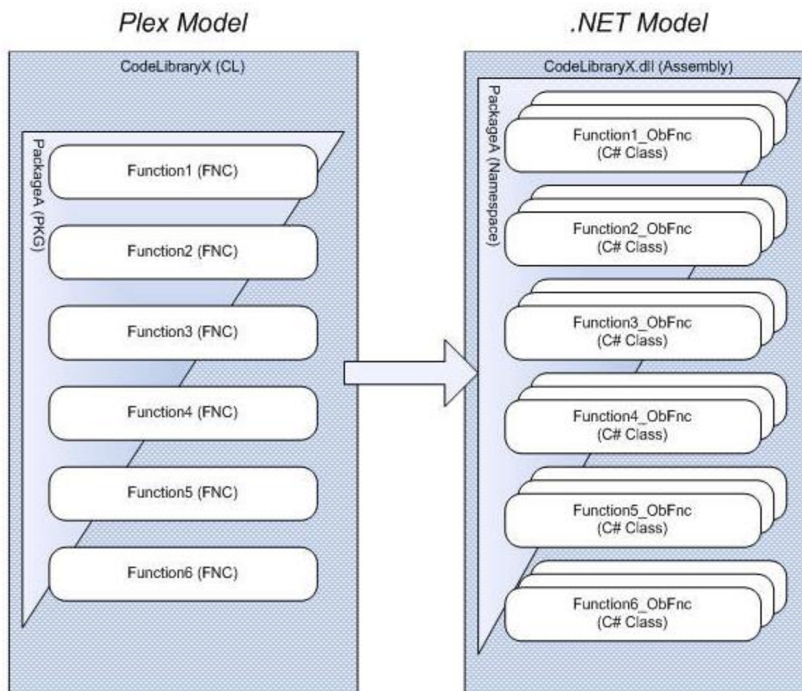
1. Introduction

This document is intended to serve as a practical guide to the use of the C# in a CA Plex development organization. It contains basic explanations of the concepts behind C# / .NET development, and provides guidance on usable mechanisms to implement different practices. It is not intended to be a set of standards, but can be used as guidance for an organization to incorporate .NET packaging into their CA Plex development standards.

This document was created in collaboration between ADC Austin and CA reference customers, and represents practices that are in production use. In the case where there are alternatives present or the CA Plex manual is unclear, **<u>recommendations in bold underline.</u>**

2. Use of assemblies and modules when creating code libraries

The following diagram shows the basic structure of assemblies, packages (namespaces), and classes as they related to Plex model. For more detailed information, please consult the Plex help topic Chapter 18 Packaging and deploying applications.



CA Plex allows you to create 2 types of code libraries:

i. **Assembly** – These are the building blocks of the .NET framework and contain manifest data about its contents. Assemblies can be made up of packages or other code libraries.

ii. **Module** – Usually are contained within an assembly, Modules do not contain any manifest data as that is stored in the assembly.

Modules are used for the following situations:

i. **Multi language assembly** - If the assembly contains source files with different languages, they have to be split out into different modules within the assembly.
ii. **Small download footprint** - If the assembly is downloaded at runtime to an HTTP site, it is recommended that the code is divided up into the code that will be required at login and accessing the application, and then into logical areas of functionality. This makes the download much faster as code is only downloaded when it is needed.
iii. **Shared code.** – If you have code that is shared by multiple assemblies you can place those functions in a module and attach it to multiple assemblies. The literature suggests care is taken when using this approach as any changes made to that code will mean that the entire assembly will have to be created and shipped. Therefore it is recommended that shared code be placed in its own assembly.

Given that only C# code is managed in .NET (and currently C# code is only created on the server on PLEX and not downloaded to the client), **the use of modules is not recommended**.

From the Plex manual note the following recommendation. Although default code libraries should be used in certain circumstances for unit testing, they should not be used for production implementation due the problems with maintainability.

> *Plex C# functions can be built outside of the Code Library modeling concept, using the Default Package and Code Library settings in the Generate and Build options. However, as a best practice for .NET application deployment, **it is recommended that Plex generated C# functions be arranged by Package and Code Library**. As with Java, the Package objects define the namespace boundaries into which the Function classes are defined. The Code Library object is the main compilation block for a Plex .NET application; and to allow maximum flexibility, can be created as either a module or an assembly.*

Another reason for the recommendation to package all functions is cross-model references – code will not be generated properly if defaulting techniques are used. **All functions should be assigned to an explicit package. It is recommended that the CA Package Tool be utilized for package creation.**

3. Creating and using assemblies

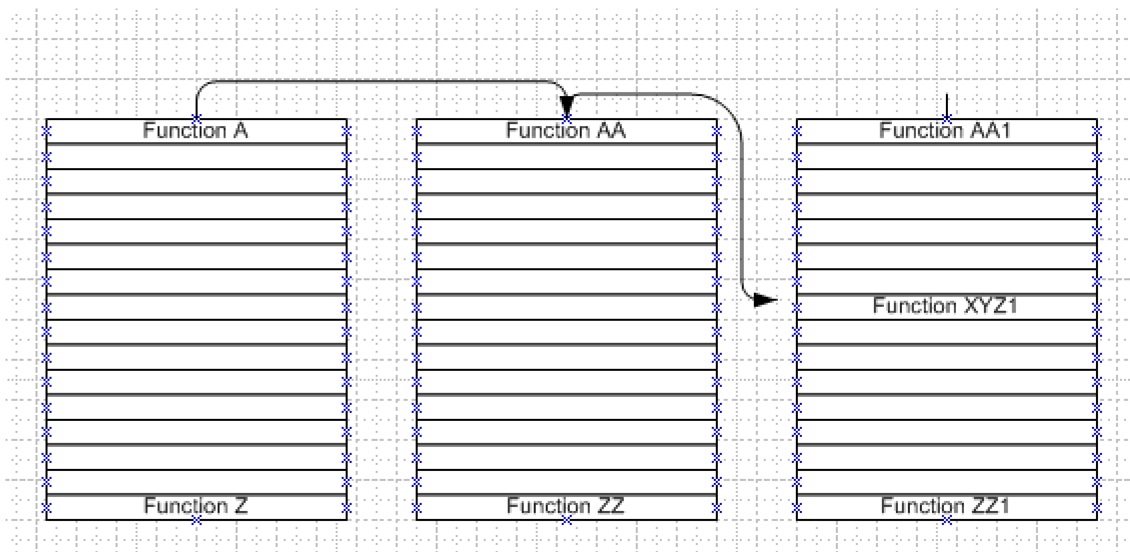
While having many assemblies has advantages in terms of maintenance and distribution, there are tradeoffs with the performance of the application. The more assemblies you have the less efficient your application will be. This is because when the application runs, it will open all the assemblies in the assembly list (note that the CA Plex manual is unclear on this topic, but this has been determined to be the case with the current Plex version). **Therefore it is recommended that fewer assemblies are used.** In most cases this does not mean that all functions should be put in a single assembly, as this would make maintenance difficult.

As part of creating this document, interviews were conducted with a large C#/PLEX developers, including a software house who has been using C# for 3 years. Their application has thousands

of Plex functions, but they have less than 10 assemblies. The key is having enough assemblies to make management straightforward while achieving acceptable performance at run time.

Performance can be improved at runtime by giving consideration to the order of your assemblies in the assembly list. An assembly list works much like a library list – at run time the application will look in each assembly, starting at the beginning of the list, until it finds the first instance of the function in needs.

Therefore when packaging your application it is recommended that consideration is given to the frequency that functions are called. **Functions that are called frequently should be packaged together and the resulting assembly should be placed first or high in the list.** Rarely used functions should also be packaged together and the assembly should be placed last in the list. The packages in the middle should be designed around function areas since it is more efficient for a function to call another function within the same assembly.



4. Considerations when packaging your model

Taking all the above into consideration and knowledge of the application, the list of assemblies would look something like this, **with our recommendation being somewhere between 3-10 assemblies**.

- Core functions called frequently (Date/Common routines, Reads to common tables)
- Subsystem A
- Subsystem B
- Subsystem Z
- Core functions called infrequently (updates to common tables)

There are several techniques for adding CA Plex functions to packages. It is recommended that an approach be taken that is easy to implement from a standards point of view, but that is also efficient. This is most commonly done by placing entire entities into packages, and to create a separate package or packages for unscoped functions (or scope to entities). This works well in most cases for dividing models into subsystems without burdening the developer. In some cases that may lead to

less efficient assemblies (perhaps some entities are read only in most cases), so depending on the performance packaging structure may be more complex.

This must be used with care. For example, you may choose to add read functions to one package and write functions to another. It should be noted that this should only be used in rare circumstance, if there are extreme performance concerns. **In almost all circumstances this is not recommended, the performance gains are slight compared to the significant extra overhead of modeling packages in this manner.**

It is recommended that a function be placed in only one package/namespace. Assemblies can contain more than one package, and assemblies can contain assemblies. This is not important from a performance standpoint (the number and structure of the final assembly list is the key), but may be helpful from an application understandability point of view. Regarding the use of stub functions (function call definitions in one model where the called function resides in another, unreferenced model), it is important to understand that stub packages that match the package in the called model must be created.

It is recommend that packages be used to model the overall application structure, and not be used as a version control technique (i.e. packaging a "patch" into a special package / assembly). Packaging in the version control technique leads to difficulties in building the application, and we believe will lead to errors and application problems. We did not interview any C# development shop using packages / assemblies in a version control manner, and were strongly advised against it.

On package and assembly names – **it is recommended to avoid embedded spaces and special characters**. Although .NET has one set of rules for naming that includes some special characters, we have found that other 3rd party applications and other languages like Java have more restrictive naming requirements so this is the safest approach.

5. Software required on the build server

When setting up a server to do the builds, you will need the following software installed.

i. **Cruise Control –** A tool for managing automated builds on the server. Download from http://sourceforge.net/projects/ccnet/files/
Select CruiseControl.NET-1.x.x-SPX-Setup.exe

ii. **Microsoft.NET Framework Version 2.0 and 3.5 –** You can download from: http://www.microsoft.com/downloads/details.aspx?FamilyID=0856EACB-4362-4B0D-8EDD-AAB15C5E04F5&displaylang=en
Or if you install CA PLEX v6.1 on the server there is an option to install the .net framework.

iii. **Microsoft.NET Framework Version 2.0 Software Development Kit (SDK) -** You can download from:
http://www.microsoft.com/downloads/details.aspx?FamilyID=fe6f2099-b7b4-4f47-a244-c96d69c35dec&displaylang=en

iv. **Subversion** – This is the repository. Download from http://www.visualsvn.com/server/

v. **Tortoise** – A good front end UI for managing and working with Subversion. Download from http://tortoisesvn.net/downloads

vi. **NANT** – Allows you to write scripts to manage the automated build process. Download from http://sourceforge.net/projects/nant/

vii. **IIS** – Microsoft's internet server. This is available from the XP Professional disc. This link tells you how to install IIS. http://www.webwizguide.com/kb/asp_tutorials/installing_iis_winXP_pro.asp

6. Modelling C# Code Libraries in Plex

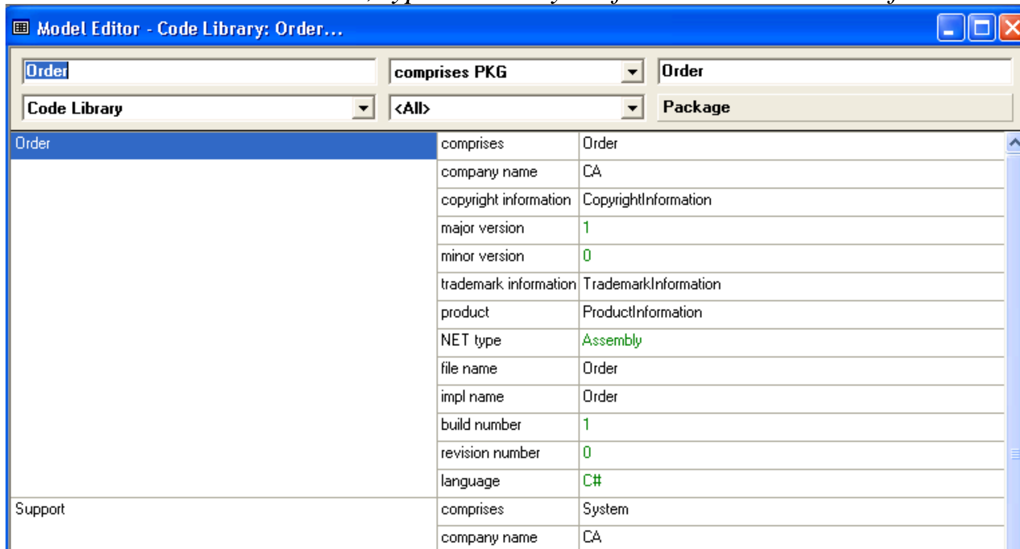
The next section shows the modeling of the C#/.NET application in Plex. We are using the supplied CA "SalesSystem" model typically located in the following directory:

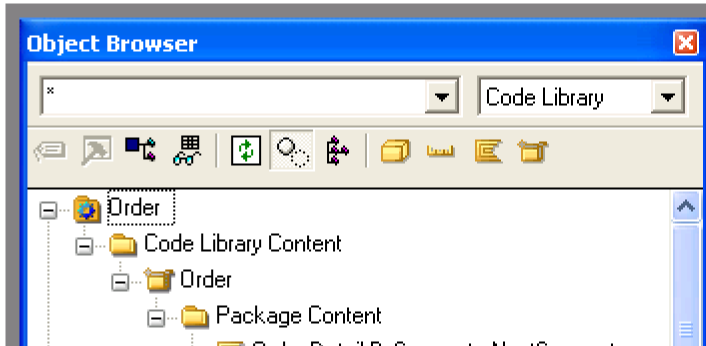
C:\Users\Public\Documents\CA\Plex\6.1\Samples\Dot NET Support and Code Libraries

Two types of CA Plex objects must be modeled, Code Libraries and Packages.

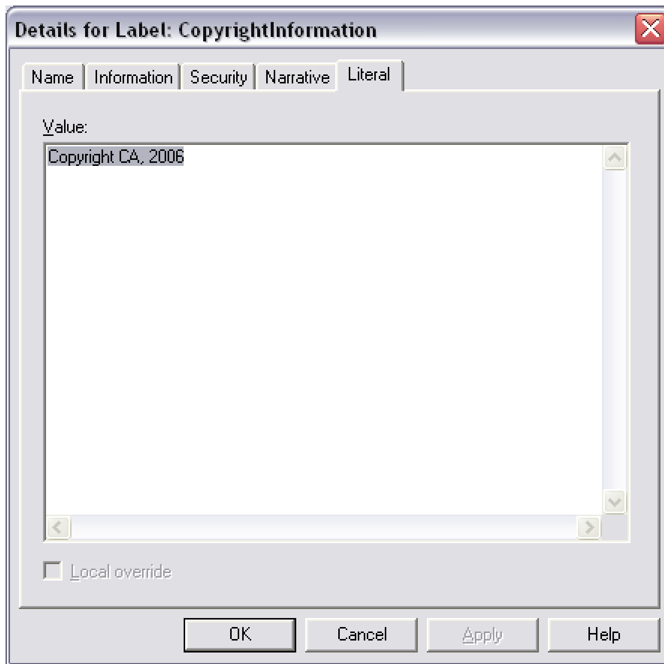
Code Libraries

There are two code libraries, type Assembly. Refer to the screen shot for an example of the triples



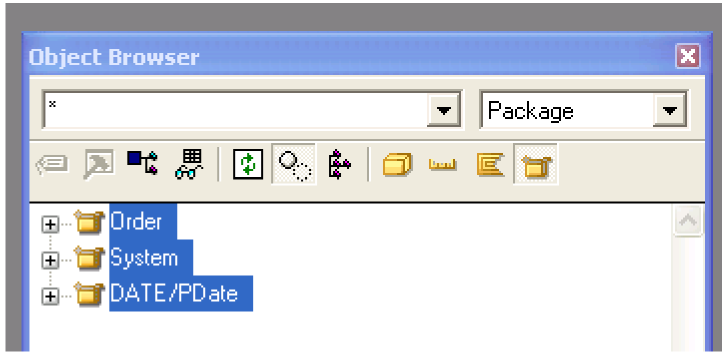


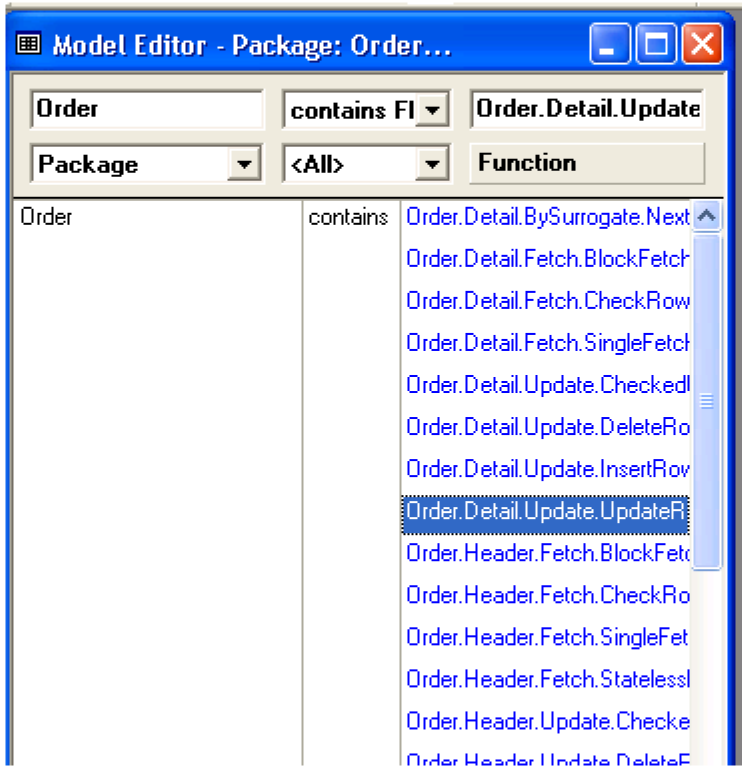
Note that some labels defined by the triples should have descriptive information entered into the label name value



Packages (Name Spaces)

There are two packages in the SalesSystem model, and one (PDate) as a library model. Functions are set up directly in the package. For large model, it is recommended to scope entities to the package for ease of maintenance.

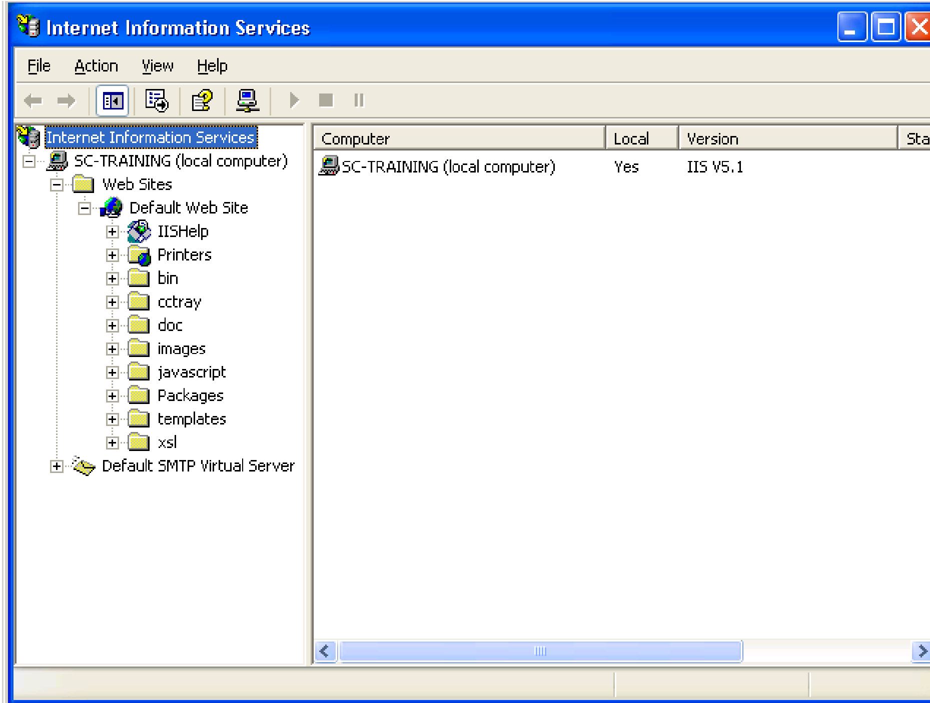




7. Setting up the build server and installing Cruise Control

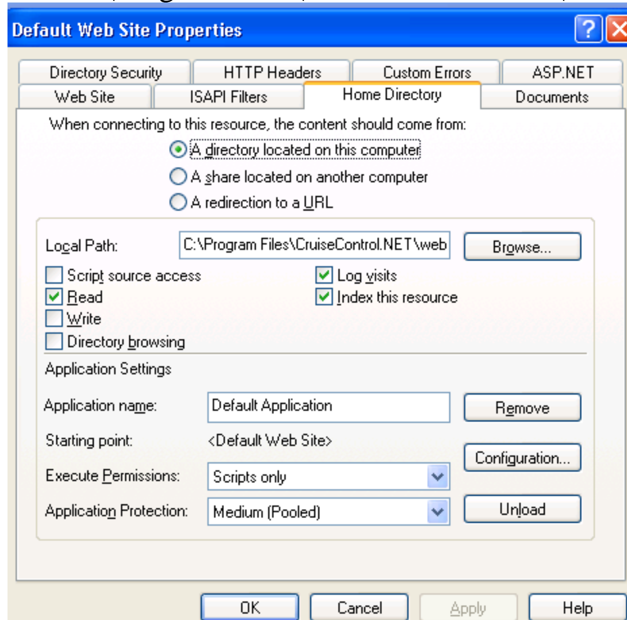
i. Configure a website on IIS for the CruiseControl.Net Dashbboard.

Go to *control panel* and in 'Classics View' select *Administrative Tools*. Open *Internet Information Services* and find the Default Website in the tree.

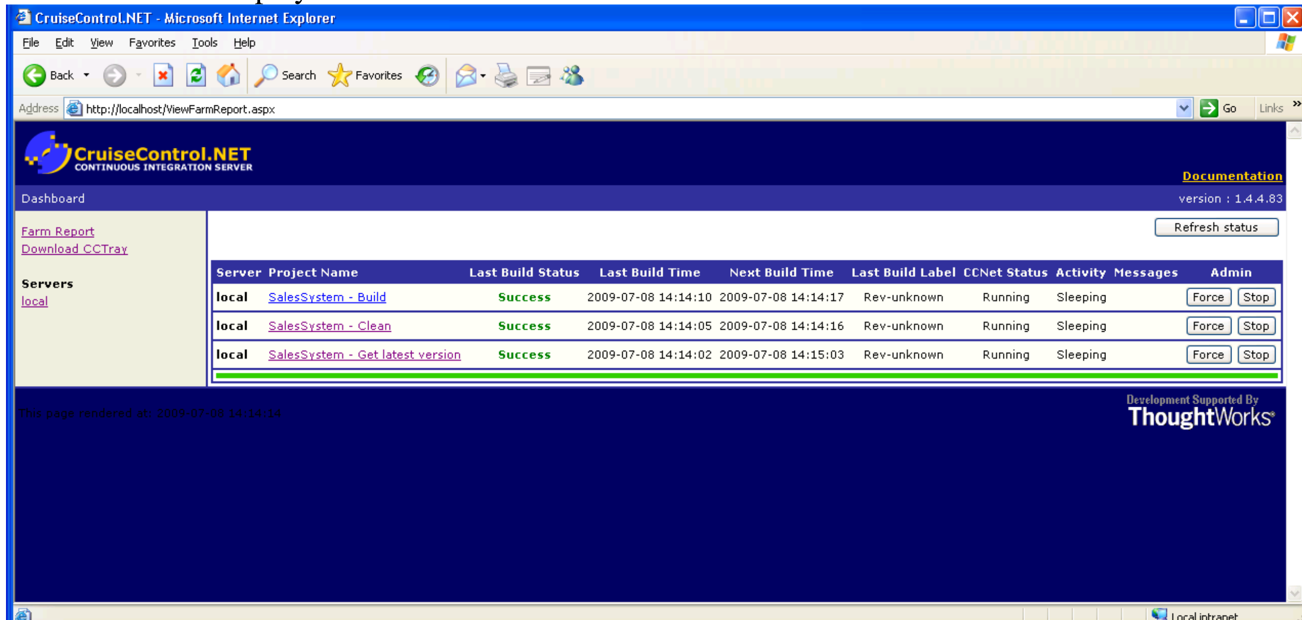


Select the Default Web Site. Right click and select *Properties*. Go to the Home Directory Tab. The local path will be C:\inetpub\wwwroot. Change the Local Path by prompting to the Cruise Control Dashboard. The default loacation is:

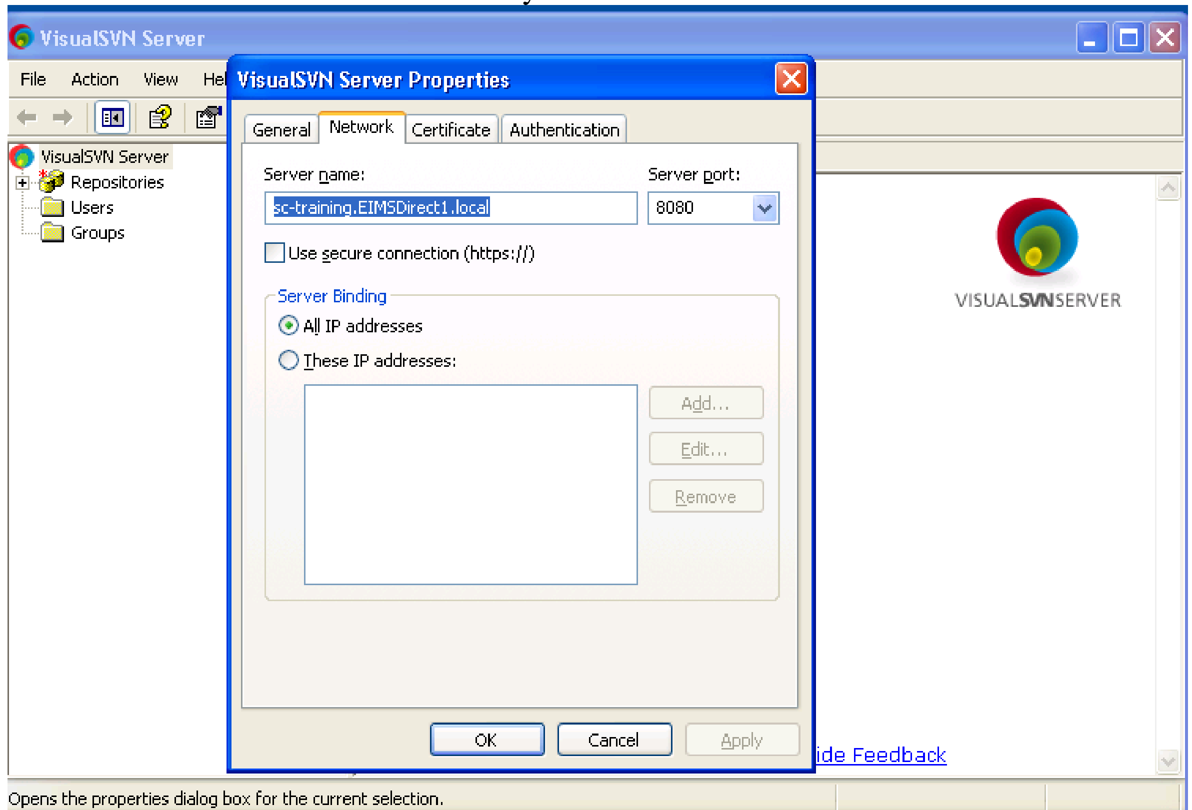
C:\Program Files\CruiseControl.NET\Webdashboard. Press *OK*.



Open Internet Explorer and type: http://localhost. The Cruise Control Dashboard will display



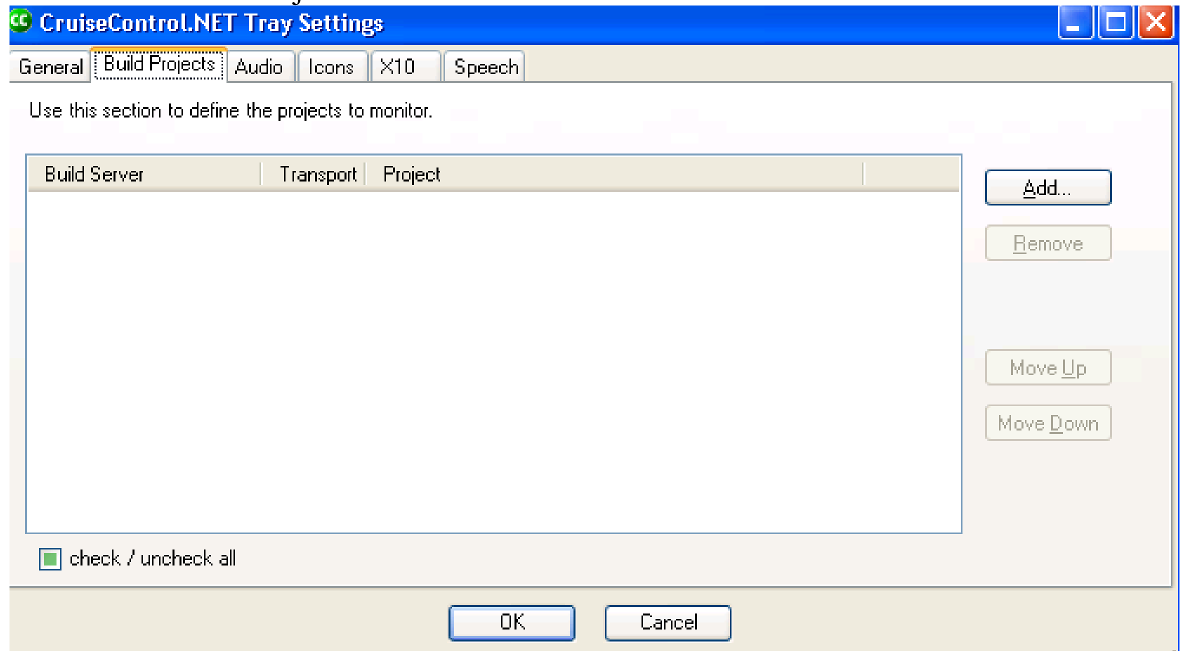
Note: In VisualSVN server properties, checkbox "Use secure connection (https://)" has to be unchecked in order to be reached by Cruise Control.NET



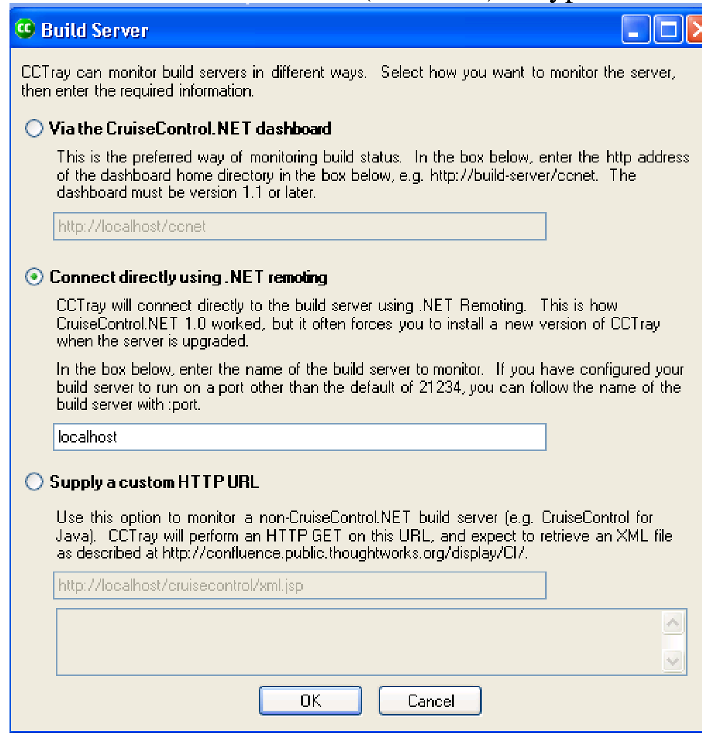
ii. Install CCTray

a. Click on Download CCTray link on the left panel of CC Dashboard.

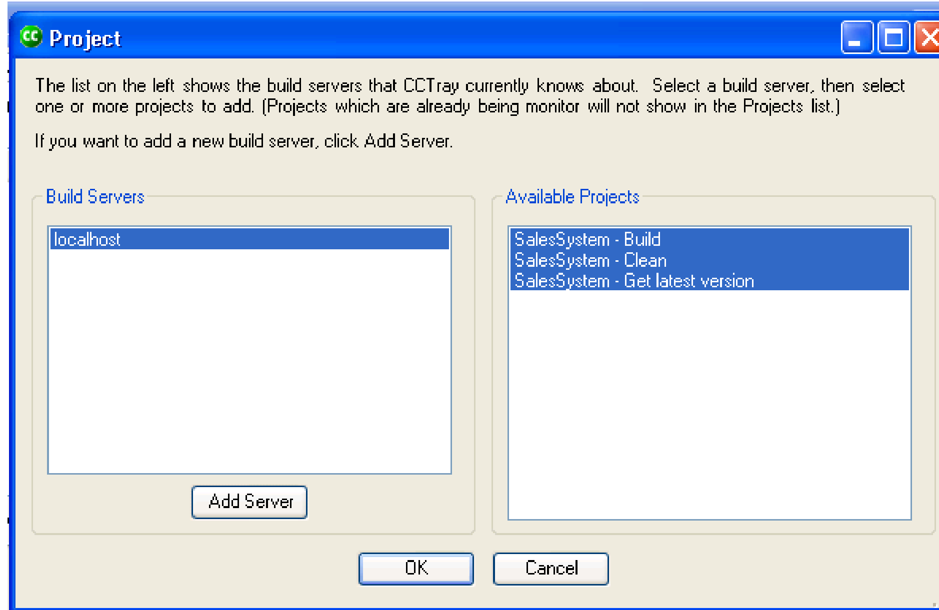
b. Save the file.
c. Execute the file saved and follow the wizard instructions.
d. Open the CCTray already installed
e. Select File → Settings on the top menu
f. Click on Build Projects tab and click on Add buton



g. Click on Add Server button
h. Select the option "Connect directly using .NET remoting"
i. Leave the default value (localhost) or type it



j. Click OK
k. Projects should be displayed in the right panel
l. Select the project to be added to CCTray and click on OK button

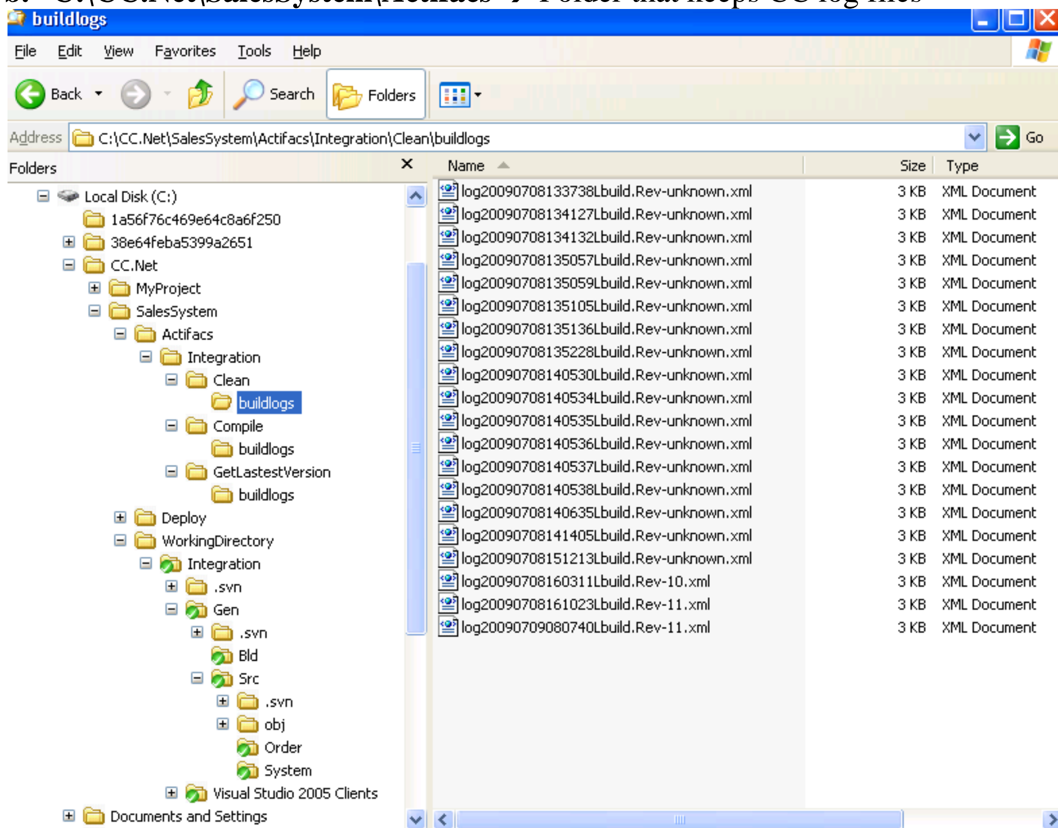


m. Click on OK button
n. Project selected will be displayed in CCTray

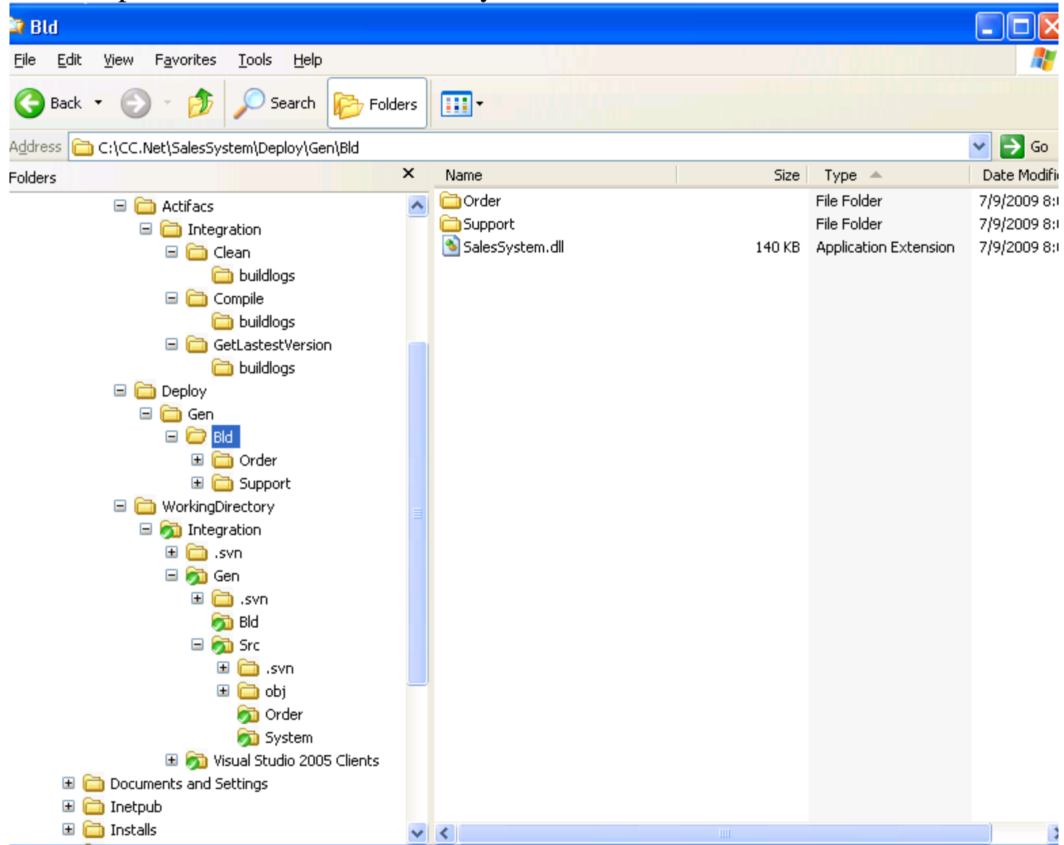
iii. Folders Structure

a. C:\CC.Net\SalesSystem → Root directory of the project. Here is the SalesSystem.Build file

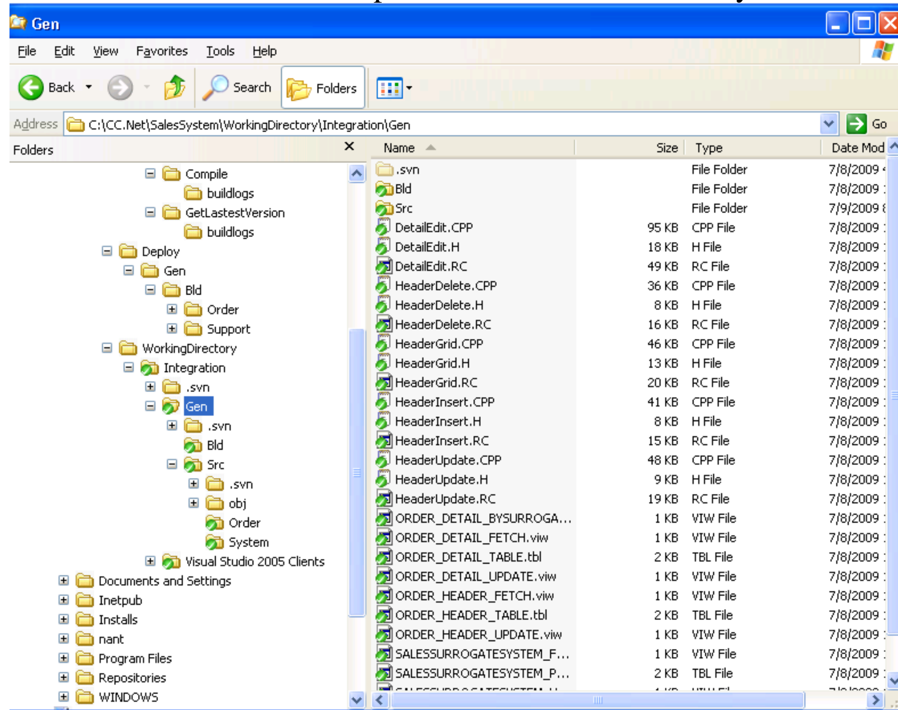
b. C:\CC.Net\SalesSystem\Actifacs → Folder that keeps CC log files



c. **C:\CC.Net\SalesSystem\Deploy →** Folder that keeps all the .dll files generated. This path is defined in the SalesSystem.build file



d. **C:\CC.Net\SalesSystem\WorkingDirectory →** Folder used to keep the latest version of the files. This path is defined in the SalesSystem.build file



iv. ccnet.config file. It's located in C:\Program Files\CruiseControl.NET\server\ folder and contains the definitions of the projects that will be displayed in the Cruise Control Dashboard. The most important tag are the follows:

a. **<project name="SalesSystem - Get latest version">**. Obtains the latest version of the files committed in repository.

   i. <trunkUrl>http://sc-training.eimsdirect1.local:8080/svn/SalesSystem/trunk</trunkUrl> is the path where the source code has been committed.

   ii. <username>&svnusername;</username> is the valid user to log into the repository.

   iii. <password>&svnpassword;</password> is the valid user to log into the repository.

   iv. <triggers>
   <intervalTrigger buildCondition="IfModificationExists" />
   </triggers>**.**

   Trigger blocks allow you to specify when CruiseControl.NET will start a new integration cycle.
   <intervalTrigger> is used to specify that an integration should be run periodically.
   IfModificationExists, means that **SalesSystem - Get latest version** project will only be triggered if modifications have been detected.

b. **<project name="SalesSystem - Clean">**. Deletes the folder C:\CC.Net\SalesSystem\Deploy\Gen\Bld

   i. <triggers>
   <projectTrigger project="SalesSystem - Get latest version" />
   </triggers>.

   <projectTrigger> is used to trigger a build when the specified dependent project has completed its build; so, in this case, **SalesSystem – Clean** project will start after **SalesSystem - Get latest version** has completed its build.

   ii. <tasks>.
```
<nant>
  &nant.exe;
  <buildArgs>-D:client.dir=&deployDirectoryBase;\Integration\SalesSystem\</buildArgs>
  <buildFile>C:\CC.Net\SalesSystem\SalesSystem.build</buildFile>
  <targetList>
    <target>SalesSystem.Clean</target>
  </targetList>
</nant>
</tasks>
```
   <task> *blocks* are the action elements of CruiseControl.Net. They're the elements that do things, like executing a program, etc

<buildFile> is the name of the build file to run.
<buildArgs> are the arguments to pass through to NAnt
<targetList> is a list of targets to be called.

c. **<project name="SalesSystem - Build">**. Compiles the project

i. <triggers>
<projectTrigger project="SalesSystem - Clean" />
</triggers>
<project Trigger> is used to trigger a build when the specified dependent project has completed its build; so, in this case, **SalesSystem – Build** project will start after **SalesSystem - Clean** has completed its build.

ii. <tasks>
<nant>
&nant.exe;
<buildArgs>-D:server.publishdir=&deployDirectoryBase;\Integration\ -D:server.configuration=Integration</buildArgs>
<buildFile>C:\CC.Net\SalesSystem\SalesSystem.build</buildFile>
<targetList>
<target>SalesSystem.Build</target>
<target>Order.Build</target>
<target>Support.Build</target>
</targetList>
</nant>
</tasks>
<task> *blocks* are the action elements of CruiseControl.Net. They're the elements that do things, like executing a program, etc
<buildFile> is the name of the build file to run.
<buildArgs> are the arguments to pass through to NAnt
<targetList> is a list of targets to be called.
<target>. In this case we have three targets; they're in charge of launch the build of SalesSystem.csproj, Order.csproj and Support.csproj respectively. If more projects have to be build more <target> tags have to be added. Also, targets have to be defined in the SalesSystem.build files described below

v. **SalesSystem.build file.** It's located in C:\CC.Net\SalesSystem folder and it contains the target that can be executed by NAnt and CC. The most important tag are the follows:

a. **Properties:**

i. <property name="root" value="${project::get-base-directory()}" />. In this case C:\CC.Net\SalesSystem is the root folder.
ii. <property name="msbuild.exe" value="C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe" overwrite="false"/>. File that will be user to build the projects.
iii. <property name="server.configuration" value="obj" overwrite="false"/>. Folder where configuration files will be saved
iv. <property name="deploy.dir" value="${root}\Deploy\Gen\Bld\" overwrite="false"/>. Folder where projects compiled will be saved

b. **Targets:**

i. **<target name="SalesSystem.Clean">**

<echo message="${deploy.dir}"/>
<delete dir="${deploy.dir}" failonerror="false" /> <!--Deletes folder C:\CC.Net\SalesSystem\Deploy\Gen\Bld -->
<mkdir dir="${deploy.dir}"/> <!--Creates folder C:\CC.Net\SalesSystem\Deploy\Gen\Bld -->
</target>

ii. **<target name="SalesSystem.Build">**

<exec program="${msbuild.exe}" workingdir="${root}\WorkingDirectory\Integration\Gen\Src" failonerror="true">
<arg value="${root}\WorkingDirectory\Integration\Gen\Src\SalesSystem.csproj "/>
<arg line="/t:Build /p:Configuration=${server.configuration} /p:OutputPath=${deploy.dir}\SalesSystem /verbosity:quiet"/>
</exec>
</target>

- **SalesSystem.csproj.** This project, generated by PLEX, has two properties that have to be changed in order to obtain the source code from the right place and to save the result .dll file in the desired place; that properties are:
  a. <SrcDir>C:\CC.Net\SalesSystem\WorkingDirectory\Integration\Gen\Src</SrcDir>
  b. <BldDir>C:\CC.Net\SalesSystem\Deploy\Gen\Bld</BldDir>

iii. **<target name="Order.Build">**

```
        <exec program="${msbuild.exe}"
    workingdir="${root}\WorkingDirectory\Integration\Gen\Src"
    failonerror="true">
            <arg
        value="${root}\WorkingDirectory\Integration\Gen\Src\Order.cspr
        oj"/>
            <arg line="/t:Build
        /p:Configuration=${server.configuration}
        /p:OutputPath=${deploy.dir}\Order /verbosity:quiet"/>
        </exec>
</target>
```

- **Order.csproj.** This project, has been generated using Code Library Wizard from PLEX; in this project we have to add a reference in order to be reached by NAnt and avoid compilation problems; the line that have to be added in the **Order.csproj** file is:
    a. <Reference Include="$(PlexInstallDirectory)\Ob.NET\Release\Plex.ObRun.dll" />

    This line must be placed in the <ItemGroup> block

```
<ItemGroup>
   <Reference Include="$(PlexRuntime)">
          <SpecificVersion>False</SpecificVersion>
          <Private>False</Private>
   </Reference>
   <Reference Include="System" />
   <Reference Include="System.Data" />
   <Reference Include="System.Xml" />
   <Reference
Include="$(PlexInstallDirectory)\Ob.NET\Release\Plex.ObRu
n.dll" />
</ItemGroup>
```

iv. **<target name="Support.Build">**

```
        <exec program="${msbuild.exe}"
  workingdir="${root}\WorkingDirectory\Integration\Gen\Src"
  failonerror="true">
                <arg
        value="${root}\WorkingDirectory\Integration\Gen\Src\Support.cs
        proj"/>
                <arg line="/t:Build
        /p:Configuration=${server.configuration}
        /p:OutputPath=${deploy.dir}\Support /verbosity:quiet"/>
        </exec>
</target>
```

- **Support.csproj.** This project, has been generated using Code Library Wizard from PLEX; in this project we have to add a reference in order to be reached by NAnt and avoid compilation problems; the line that have to be added in the **Order.csproj** file is:

    a. <Reference Include="$(PlexInstallDirectory)\Ob.NET\Release\Plex.ObRun.dll" />

    This line must be placed in the <ItemGroup> block

```
<ItemGroup>
  <Reference Include="$(PlexRuntime)">
        <SpecificVersion>False</SpecificVersion>
        <Private>False</Private>
  </Reference>
  <Reference Include="System" />
  <Reference Include="System.Data" />
  <Reference Include="System.Xml" />
  <Reference
Include="$(PlexInstallDirectory)\Ob.NET\Release\Plex.ObRu
n.dll" />
</ItemGroup>
```

v. **<target name="Commit.Add">**

```
        <exec program="C:\Program Files\VisualSVN Server\bin\svn.exe"
                commandline="add --force *.*"
                workingdir="${assemblies.dir}" />
        <exec program="C:\Program Files\VisualSVN Server\bin\svn.exe"
                commandline="commit -m&quot;Release&quot; --
username PJL --password PJL"
                workingdir="${assemblies.dir}" />
 </target>
```

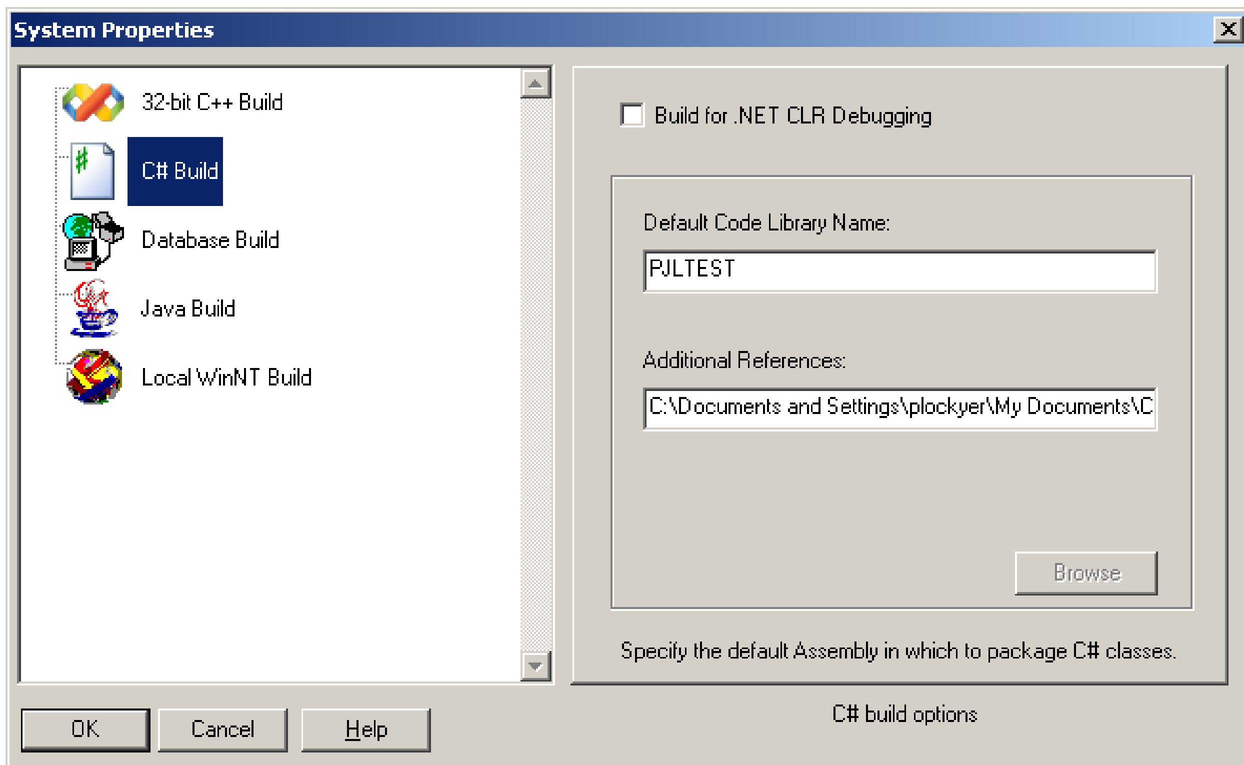
- *Commit.Add* target is used to upload the new files generated to repository path.
- The files to be uploaded must be placed into *C:\CC.Net\SalesSystem\Assemblies* folder and they will be committed into http://sc-training.eimsdirect1.local:8080/svn/SalesSystem/Assemblies path

8. Development life cycle

The next section describes the recommended developer workflow for C# development. This workflow is utilized **AFTER** the initial setup of the model for C#, including packaging, .NET assembly modeling, etc.

Developers

Using this methodology, developers should not use the Code Library Wizard during normal development. Instead developers should rely on the Default Code Library Name in their Gen and Build options defined below.



When you generate and build code, CA Plex will automatically create an assembly with the name of the default code library. This assembly should be first in the developer's code library list so it runs when they test.

For example, let's say that the application is made up of assembly1 through assembly5. In the developer's environment, their code library list should look like this:

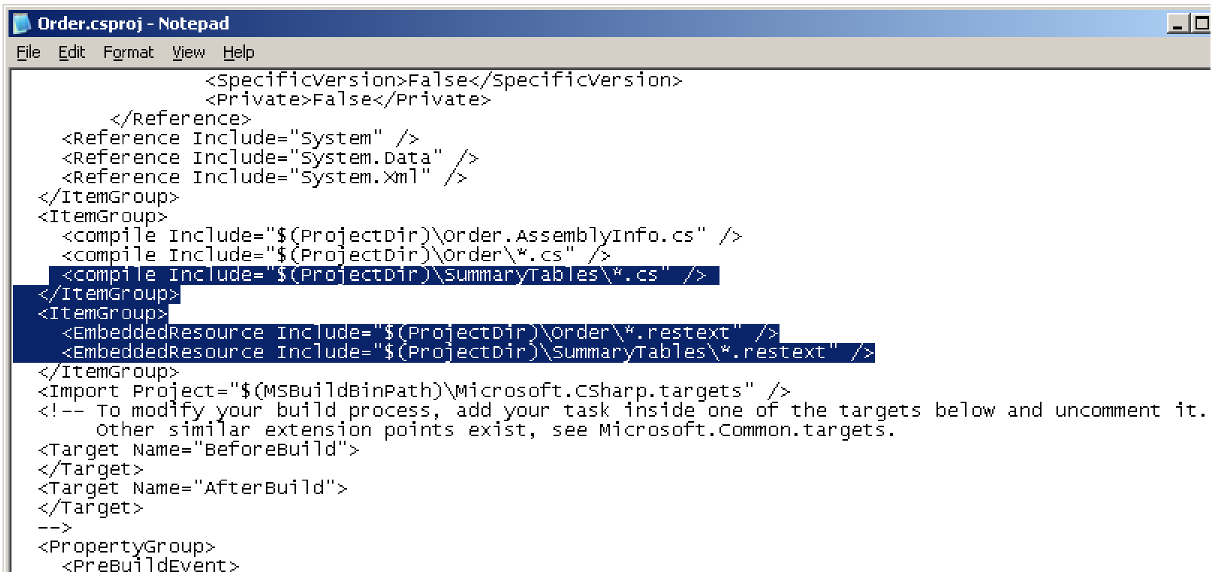
PJLTEST
assembly1
assembly2
assembly3
assembly4
assembly5

Checking tested code into subversion

Once the developer has tested their code, they should update the group model and update the repository with their new code and only their new code. A developer should never generate code that they are not working on. At this point any new functions must be packaged in the model. With regard to building the code and creating assemblies on the server, there are two options.

1. **Using CA PLEX**
   a. The developer has put any new functions in packages and if they have created any new packages they must be put into assemblies in the model.
   b. Once the model is updated to the group and the code has been updated to the repository, a model can be extracted on the server. Any new code can be extracted from the repository and built from within CA Plex. The Code library wizard should also be used to recreate the assemblies. These assemblies should be updated to the repository.
   c. The developer should delete their test assembly (PJLTEST in the example above) and the new assemblies should be extracted from the repository. The developer has the latest functionality from other developers and is running their recent changes from the correct assemblies. They can continue developing and any functions they build will be written to their test assembly.

2. **Using Cruise Control**
   a. The developer has put any new functions in packages. Since under cruise control the assemblies do not need to be managed in CA Plex any packages that are to be added to an assembly must be added to the csproj file for an assembly. As an example, if we have an assembly called *Order*, there will be a file called *Order.csproj*. If I add a package called *SummaryTables* it needs to be added to the *Order.csproj* as shown below in the first and last highlighted lines.



```
Order.csproj - Notepad
File  Edit  Format  View  Help
                <SpecificVersion>False</SpecificVersion>
                <Private>False</Private>
            </Reference>
        <Reference Include="System" />
        <Reference Include="System.Data" />
        <Reference Include="System.Xml" />
    </ItemGroup>
    <ItemGroup>
        <Compile Include="$(ProjectDir)\Order.AssemblyInfo.cs" />
        <Compile Include="$(ProjectDir)\Order\*.cs" />
        <Compile Include="$(ProjectDir)\SummaryTables\*.cs" />
    </ItemGroup>
    <ItemGroup>
        <EmbeddedResource Include="$(ProjectDir)\Order\*.restext" />
        <EmbeddedResource Include="$(ProjectDir)\SummaryTables\*.restext" />
    </ItemGroup>
    <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
    <!-- To modify your build process, add your task inside one of the targets below and uncomment it.
         Other similar extension points exist, see Microsoft.Common.targets.
    <Target Name="BeforeBuild">
    </Target>
    <Target Name="AfterBuild">
    </Target>
    -->
    <PropertyGroup>
        <PreBuildEvent>
```

b. Cruise Control will rebuild any updated or new functions in the repository and create the associated assemblies. We can configure Cruise Control to do this as soon as it detects a change, or at a specified time such as 4am every night. It can also be triggered on demand.

c. Developers then get the update assemblies from the source repository.

d. After the new assemblies are extracted, the developer should delete their test assembly (PJLTEST in the example above) and the new assemblies should be extracted from the repository. The developer has the latest functionality from other developers and is running their recent changes from the correct assemblies. They can continue developing and any functions they build will be written to their test assembly.

Either of these work flows can be modified over time to suit your exact needs. **ADC Austin recommends cruise control for large applications, as a proven production-quality solution that can more completely automate the C# build process.**

Appendix

Document from CA Plex Sample Model

.NET Support and Code Libraries

Sample Model

Introduction

CA Plex r6.0 introduces the ability to generate your server-based applications as .NET managed code applications. Functions given a language of C# are generated as C# class files, which are then compiled into assemblies and are run using the CA Plex .NET runtime.

Whilst many of the concepts associated with Plex .NET applications will be very familiar to most Plex developers, the .NET platform introduces a number of new concepts and improvements which this document addresses.

The main topics that this guide covers are:

(1) Prerequisites: What you need to have installed in order to successfully run these examples.
(2) Running a Simple Client-Server Application: Shows the steps required in order to generate, build and run a Plex application based over the pattern libraries.
(3) Using the Plex .NET Runtime Service: Shows how you can configure Plex generated .NET Server applications to use the new Plex .NET Runtime Service.
(4) Visual Studio 2005 Integration: This section shows how to integrate Plex generated .NET applications with other client applications.
    (a) Stateful Calls: Shows how you can make calls into the Plex runtime where a client is bound to a Plex runtime server instance, so that transaction, cursor and other information is maintained between calls to the runtime. This example is based over a Windows Application developed in C#.
    (b) Stateless Calls: Shows how you can make calls into the Plex runtime where a client is not bound to a Plex runtime server instance. This example is based over an ASP.NET Web Application, also developed in C#.
(5) Using Code Library Objects to Create Assemblies: This section shows you how to divide your application into logical deployment units, and how to deploy these units as an application.

.NET Support and Code Libraries

Sample Model

(1) Prerequesites

In order to run this example, you will need a minimum of the following installed on your PC.

(1) **CA Plex r6.0 or later version.**

(2) **Microsoft SQL Server:** You should be able to build the database schemas for this example against any version of SQL Server, although SQL Server 2005 is recommended. For a comparison of SQL Server 2005 editions, refer to the Microsoft SQL Server Website.

(3) **Microsoft .NET Framework Version 2.0 Redistributable Package:** The version recommended is v2.0.50727, and is available from the Microsoft .NET Framework Website. It is also installed by default with Microsoft Visual Studio 2005.

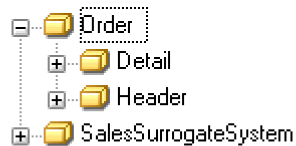
(4) **Microsoft Visual Studio 2005:** Although the .NET Framework Version 2.0 Redistributable Package contains the tools necessary to build generated C# source code, you will need to install Visual Studio 2005 in order to compile the Plex generated unmanaged C++ Client applications used in this example, and also to integrate your generated Plex .NET server applications with other 3rd party client applications. This is especially true if you would like to take advantage of the new CA Plex r6.0 .NET Runtime API layer. Version 8.0.50727 or above is recommended.

.NET Support and Code Libraries
Sample Model

(2) Running a Simple Client-Server Application

Introduction

The following example uses the sample model `SalesSystem.mdl` that can be found in the `Samples\Dot NET Support and Code Libraries\` directory.
The sample consists of three relatively simple entities that comprise a Sales Order System; **Order.Header**, **Order.Detail** and **SalesSurrogateSystem**; the latter is a support entity used to maintain information on order keys that have been allocated.



As you can probably imagine, Order.Detail is owned by Order.Header. Both Header and Detail obtain their primary keys automatically by inheriting from patterns that allocate surrogate keys. Header inherits from FOUNDATION/Surrogate and Detail inherits from FOUNDATION/SurrogateOwned (so that the primary keys are allocated uniquely within their super-ordinate keys).

All three entities have database tables and views associated with them, and as such, they inherit from STORAGE/RelationalTable.

In order to generate and build this example targeting a Windows C++ client running to a C# .NET server, we simply need to set the relevant variants for the libraries attached to SalesSystem from which it inherits. Because the Windows C++ client is the default variant for the client pattern libraries, this only means we have to set the STORAGE library to a variant of '.NET server'.

Model	Variant	Language	Version	
GroupMod	Base	Base	Base	Base
ACTIVE	Base	Base	V6.0 Patterns	V6.0 Patterns
DATE	Windows client	Base	V6.0 Patterns	V6.0 Patterns
FIELDS	Base	Base	V6.0 Patterns	V6.0 Patterns
Foundati	Base	Base	V6.0 Patterns	V6.0 Patterns
JAVAAPI	Base	Base	V6.0 Patterns	V6.0 Patterns
OBJECTS	Base	Base	V6.0 Patterns	V6.0 Patterns
STORAGE	.NET server	Base	V6.0 Patterns	V6.0 Patterns
UIBASIC	.NET server	Base	V6.0 Patterns	V6.0 Patterns
UISTYLE	Base	Base	V6.0 Patterns	V6.0 Patterns
VALIDATE	HP UNIX/Oracle server	Base	V6.0 Patterns	V6.0 Patterns
WINAPI	JAVA JDBC server	Base	V6.0 Patterns	V6.0 Patterns
	JAVA Oracle server			
	Jet Engine Server			
	NT ODBC server			

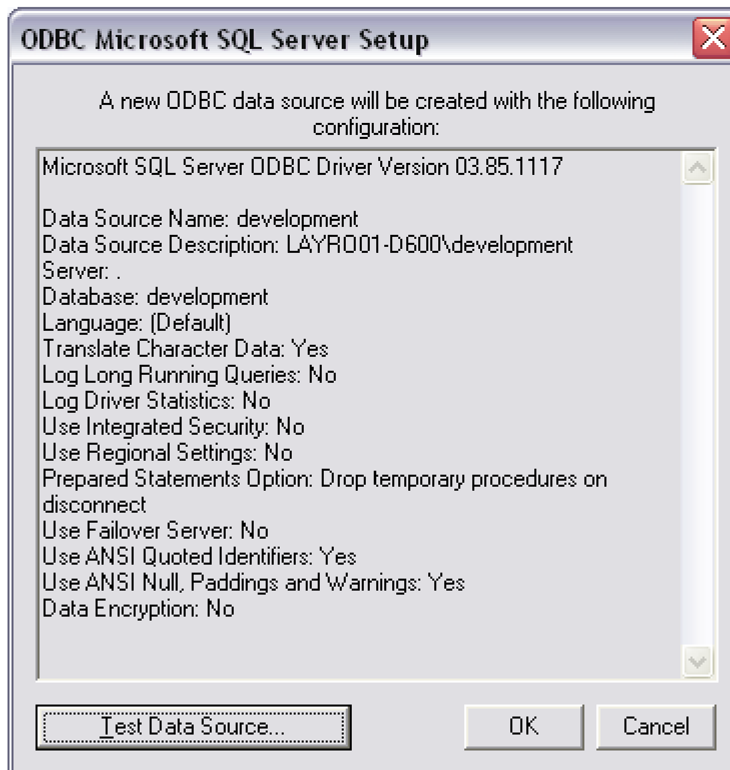
This has already been set in the SalesSystem model for you. The only thing you need to do in order to run the example is to generate it, build it and configure the Plex .NET Runtime so that the client can call the server. The following steps will walk you through this process.

.NET Support and Code Libraries

Sample Model

Steps to Create and Run the Application

(1) First, you will need to configure the database to which you wish to run to. In order for Plex to build the necessary table and view schemas, you will need to configure an ODBC data source to connect to the database. The remainder of this document will reference a data source called 'development' that connects to a SQL Server 2005 instance.



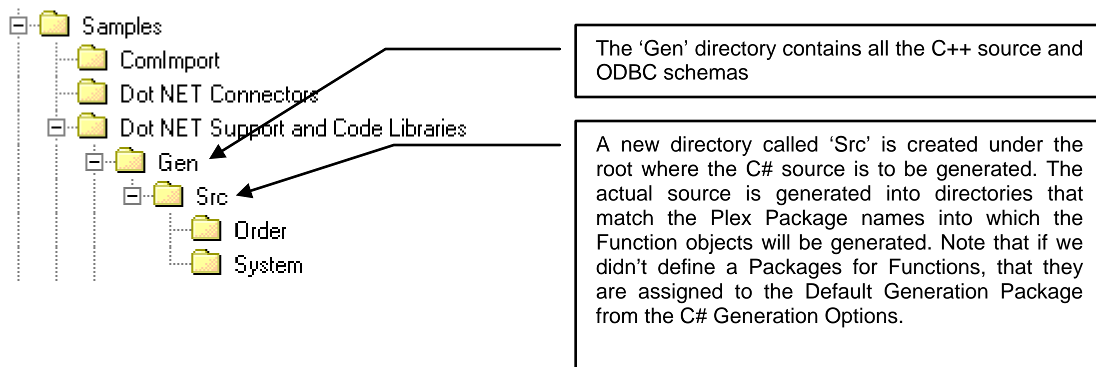
(2) With the model open, go to the Generate and Build window by selecting **Tools→Generate and Build…** from the main menu. Take a few moments to review the settings associated with the model by selecting **Build→Generate and Build Options…**. Out of all the settings, take special note of the following:

(a) **Build Directories:** Make sure that you set the build directories.
(b) **C# Generation Options:** This tab contains options that affect the generated C# source code. Currently, the only option is the Default generation package. This defines the .NET namespace into which a Plex generated C# function will be placed if it is not assigned to a Plex Package object. As none of the functions in this simple example have been modeled into packages, this will be the namespace into which our generated C# classes will be placed.
(c) **System Definitions/<Local machine name>/C# Build:** This tab contains the options required for building C# source code.
   (i) **Build for .NET CLR Debugging:** Does exactly what it says.
   (ii) **Default Code Library Name:** Defines the name of the assembly dll into which Plex generated C# functions will be placed when they are built from the standard Generate and Build window. Deploying Plex generated .NET applications will be the subject of a separate discussion on Code Libraries.

(iii) **Additional References:** Allows you to specify additional .NET assemblies that your Plex generated C# functions might reference. For example, you may add C# source code to your action diagrams that reference external .NET components or pre-built Plex C# functions from another Plex model.

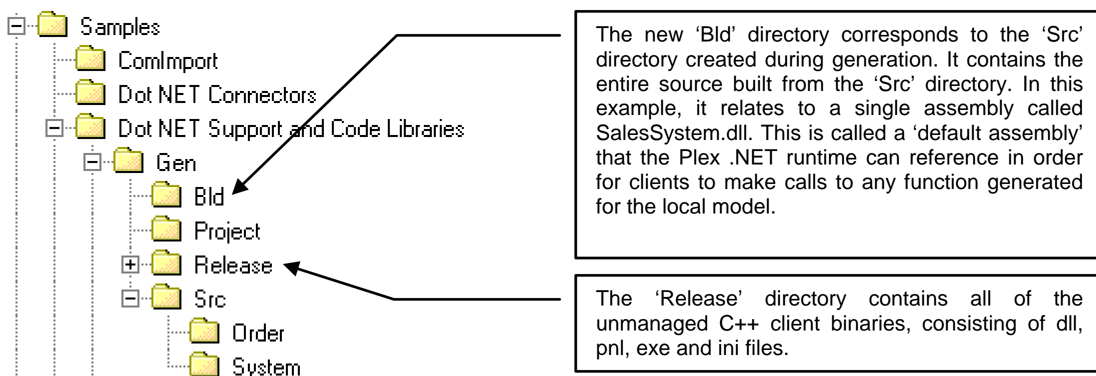
(d) **System Definitions/<Local machine name>/Database Build:** This is where you set the options used when generating database schemas to an SQL database. Make sure you select the ODBC data source you defined in (1) so that Plex can automatically generate the necessary database schemas.

(3) Select the subject area called 'Generate and Build Me', and generate the source for this subject area (**Build→Generate**). You should be prompted to generate 37 objects. Once the generation has completed take a look at the directory into which the source has been generated.



(4) Reselect the subject area 'Generate and Build Me', and build the source for this subject area (Build→Build). You should be prompted to build 15 C++ functions and 22 C# functions/ODBC objects.

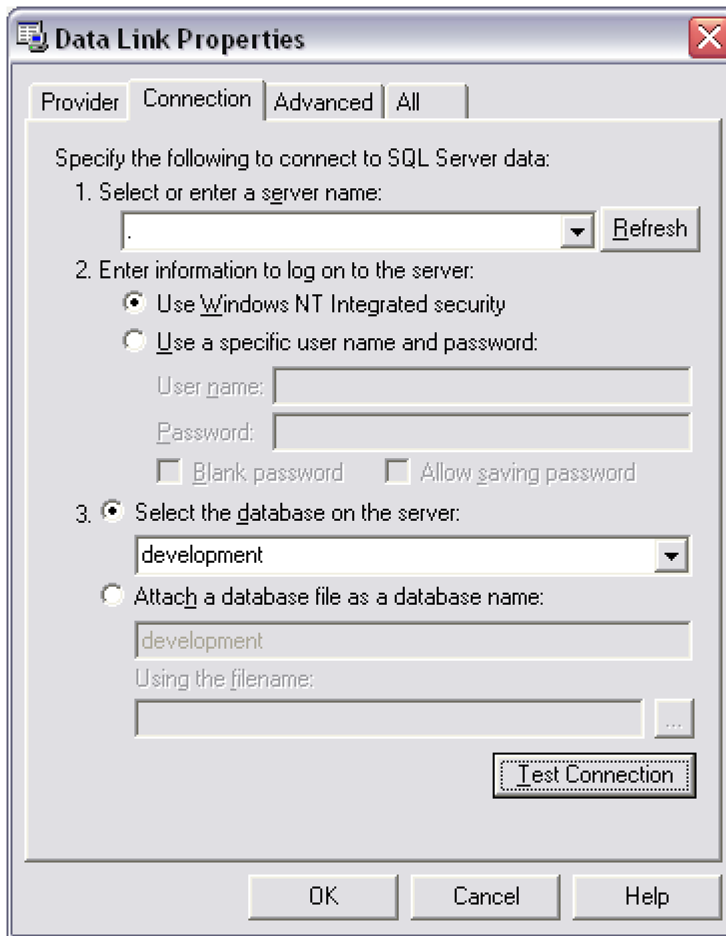
(5) Highlight the Function in the Subject Area 'Create an Executable for Me' and select **Build→Create Exe** in order to create a client executable entry point for the application. Now look at the generation directories to see the locations of the various binaries that comprise this application.



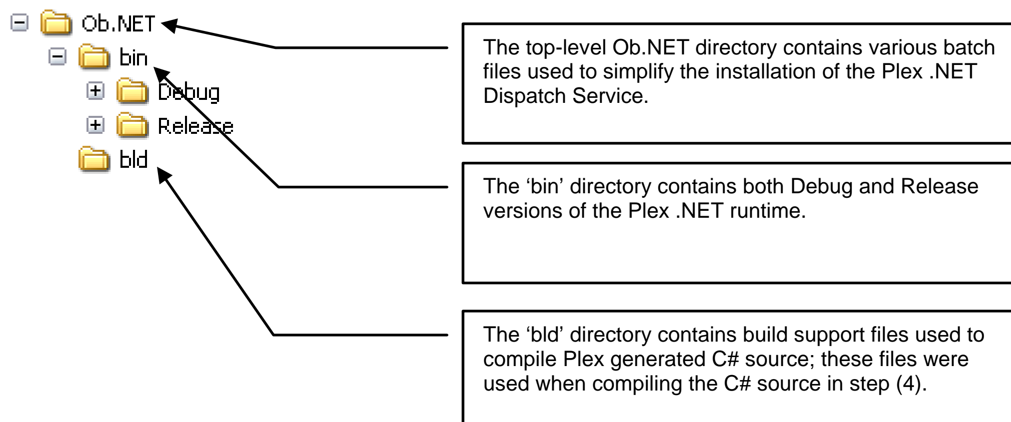
.NET Support and Code Libraries

Sample Model

(6) In order for the Plex generated .NET Server application to connect to the database, an OLEDB.NET connection must be used. This example uses a Microsoft Data Link (UDL) file in order to connect to the database. A sample UDL file is located in the `Samples\Dot NET Support and Code Libraries\` directory for you to configure for your particular database. Double-click the file in order to configure it. The following shows `SalesSystem.udl` configured to access the development database described in (1).



(7) Now locate the Plex .NET Runtime directory under the main CA Plex r6.0 directory (called '`Ob.NET`'). The Plex .NET Runtime is structured as follows.



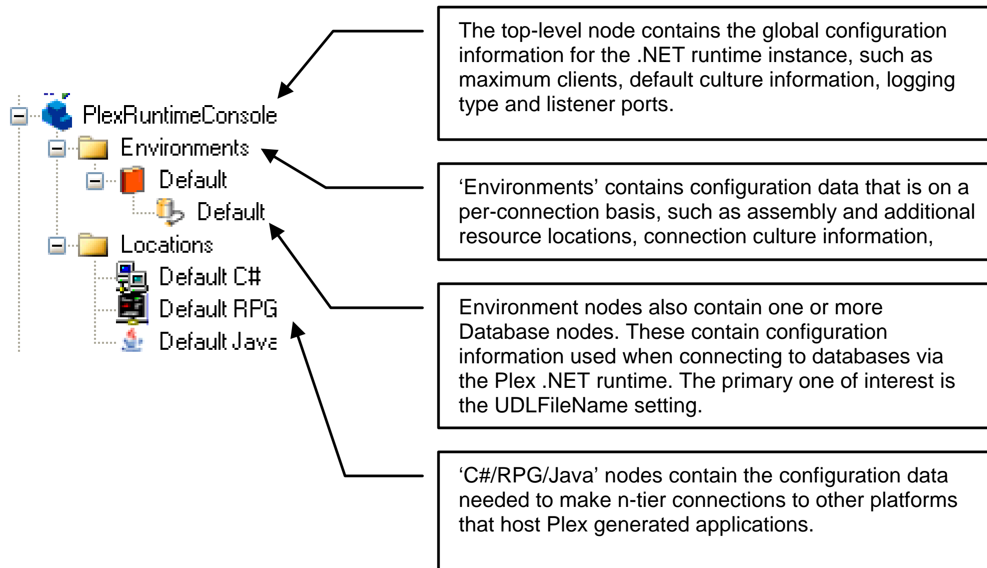
.NET Support and Code Libraries

Sample Model

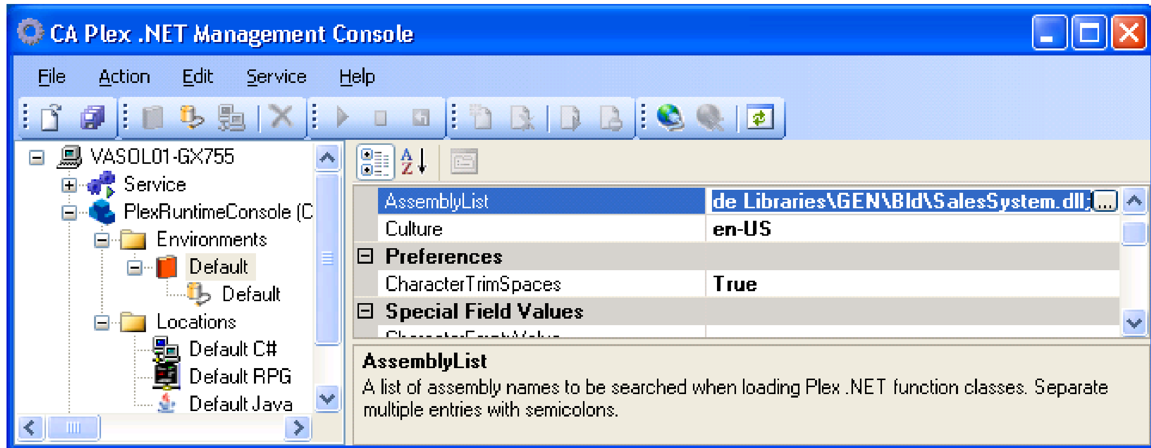
In order to run our Plex client-server application, we must first configure the Plex .NET Runtime instance that we wish to run the generated Plex .NET server application under. Because we will be running the server application as a console application which is started via `PlexRuntimeConsole.exe`, we need to configure the configuration file `PlexRuntimeConsole.exe.config`.

(a) You can configure the Plex .NET Runtime using the new CA Plex .NET Management Console. Start the interface by double-clicking the executable `PlexManagementConsole.exe`, which is located in the `<PlexDirectory>\Ob.NET\bin\Release\` directory or use the start menu item – Start/Programs/CA/CA Plex r6.1/Plex .NET Tools/Application Management Console. By default, the CA Plex .NET Management Console opens two configuration files – PlexRuntimeConsole and PlexGenericRuntime. For the .NET Console runtime, we need to update the PlexRuntimeConsole configuration file.

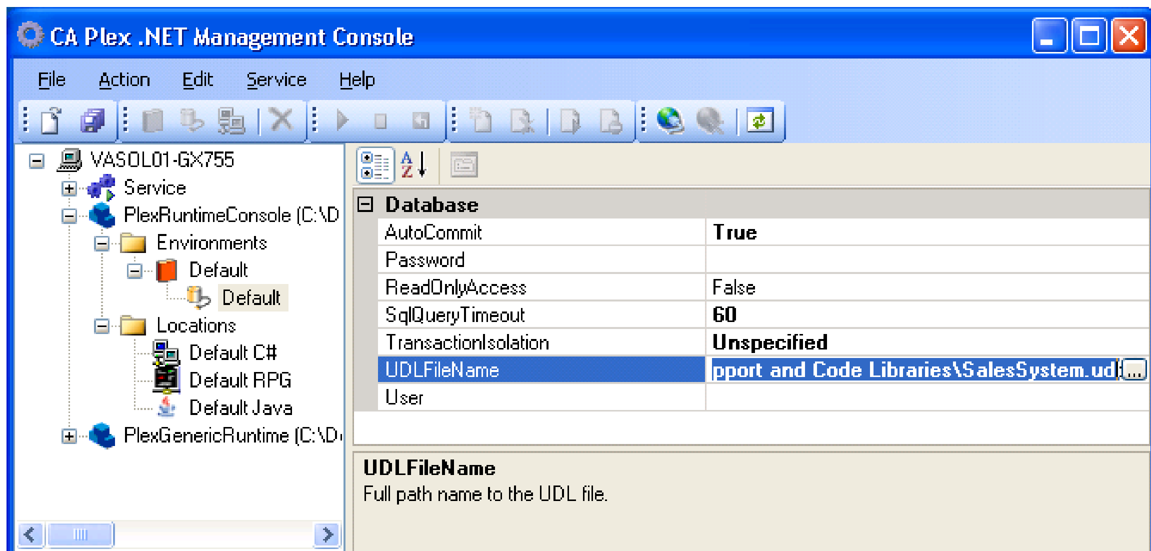
(b) The .NET Management Console allows you to maintain configuration information on the following items.



(8) Highlight the environment called ‘Default’ to see its settings. Highlight the AssemblyList parameter, browse to the assembly `SalesSystem.dll` built in step (5), and click OK.

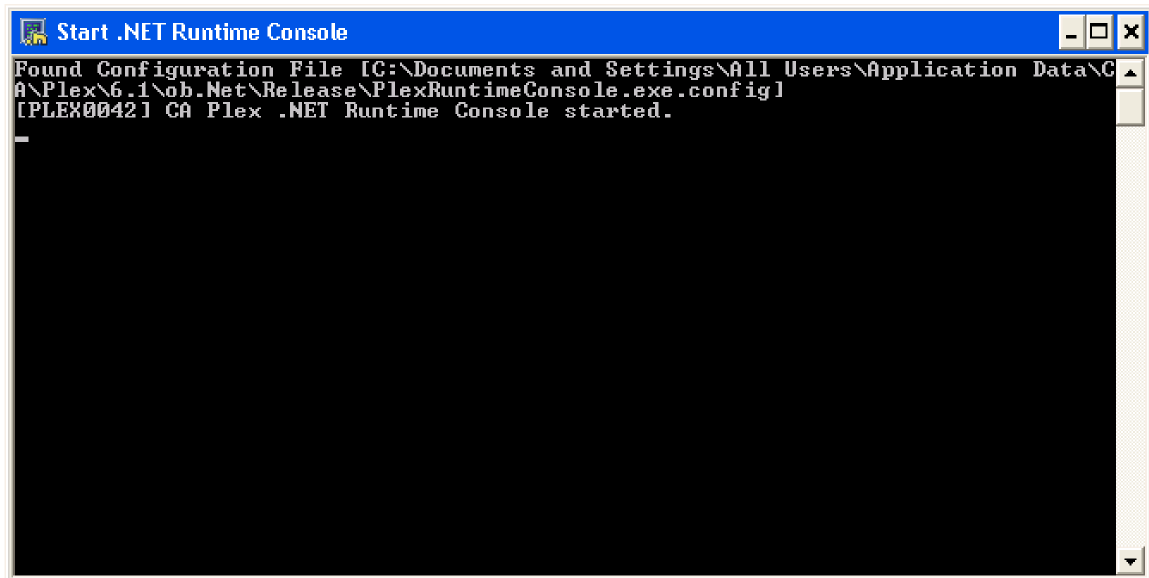


Highlight the database called ‘Default’ under the environment and browse to the SalesSystem.udl file defined in step (6), and click OK.



(9) Save the configuration file by selecting **File→Save All**, and exit the Plex .NET Management Console.

(10) To start the server application, double-click the `PlexRuntimeConsole.exe` executable or use the start menu item Start/Programs/CA/CA Plex r6.1/Plex .NET Tools/Start .NET Runtime Console. A DOS-style command window should appear, telling you that the Plex .NET Runtime Console has been started, and is listening for requests on port 1998 (this is defined in the Port setting under the root node of the configuration file).



(11)Locate the Plex generated client application in `Samples\Dot NET Support and Code Libraries\Gen\Release\` directory. In order for the Plex generated client to call the server application, it needs to be configured to connect to the server (although the client and server are running on the same machine, the principle is the same as if they were located on different machines). To configure the client runtime, open the file `HeaderGrid.ini`. This contains a section called `[RemoteCSharp]` that contains the connection information to connect to the .NET server. The primary values of interest are:

(a) **System:** The name of the system on which the Plex .NET server application resides.
(b) **Port:** The port number that the remote machine is listening on.
(c) **Environment:** The name of the environment on the server that contains the configuration information for the application.

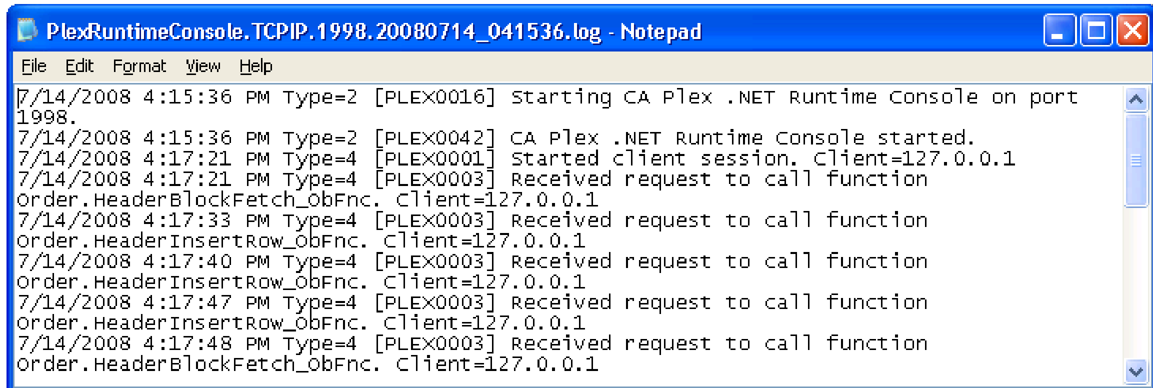
Because this example is running to the local machine, on port 1998 using the default environment, these values can be left as they are.

(12)To run the application, double-click `HeaderGrid.EXE` to start the client application. Try out the application by doing the following:

(a) Enter some header records using the 'Add new data.' Button.
(b) Enter some detail records for a selected header by selecting the 'Detail' button.

You can check the server status by checking the text log file associated with the server runtime. You should see trace messages created in a file called `PlexRuntimeConsole.TCPIP.1998.<timestamp>.log` each time a server function is called (this can be switched off by a configuration setting in the runtime). The log file location is C:\Documents and Settings\All Users\Application Data\CA\Plex\6.1\ob.Net\log or C:\ProgramData\CA\Plex\6.1\ob.Net\log, depending on the OS. The trace messages should all be informational, and should look as follows.

.NET Support and Code Libraries
Sample Model



You have the choice of directing Plex .NET Runtime messages to either a trace log file, or the Application Event Log. You can select where to send the messages via the configuration file associated with the Plex runtime you are executing.

The trace file or event log should also be the first place to look if you have problems with your Plex .NET Runtime.

.NET Support and Code Libraries

Sample Model

Grid

Detail

OrderNumber	CustomerName	OrderDate
1	Rob's Records	5/19/2006
2	Kiyoshi's Lamp Co.	2/19/2007
3	Paul's Boutique	2/19/2007

Edit

OrderNumber	LineNumber	ProductName	LineQuantity	LinePrice
1	1	Widgets	5	199.95
1	2	Dongles	10	34.99
1	3	Tartan Paint	60	2000.50

OrderNumber 1
ProductName Widgets
LineQuantity 5
LinePrice 199.95
Apply New Delete Refresh
Continue new?

(13)Close the client application and the Plex .NET Runtime Console before continuing to the next section.

(3) Using the Plex .NET Runtime Service

Introduction

In addition to running the Plex .NET Runtime as a console application, you can also run your server applications under a service on the target machine.

Running your Plex .NET server applications under a service has a few noticeable benefits:

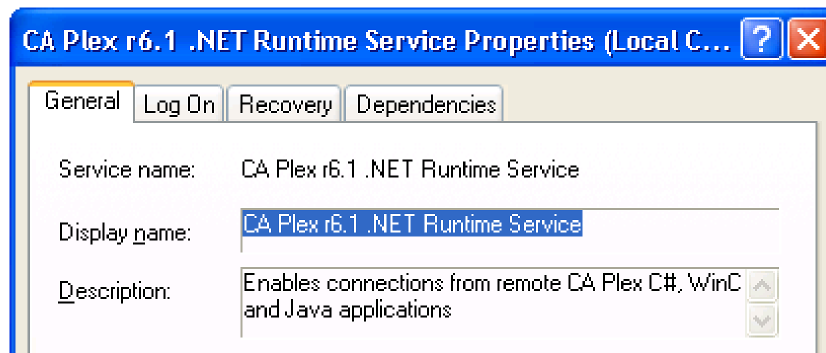
(1) You can configure multiple ports on which to service Plex client applications. These are called 'Listeners', each of which has their own unique configuration information, memory space and state information.
(2) The configuration of these listeners can be carried out by the Plex .NET Management Console interface, thereby making it simple to manage the application instances you have installed and running on a particular server, even remotely.
(3) You can obtain extended diagnostic information on your running Plex .NET server application instances for each listener. This includes information on active threads, database connections, SQL statement execution and remote n-tier connections that have been made in the runtime.

In this example, we will configure the application generated in section ***(2) Running a Simple Client-Server Application*** to use the new Plex .NET Runtime Service.

Steps to Configure the Plex .NET Runtime Service

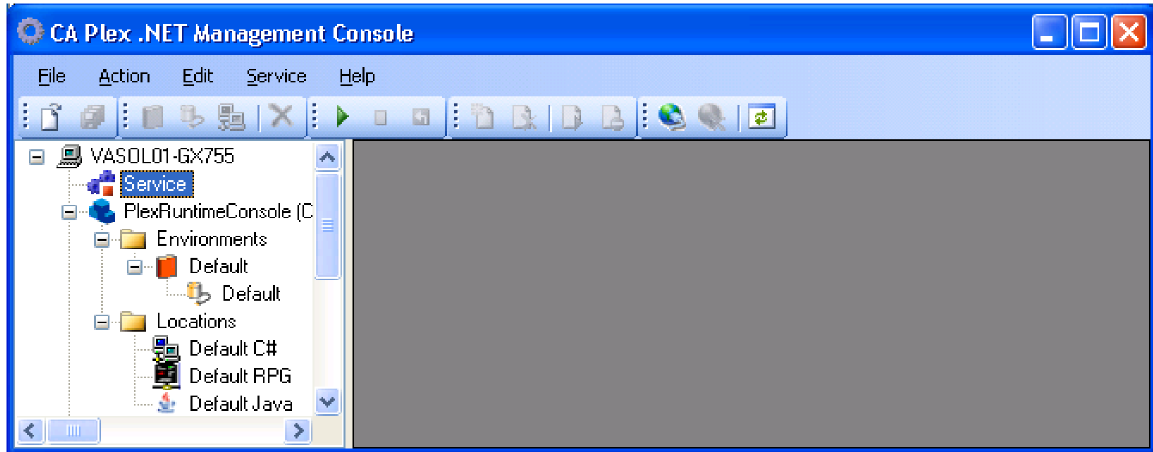
(1) If you selected the .NET Runtime option on the Plex install 'Select Features' screen, the CA Plex r6.1 .NET Runtime Service should be already installed. To install the service, you need to use the .NET Framework utility `installutil` on the service executable `\Ob.NET\bin\Release\PlexRuntimeService.exe`.

For convenience, batch files to install and uninstall the service are installed in the `Ob.NET` directory. To install the service, double-click on the batch file `\Ob.NET\InstallService.bat`. Once installed, you should be able to see the Plex .NET Runtime Service installed under the Services node in the Microsoft Management Console. Do not start the service yet – we will do that a bit later.

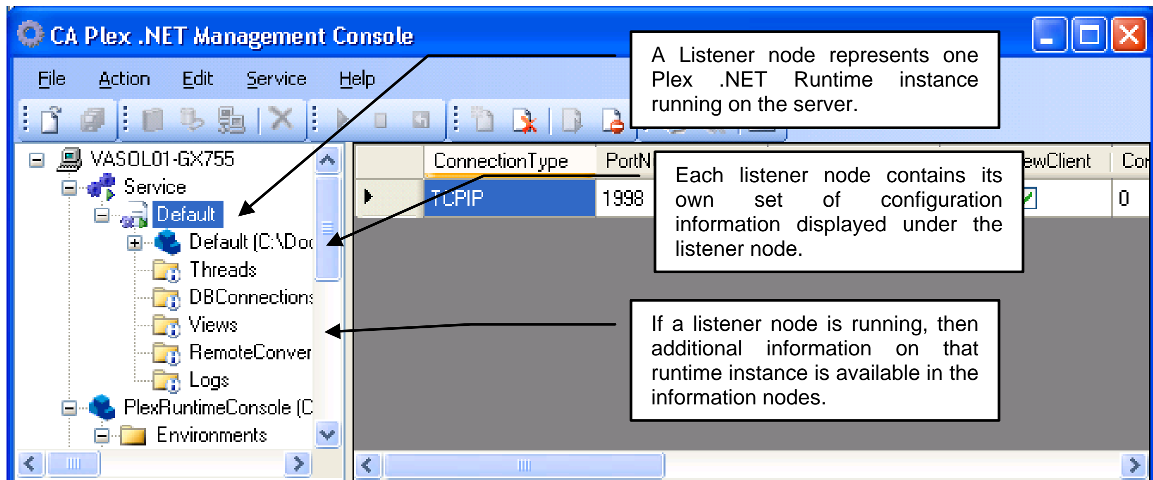


(2) Now open the Plex .NET Management Console. You should see a service node appear under the root machine node in the left-hand tree view. The red square against the service

node denotes that the service is currently stopped. You can now use the Plex .NET Management Console to control the service installed on the local machine.



(3) Select the service node, and select **Service→Start** in order to start the Plex .NET Runtime Service. After a brief pause, the service node icon should change from showing a red square to showing a green arrow, showing that the status of the service has changed into the running state. The service node should also have a (+) next to it showing that it contains additional information. Expand the service node and select the listener node called 'Default' running under the service.

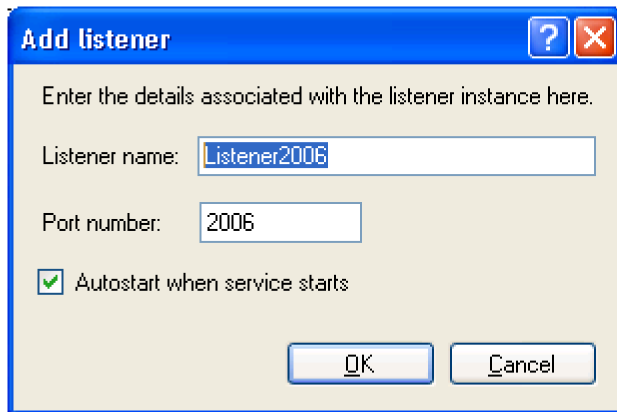


(4) To make sure that we're not using the default listener, stop it by selecting **Service→Listeners→Stop**. You should see that the green arrow icon next to the listener node changes to a red square. The configuration and information nodes under the listener should also disappear.

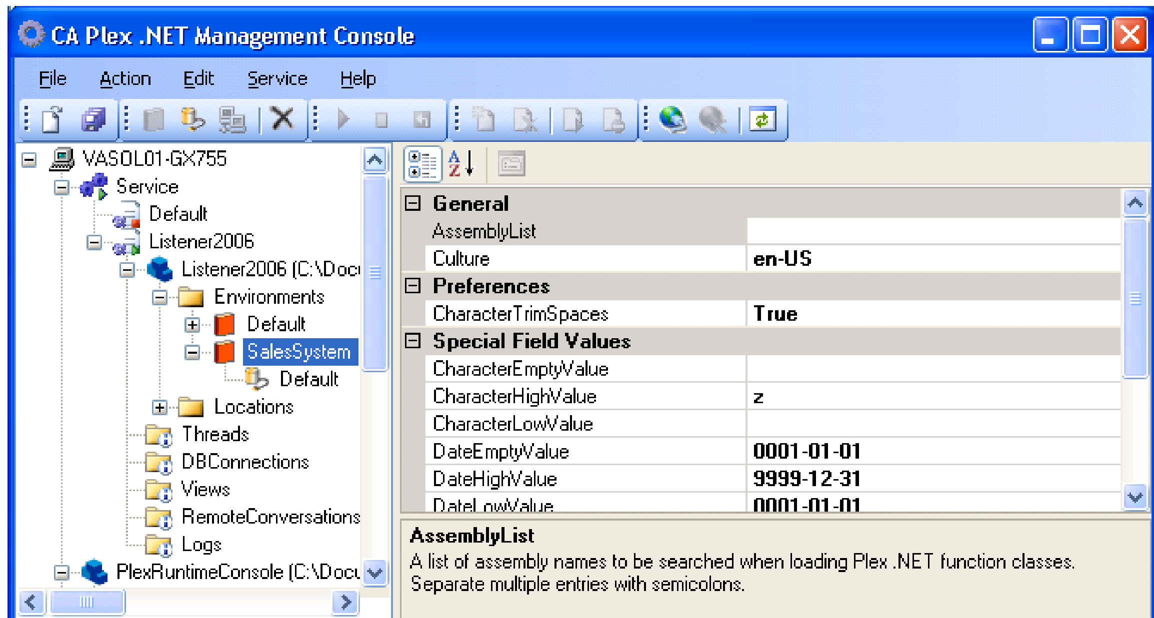
(5) To add a new listener, select **Service→Listeners→Add…** You should be presented with a dialog to enter the unique listener details, which consist of the following:

(a) **Listener name:** The name used to identify the listener. This name is also used to set the name of the .config file used to store the configuration information on the server for the runtime instance.
(b) **Port number:** This is the unique port number that the listener will take incoming requests from Plex client applications.
(c) **Autostart:** By selecting autostart, the listener will start running as soon as the .NET Runtime Service starts. If autostart is not enabled, then the listener must be started manually once the service enters the running state.

Enter 'Listener2006' for the listener name, 2006 for the port number and leave autostart set to true, then select 'OK'.

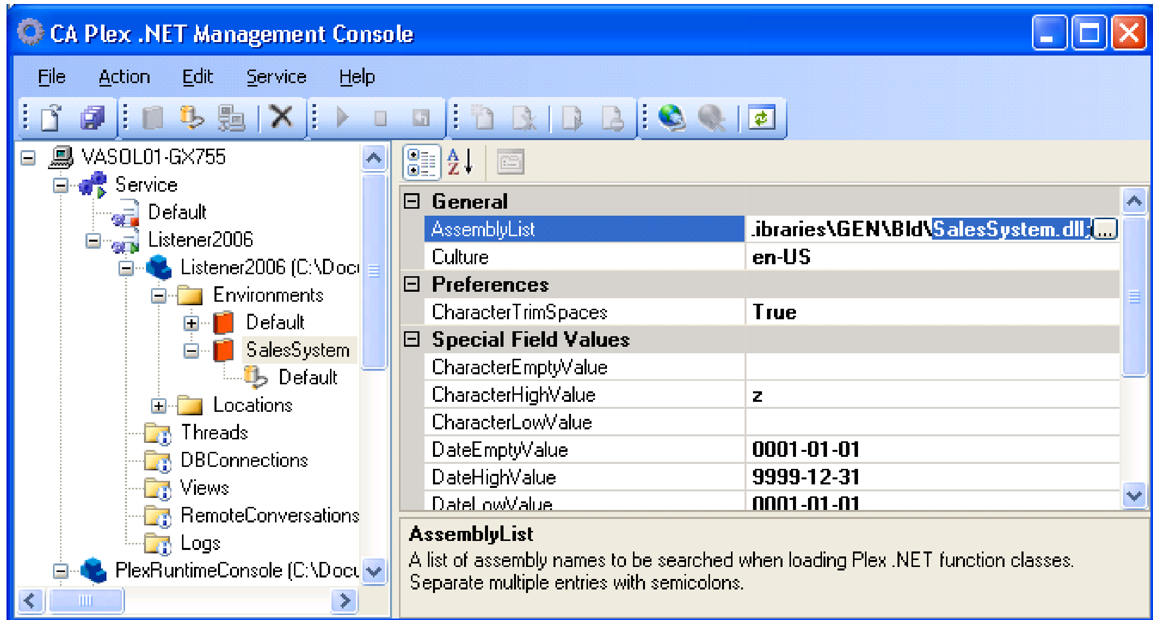


(6) You should see that Listener2006 has been added under the service, and because it was set to autostart, it should be running. Select the new listener node in order to load the configuration file associated with it. Highlight configuration node and select **Add→Environment** to create a new environment entry. Enter 'SalesSystem' for the environment name, and press enter.

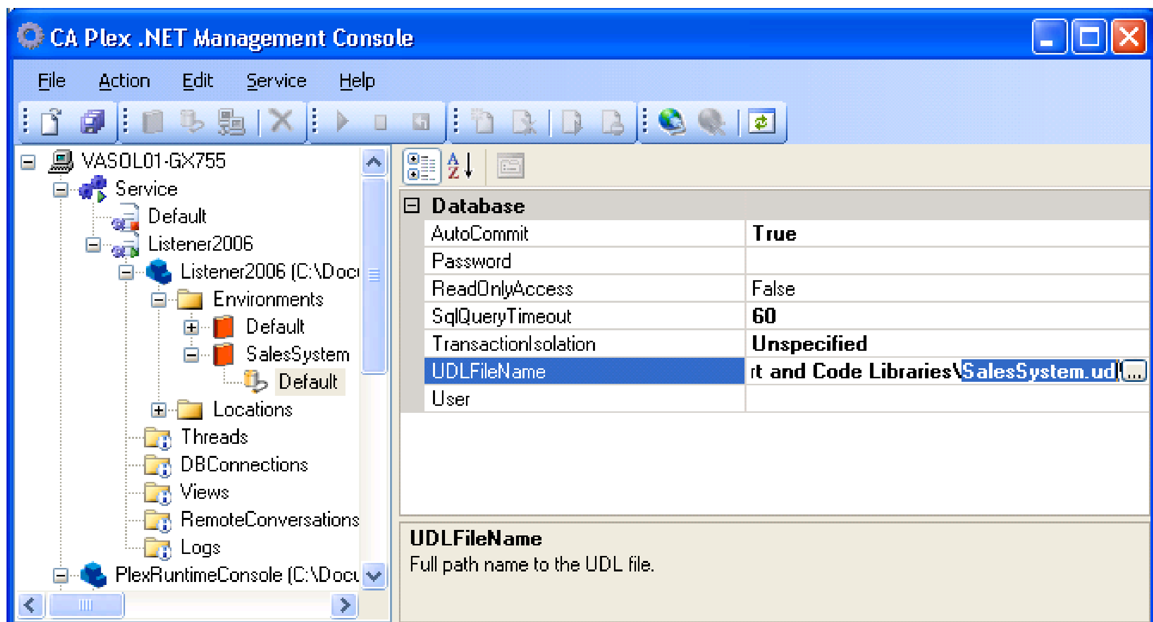


.NET Support and Code Libraries
Sample Model

In the AssemblyList parameter of the newly added environment, and enter the path to the assembly `SalesSystem.dll` built in the previous example.



(7) Highlight the database called 'Default' under the new environment. Browse to the SalesSystem.udl file.



(8) Save the configuration file by selecting **File→Save All**. This will save the configuration file onto the server. If you want to view the configuration file it is located in the same directory where the service executable resides (i.e. `\Ob.NET\bin\Release\`).

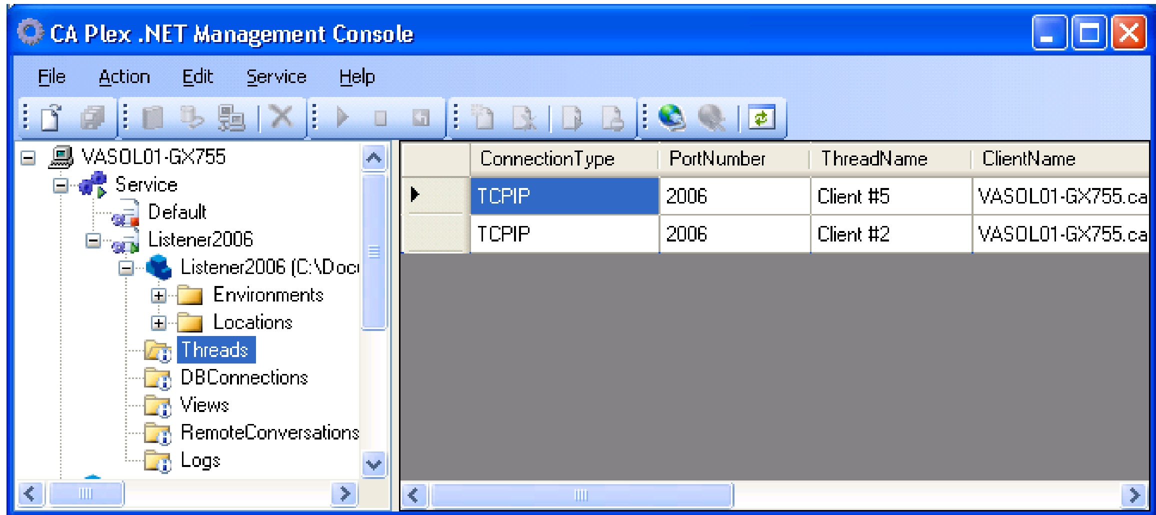
(9) The service and listener are now ready to receive connections from a Plex client application. Locate the Plex generated client application in `Samples\Dot NET Support and Code Libraries\Gen\Release\` directory. Open the file `HeaderGrid.ini`, and change the section called `[RemoteCSharp]` as highlighted below.

```
[RemoteCSharp]
System=localhost
OpSys=WINCLR
Protocol=WINTCPIP
Port=2006
Environment=SalesSystem
Timeout=60
Client Encoding=Windows-1252
Program=
Path=
Buffer Size=1
Package=
```

(10) Start the client application and make sure everything works as expected; it should work exactly as it did before. Now start up a second client on the same machine, and make some more server calls by adding, updating or deleting records.

(11) Open up the Plex .NET Management Console and look at the following information displayed for the listener.

(a) **Listener node:** Shows properties associated with the listener, such as current number of clients connected, total number of clients serviced and other runtime configuration properties such as the logging type and level.
(b) **Thread node:** Shows information on each individual client currently connected to the listener, such as the machine name, IP address and number of client to server calls.
(c) **DBConnections node:** Shows information on each database connection currently active, such as connection string and connection type.
(d) **Views node:** Shows information on the individual SQL statements currently open. You can see the SQL statements executed and the Plex server functions they were called from.
(e) **Remote conversations node:** Shows any .NET Server to .NET/Java/System i connections that might be active against the listener.
(f) **Logs node:** Shows runtime log messges.



(12)Close the client applications and stop the Plex .NET Runtime Service before continuing to the next section.

(4) Visual Studio 2005 Integration

Introduction

The CA Plex .NET Runtime allows simple integration with other applications based on the .NET platform by using the DataSet and other associated classes from the System.Data namespace in the .NET Framework.

When calling a Plex generated function, a DataSet represents either the input or output variable groups associated with that function. DataTables are used to represent the individual variables within the input or output group; with individual DataColumns used to represent the fields of those variables.

InHeaderBlockFetch (.Input)

Position

OrderNumber
0

Control

Position	RowsFetched
'Y'	0

A DataSet represents a parameter interface variable group from Plex. Input contains both input and dual variables.

A DataTable represents one variable on the parameter interface.

A DataColumn contains the information on one field in the variable on the parameter interface.

OutHeaderBlockFetch (.Output)

FetchedData

OrderNumber	CustomerName	OrderDate
1	Rob's Records	01/01/2006
2	Kiyoshi's Lamp Co.	02/02/2006
64	Cheeky's Loop Store	10/21/2006

A DataRow contains the actual field data passed to and from the function call. If the variable is a single instance, then there will only be one DataRow in the DataTable instance. If the variable is a MOV, then there will be a DataRow object for each instance passed to or from the function.

This section consists of two examples, showing both a stateful and stateless call into the Plex runtime.

At the end of a stateful call into the Plex runtime, the thread used by the client is maintained for subsequent calls into the runtime. This allows a client application to maintain cursors between calls into the Plex runtime.

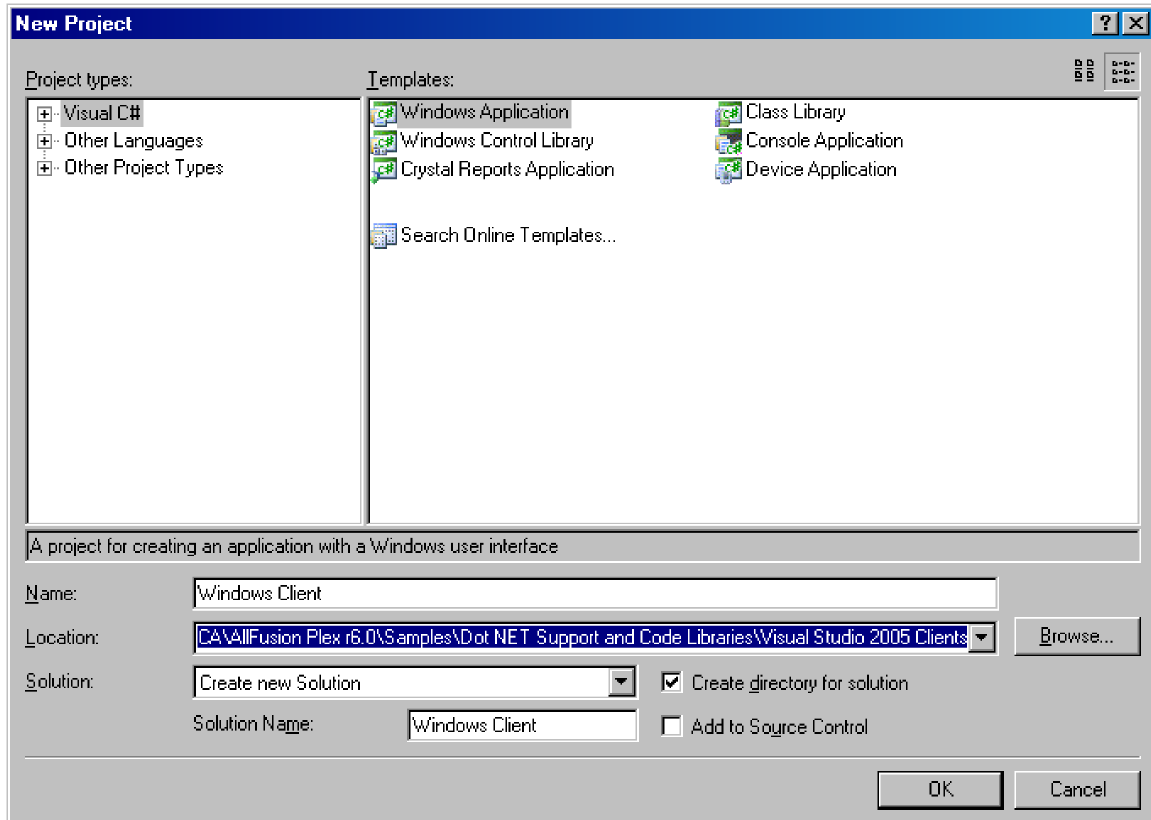
At the end of a stateless call into the Plex runtime, the thread used by the client is discarded. Thus any cursors that may have been opened during the call are lost. To make a stateless call into the Plex runtime, you need to use the static `callFunction()` method exposed on the `ObRun.ObUtils.ObUserApi` class.

.NET Support and Code Libraries

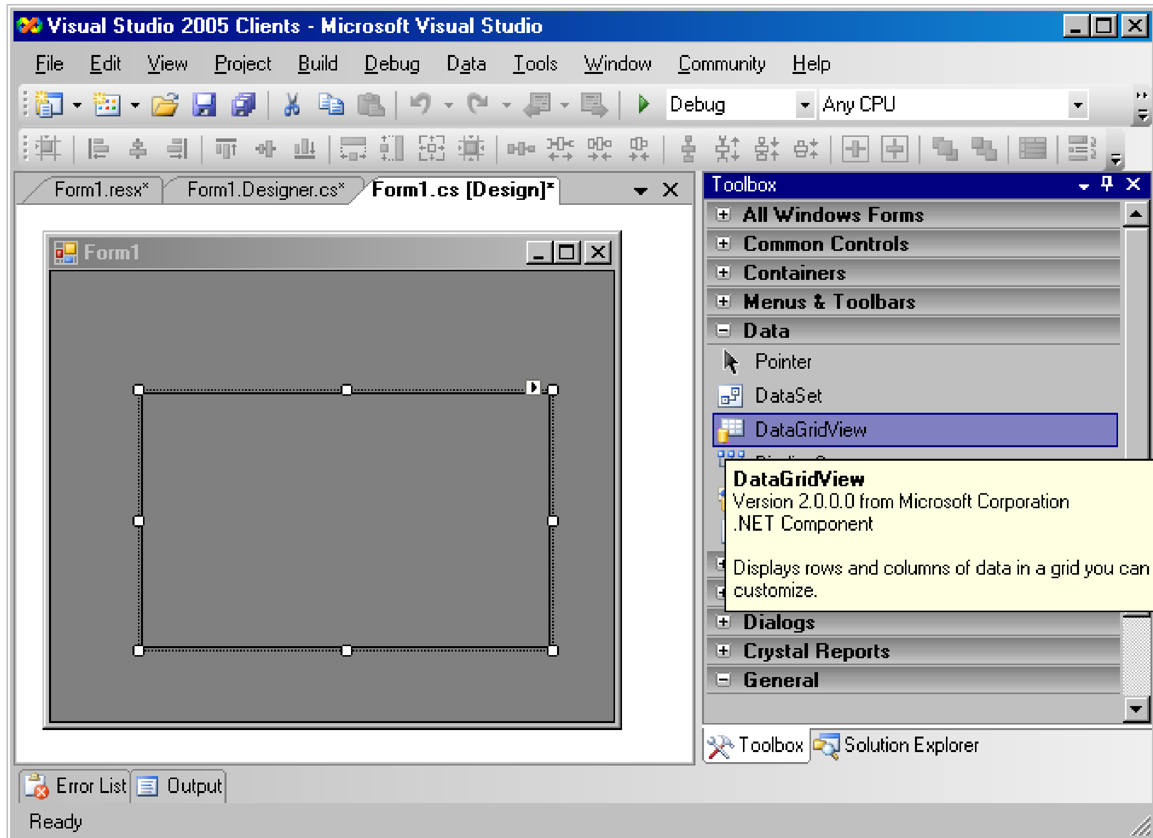
Sample Model

(A) Stateful Calls

(1) Open Visual Studio 2005 and create a new Visual C# Windows Application project, or open the supplied Windows Client project. The following shows a screenshot with the settings for the completed example under the `\Samples\Dot NET Support and Code Libraries\Visual Studio 2005 Clients\` folder.



(2) Open `Form1.cs` in design view, and add a DataGridView control to the dialog.

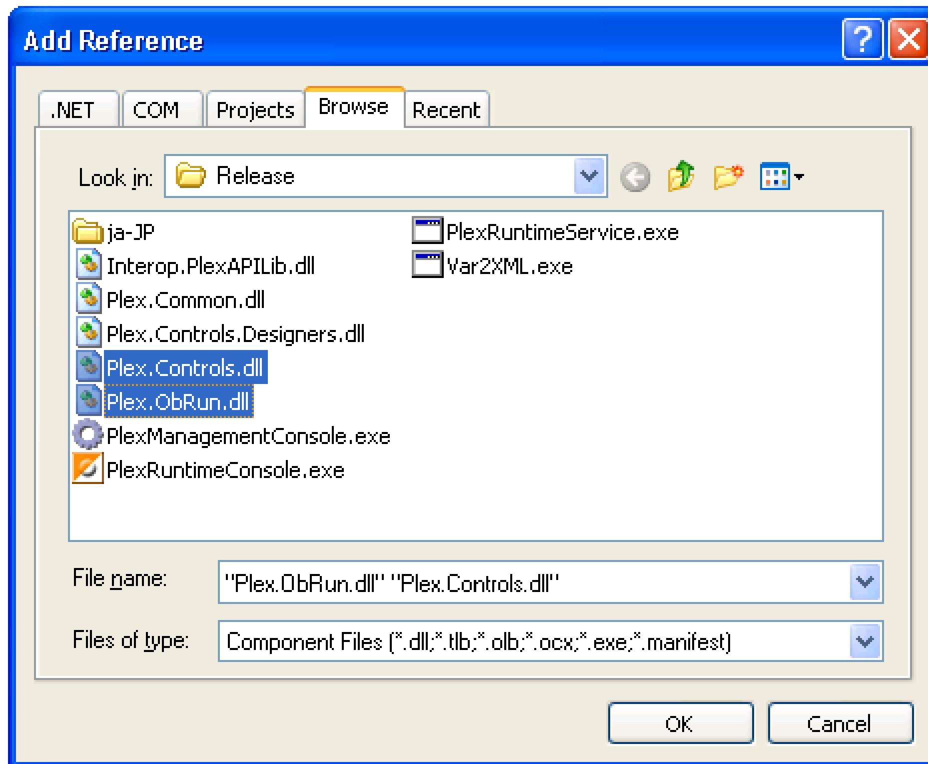


(3) Double-click on the main body of the form to add an event to handle the Load event for Form1. Visual Studio should jump to the code view for Form1.cs file, in a new method called `private void Form1_Load(object sender, EventArgs e)`.

(4) Select **Project→Add Reference…** from the main menu. From the Browse tab, navigate to where the Plex .NET Runtime is located (this should be under the `\Ob.NET\bin\Release\` directory). Add `Plex.ObRun.dll` and `Plex.Controls.dll` as references to your project.

.NET Support and Code Libraries

Sample Model



(5) Add the following code at the start of Form1.cs after the `using` sections for the .NET runtime.

```
using ObRun.ObMain;
using ObRun.ObUtils;
```

(6) Add the following code to the method `private void Form1_Load(object sender, EventArgs e)`:

```
DataSet dsInput;          // A DataSet to hold the input parameters.
DataSet dsOutput;         // A DataSet to hold the output parameters.
DataSet dsOutputGrid;     // A DataSet to hold the data to be displayed on the grid.

// Create a new Plex .NET Runtime instance.
ObApplicationUser app = new ObApplicationUser();

// Initialize dsInput with:
//       - A DataTable that is initialized with the default
//         CallInfo structure for the call.
//       - DataTables that contain each of the input/dual variables
//         used for the call.
dsInput = ObUserApi.getInputParmDataSet("Order.HeaderBlockFetch_ObIn");

// Set up any additional call information in the CallInfo DataTable.
dsInput.Tables["CallInfo"].Rows[0]["Environment"] = "Default";

// Set up any additional parameters for the call in the appropriate
// DataTable instances.
// Note the use of 'In' to prefix the table name.
dsInput.Tables["InHeaderBlockFetch_Position"].Rows[0]["OrderNumber"] = 0;
```

```
dsInput.Tables["InHeaderBlockFetch_Control"].Rows[0]["S5trh2n"] = "Y";

// Create the schema for the accumulated dsOutputGrid.
// Note: We could have waited until dsOutput was populated and then cloned
//       it's schema, but this shows the use of another Plex runtime API.
dsOutputGrid = ObUserApi.getObVariableGroupXAsDataSet("Order.HeaderBlockFetch_ObOut");

// The FetchedData output MOV is returned initialized, so clear it prior to use.
dsOutputGrid.Tables["OutHeaderBlockFetch_FetchedData"].Clear();

// Set the DataGridView.DataSource object to the FetchedData MOV.
// FetchedData is one of the output variables returned in dsOutput as
// a DataTable object.
// Note the use of 'Out' to prefix the table name.
String returnedStatus = " ";
Int32 rowsFetched = 0;
while (returnedStatus == " ")
{
    // Call the Plex function via the Plex ObCallManager.obCallFunction() method.
    dsOutput = app.M_ObCallMgr.obCallFunction(dsInput);

    // Set up the returnedSatus and rowsFetched values passed back.
    returnedStatus = dsOutput.Tables["PlexSystem"].Rows[0]["Returned"].ToString();
    rowsFetched =
Int32.Parse(dsOutput.Tables["InHeaderBlockFetch_Control"].Rows[0]["S5trh30"].ToString());

    if (rowsFetched > 0)
    {
        // Use the ImportRow method to copy from dsOutput to dsOutputGrid.
        for (int i = 0; i < rowsFetched; ++i)
        {
            DataRow dr = dsOutput.Tables["OutHeaderBlockFetch_FetchedData"].Rows[i];
            dsOutputGrid.Tables["OutHeaderBlockFetch_FetchedData"].ImportRow(dr);
        }
    }

    // Set up the parameter DataSets for the next call.
    dsInput = ObUserApi.getInputParmDataSet("Order.HeaderBlockFetch_ObIn");
    dsInput.Tables["CallInfo"].Rows[0]["Environment"] = "Default";
    dsInput.Tables["InHeaderBlockFetch_Control"].Rows[0]["S5trh2n"] = "N";
    dsOutput.Clear();
}
    // Set the DataGrid DataSource equal to the DataTable built from the
    // Blockfetch call.
    dataGridView1.DataSource = dsOutputGrid.Tables["OutHeaderBlockFetch_FetchedData"];
```

(7) Compile the project by selecting **Build→Build Solution**.

(8) Before running the example, you need to have Plex .NET Runtime configuration information in the configuration file associated with the Windows Forms executable. The simplest way to do this is to copy the configuration file you used to run the example in section (2) *Running a Simple Client-Server Application*.

Copy the file `\Ob.NET\bin\Release\PlexRuntimeConsole.exe.config` to the location where your client executable will be executed, and change its name to `Windows Client.exe.config`, as shown in the following diagram.

.NET Support and Code Libraries

Sample Model

Name	Size	Type	Date Modified
ja-JP		File Folder	4/4/2007 5:33 PM
Interop.PlexAPILib.dll	11 KB	Application Extension	3/30/2007 3:41 PM
Plex.Common.dll	76 KB	Applica	
Plex.Controls.dll	64 KB	Applica	
Plex.ObRun.dll	248 KB	Applica	
Windows Client.exe	20 KB	Applica	
Windows Client.exe.config	7 KB	XML Co	
Windows Client.pdb	22 KB	Program Debug Database	4/4/2007 5:33 PM
Windows Client.vshost.exe	6 KB	Application	9/23/2005 6:56 AM
Windows Client.vshost.exe.config	7 KB	XML Configuration File	4/4/2007 6:09 PM

`Windows Client.exe.config` is a copy of the .NET Runtime Console .config file used in the previous example. It is used to locate the class files generated for the Plex .NET Server Application.

(9) Run the example by selecting **Debug→Start Debugging**. You should see a Windows form appear, loaded with the Order Header data that you entered in the first example.

Form1

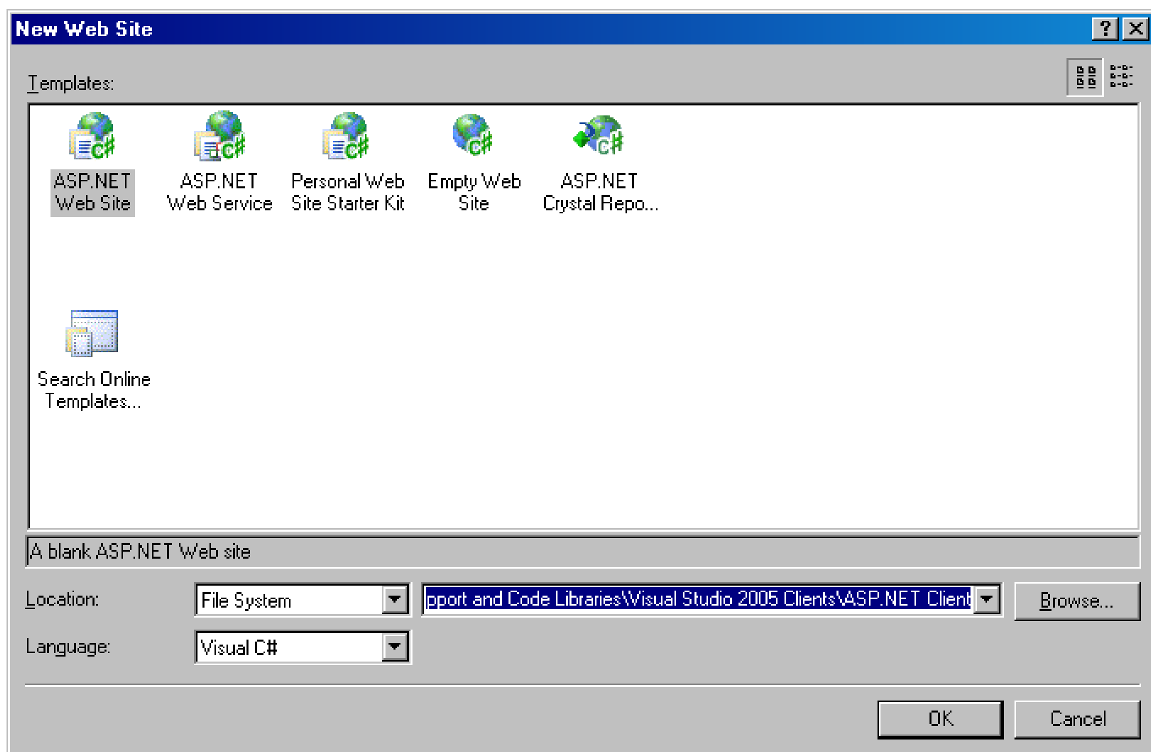
ORDERNUMBER	CUSTOMERNAME	ORDERDATE
1	Rob's Records	6/1/2006
2	Kiyoshi's Lamp Co.	6/5/2006
3	Paul's Boutique	7/2/2006
4	Olga's Dolma Co.	7/6/2006
5	Franks Franks Co.	7/9/2006
6	Cheeky's FX Corp.	7/10/2007
7	The Book Compa...	7/15/2006
8	Donkey Custard ...	7/25/2006

Order.Header.BlockFetch recalled here by C# logic until EOV is reached.

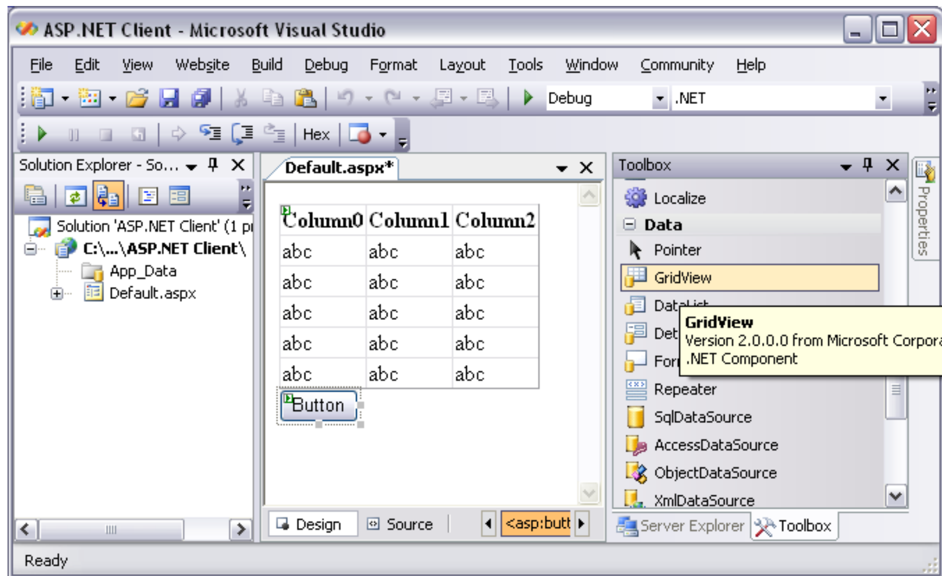
(B) Stateless Calls

(1) Open Visual Studio 2005 and create a new web site by selecting **File→New→Web Site…**. Call the web site 'ASP.NET Client', and place it on the local file system under the `Visual Studio 2005 Clients` folder. Choose a language of C# for the code behind pages associated with the web application.

The following shows a screenshot with the completed settings for the example under the `\Samples\Dot NET Support and Code Libraries\Visual Studio 2005 Clients\` folder.

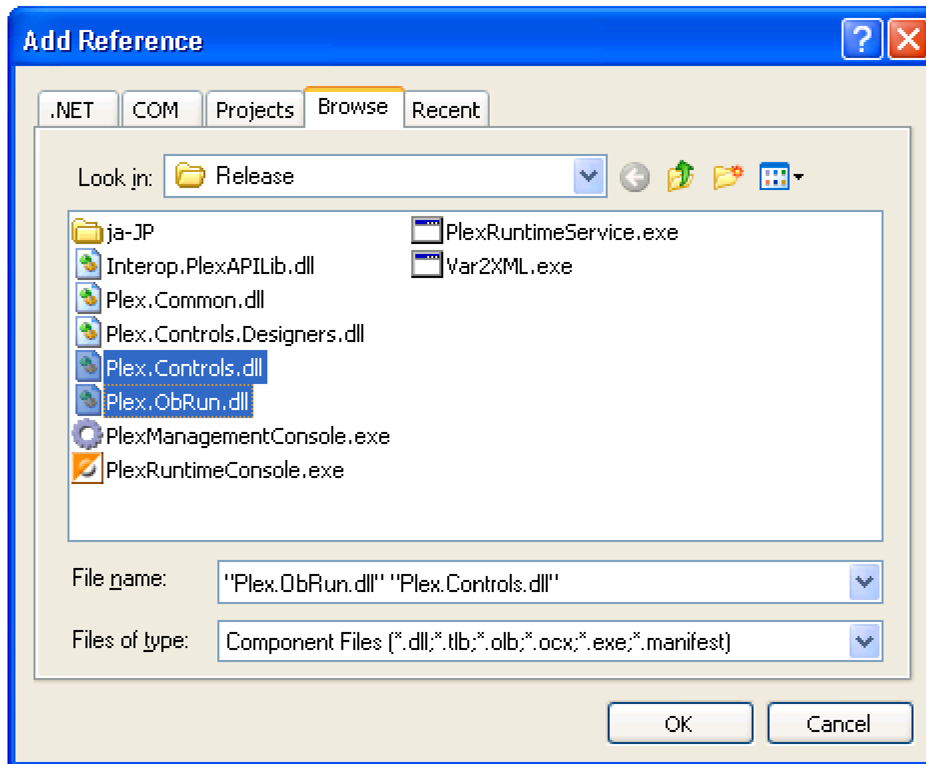


(2) Open `Default.aspx` in design view, and add a GridView and Button web control to the dialog.



(3) Double-click on the newly added button on the page to add an event to handle the Click event. Visual Studio should jump to the code-behind source associated with the aspx page, called `Default.aspx.cs`, and be positioned on a new method called `protected void Button1_Click(object sender, EventArgs e)`.

(4) Select **Web Site→Add Reference…** from the main menu. From the Browse tab, navigate to where the Plex .NET Runtime is located (this should be under the `\Ob.NET\bin\Release\` directory). Add `Plex.ObRun.dll` and `Plex.Controls.dll` as references to your project.



(5) Add the following code at the start of `Default.aspx.cs` after the `using` sections for the .NET runtime.

```
using ObRun.ObMain;
using ObRun.ObUtils;
```

(6) Add the following code to the method `protected void Button1_Click(object sender, EventArgs e)`:

```
DataSet dsInput;          // A DataSet to hold the input parameters.
DataSet dsOutput;         // A DataSet to hold the output parameters.

// Create a new Plex .NET Runtime instance.
ObApplicationUser app = new ObApplicationUser();

// Initialize dsInput with:
//       - A DataTable that is initialized with the default
//         CallInfo structure for the call.
//       - DataTables that contain each of the input/dual variables
//         used for the call.
dsInput = ObUserApi.getInputParmDataSet("Order.HeaderStatelessBlockFetch_ObIn");

// Set up any additional call information in the CallInfo DataTable.
dsInput.Tables["CallInfo"].Rows[0]["Environment"] = "Default";

// Set up any additional parameters for the call in the appropriate
// DataTable instances.
// Note the use of 'In' to prefix the table name.
dsInput.Tables["InHeaderStatelessBlockFetch_Position"].Rows[0]["OrderNumber"] = 0;
```

.NET Support and Code Libraries

Sample Model

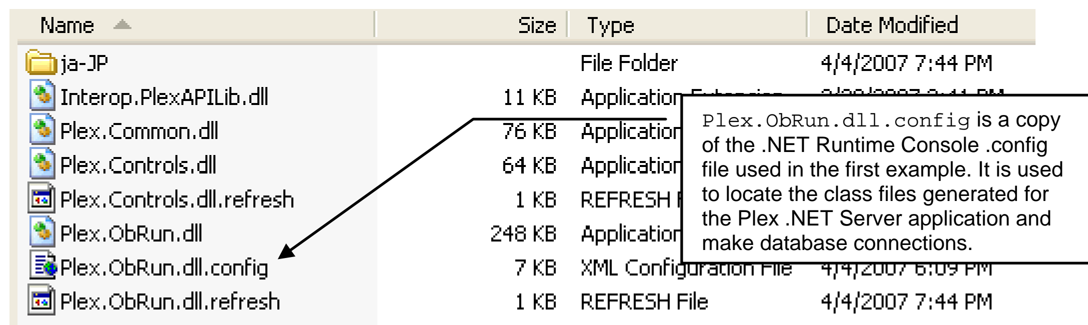
```
dsInput.Tables["InHeaderStatelessBlockFetch_Control"].Rows[0]["S5trh2n"] = "Y";

// Call the Plex function.
dsOutput = ObUserApi.callFunction(dsInput, false);

// Set the DataGridView.DataSource object to the FetchedData MOV.
// FetchedData is one of the output variables returned in dsOutput as
// a DataTable object.
// Note the use of 'Out' to prefix the table name.
GridView1.DataSource = dsOutput.Tables["OutHeaderStatelessBlockFetch_FetchedData"];
GridView1.DataBind();
```

(7) Compile the project by selecting **Build→Build Solution**.

(8) Before running the example, you need to have Plex .NET Runtime configuration information in a configuration file associated with the ASP.NET web application. When running in a web application, the Plex .NET Runtime looks for configuration information in either the `web.config` file associated with the web application, or in a configuration file called `Plex.ObRun.dll.config`.

The simplest way to do this is to copy the configuration file you used to run the example in section (2) *Running a Simple Client-Server Application*, renaming it `Plex.ObRun.dll.config`, and placing it in the directory where the web application will be executing from.

Copy the file `\Ob.NET\bin\Release\PlexRuntimeConsole.exe.config` to the location where your client executable will be executed, and change its name to `Plex.ObRun.dll.config`, as shown in the following diagram.



(9) Run the example by selecting **Debug→Start Debugging**. You should see a Microsoft Internet Explorer window open that contains the button added. Press the button and the method added in step (6) should be called, binding the FetchedData DataRows returned into the GridView control.

.NET Support and Code Libraries

Sample Model

Untitled Page - Microsoft Internet Explorer provided by Com...

http://localhost:1822/A

ORDERNUMBER	CUSTOMERNAME	ORDERDATE
1	Rob's Recordsx	6/1/2006 12:00:00 AM
2	Kiyoshi's Lamp Co.	6/5/2006 12:00:00 AM
3	Paul's Boutique	7/2/2006 12:00:00 AM
4	Olga's Dolma Company	7/6/2006 12:00:00 AM
5	Franks Franks Co.	7/9/2006 12:00:00 AM
6	Cheeky's FX Corp.	7/10/2007 12:00:00 AM
7	The Book Company	7/15/2006 12:00:00 AM
8	Donkey Custard Co.	7/25/2006 12:00:00 AM
0		1/1/0001 12:00:00 AM
0		1/1/0001 12:00:00 AM

Local intranet 100%

.NET Support and Code Libraries

Sample Model

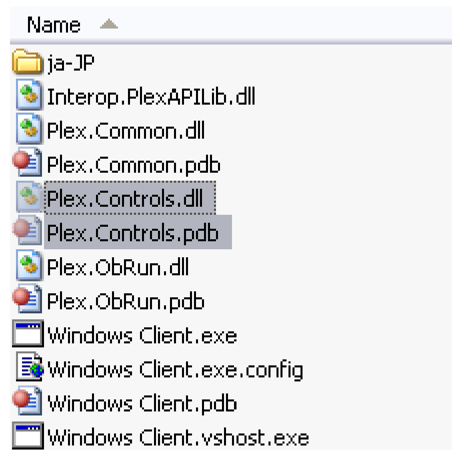
Troubleshooting

1. **Problem:** You receive the following exception at runtime:

   An unhandled exception of type 'System.Configuration.ConfigurationErrorsException' occurred in System.Configuration.dll

   Additional information: An error occurred creating the configuration section handler for Dispatchers/Service_x0020_Dispatcher_x0020_Debug/Environments/Default/Default: Exception has been thrown by the target of an invocation.

   Solution: Make sure the following Plex runtime assemblies are located in the directory where your .NET Forms application is running from. The highlighted `Plex.Controls.dll` files may not be copied by default.

2. **Problem:** You receive the following exception at runtime:

   An unhandled exception of type 'System.NullReferenceException' occurred in Windows Client.exe

   Additional information: Object reference not set to an instance of an object.

   Solution: The `Windows Client.exe.config` or `Plex.ObRun.dll.config` file your .NET application is trying to load does not have the correct configuration information stored within it. Copy the `PlexRuntimeConsole.exe.config` file into the directory where your .NET Forms application is running from, or the current directory where you are debugging your ASP.NET application from and rename it accordingly.

Notes

1. You do **not** need to have the Plex .NET Runtime Service running in order for these examples to work. The Windows Forms application process contains the Plex .NET Runtime instance being used.

.NET Support and Code Libraries

Sample Model

(5) Using Code Library Objects to Create .NET Assemblies

Introduction

For large-scale projects, generating C# source and compiling it into one assembly is less than ideal. If one function changes, then the whole assembly effectively must be changed. Also the size could be prohibitively large; making the application impossible to install and administer.

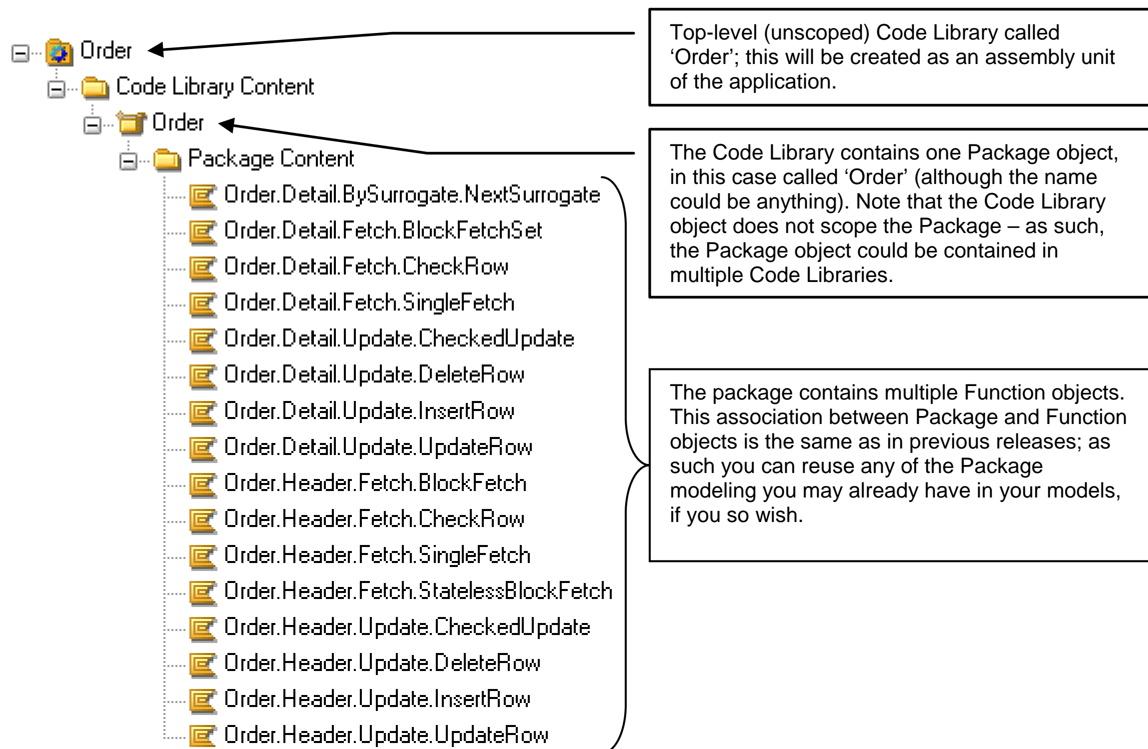
To solve this problem, Plex has introduced the concept of Code Libraries. These objects can be modeled to contain a logical subset of an application; the subsets could be functional units of an application, or could be functions that define a revision of an application.

The following document outlines the steps required in order to create multiple assemblies that contain CA Plex generated C# functions.

(A) Modelling Code Libraries

First, you must create Code Library objects that define the subsets of the application that you want to deploy. The top-level Code Library objects will ultimately get created as assemblies, so the triples that you define against the Code Library reflect the attributes and manifest information that will be created in each assembly.

A Code Library must contain one or more Package objects, which in turn, must contain one or more Function objects. The following shows a typical structure for a Code Library, as seen in the Plex Object Browser.



.NET Support and Code Libraries
Sample Model

Because Plex functions cannot be added directly to the Code Library object, they must first be scoped to Package objects. This can be achieved by using the **PKG contains FNC** triple. The following shows a Package called 'Order' that contains the server functions associated with the Order.Header and Order.Detail entities.

The Package information for a function will define the .NET Namespace into which the classes associated with that function will be generated.

Order	contains	Order.Detail.BySurrogate.NextSurrogate
		Order.Detail.Fetch.BlockFetchSet
		Order.Detail.Fetch.CheckRow
		Order.Detail.Fetch.SingleFetch
		Order.Detail.Update.CheckedUpdate
		Order.Detail.Update.DeleteRow
		Order.Detail.Update.InsertRow
		Order.Detail.Update.UpdateRow
		Order.Header.Fetch.BlockFetch
		Order.Header.Fetch.CheckRow
		Order.Header.Fetch.SingleFetch
		Order.Header.Fetch.StatelessBlockFetch
		Order.Header.Update.CheckedUpdate
		Order.Header.Update.DeleteRow
		Order.Header.Update.InsertRow
		Order.Header.Update.UpdateRow

This Package object is associated with the Code Library by using the **CDL Comprises PKG** triple. By doing this, you can view the contents of the Code Library in the Plex Object Browser.

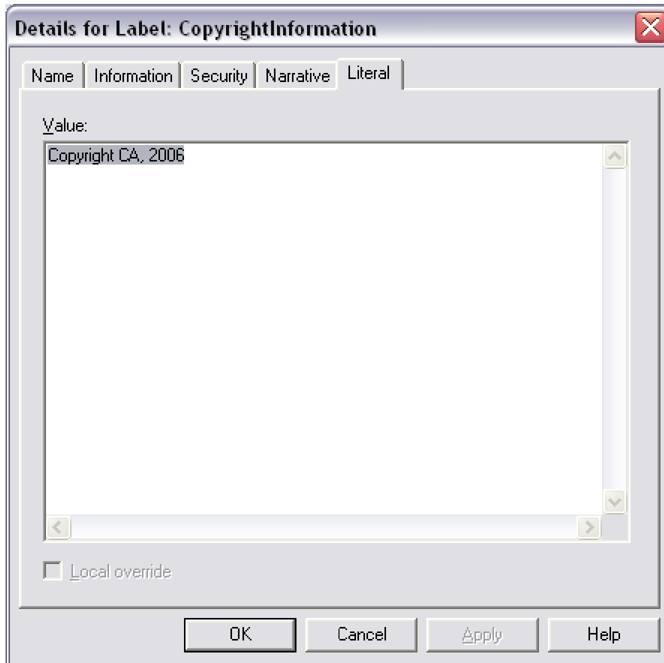
The following shows the triples for a Code Library called Order which is contained in the SALESSYSTEM sample local model.

Order	comprises	Order
	company name	CA
	copyright information	CopyrightInformation
	major version	1
	minor version	0
	trademark information	TrademarkInformation
	product	ProductInformation
	NET type	Assembly
	file name	Order
	impl name	Order
	build number	1
	revision number	0
	language	C#

.NET Support and Code Libraries

Sample Model

There are a number of descriptive label objects in the above list, such as 'Company name', 'Copyright information', 'Trademark information' and 'Product'. These labels should have their values coded into their literal values as follows:



You have now modeled your Code Library and are in a position to generate the source used to create it.

If you have followed the examples right from the start, you do not have to generate any C# code in order to create your assemblies. In the first example, you generated your source code, and then compiled it into the Plex .NET Default Assembly. The information in this generated code is completely independent of any Code Library in which you place it. It is perfectly acceptable to generate, build and test your application using the default assembly created from the generate and build window, and then create your .NET assembly units at a later point in your development cycle – all without having to regenerate a single line of code.

Note: Because Package information is generated into the source for a function in the form of namespace information, you must make sure your any package modeling you wish to do is completed before you generate your application source.

All information specific to the Code Library objects being created is generated and compiled into the target assemblies by the new CA Plex Code Library Deployment Wizard.

.NET Support and Code Libraries
Sample Model

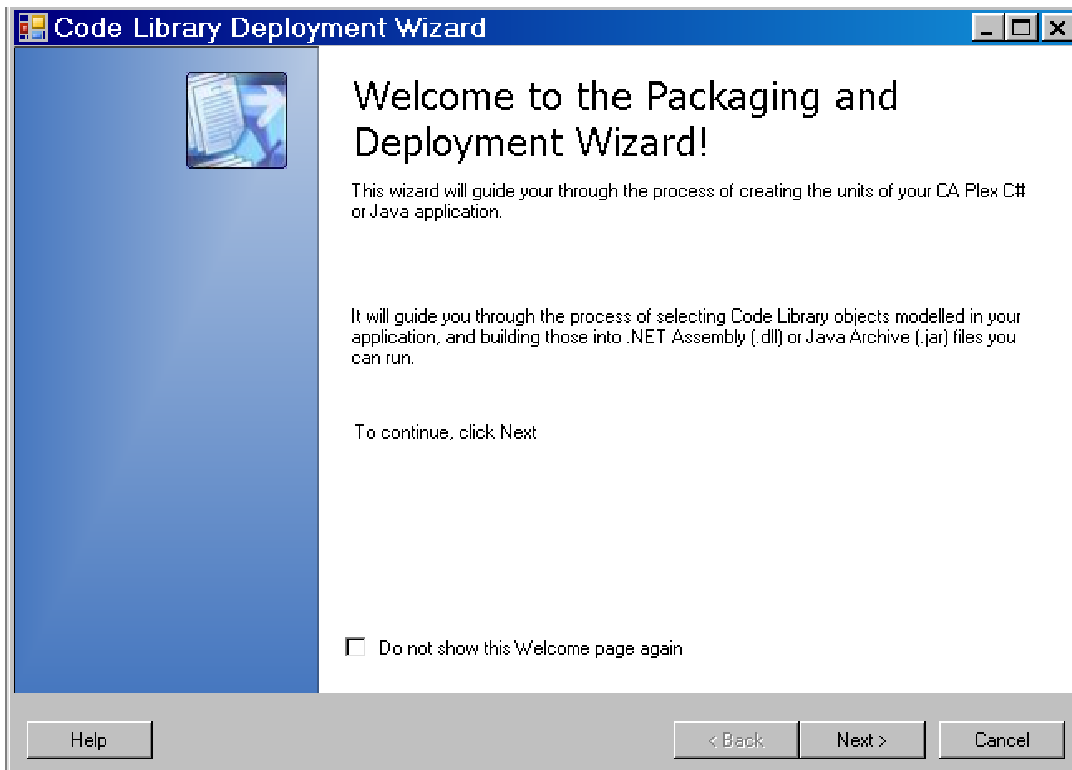
(B) Packaging and Deploying a Code Library

In order to create an Assembly from a Code Library, you need to use a new Plex tool called the Plex Code Library Deployment Wizard.

Note 1: Because this tool uses the Plex COM API interface you must make sure that the PlexAPI interface is registered on your target machine. This should be taken care of by the Plex install process, but if you need to perform this step manually, please refer to the Troubleshooting section at the end of this document.

Note 2: To launch the Wizard, select Code Library Wizard menu item from the Tools menu.

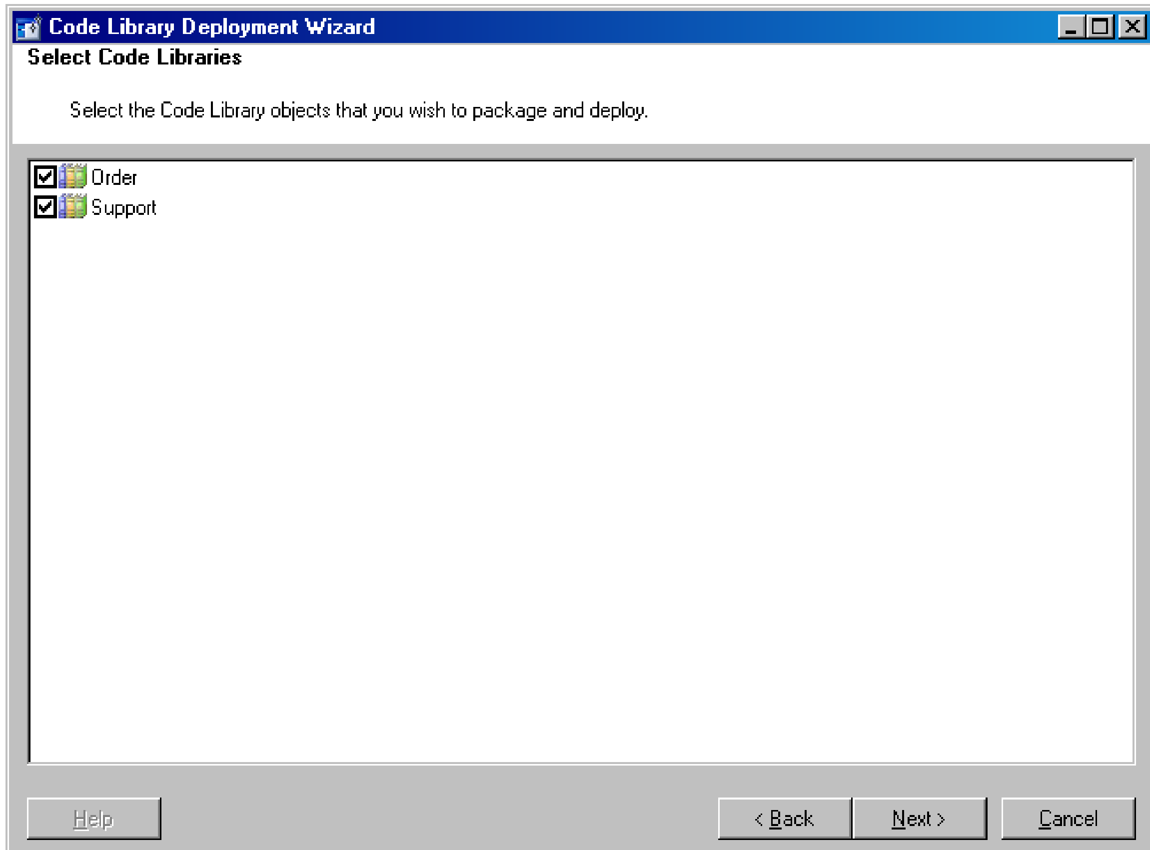
Starting the wizard should show the welcome screen, select 'Next' to start the packaging and deployment process.



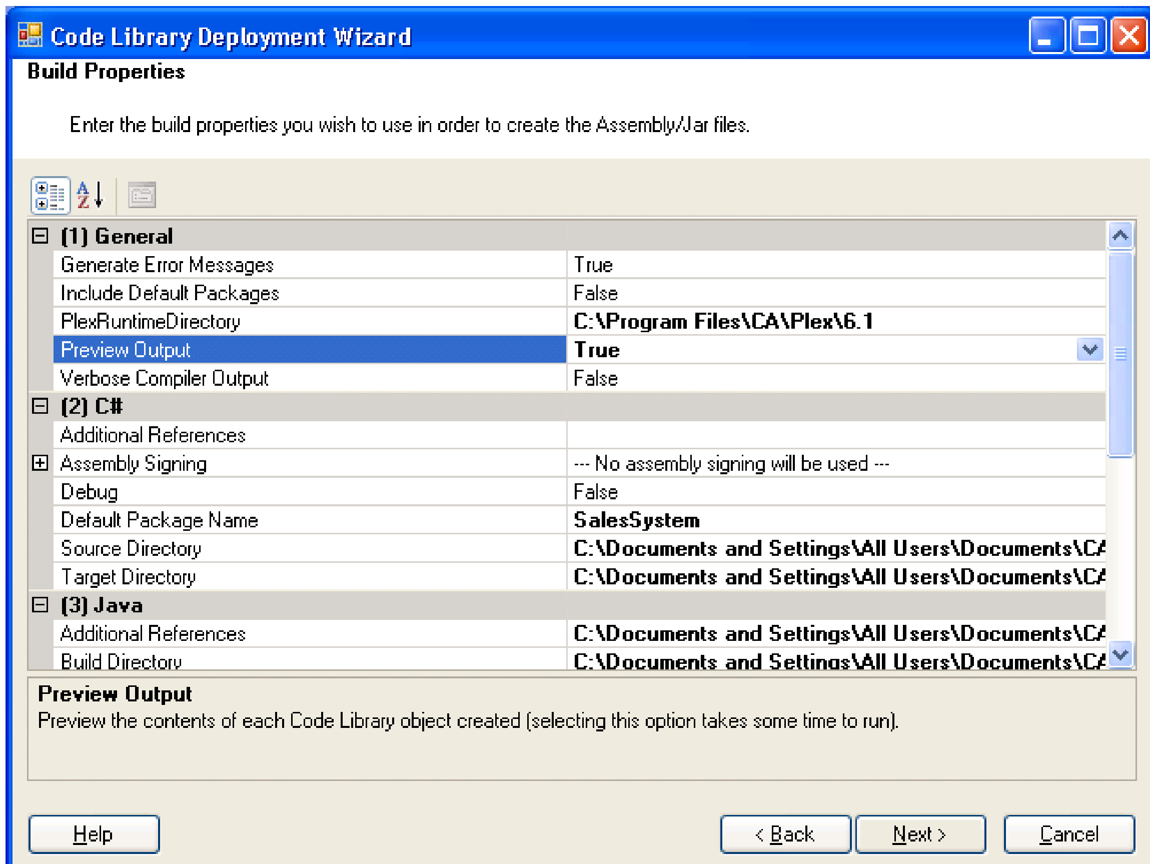
The next screen lists the unscoped Code Library objects that are present in your model, and are have enough information to allow the creation of an Assembly.

Select the Code Library objects that you wish to deploy and select 'Next'.

.NET Support and Code Libraries
Sample Model



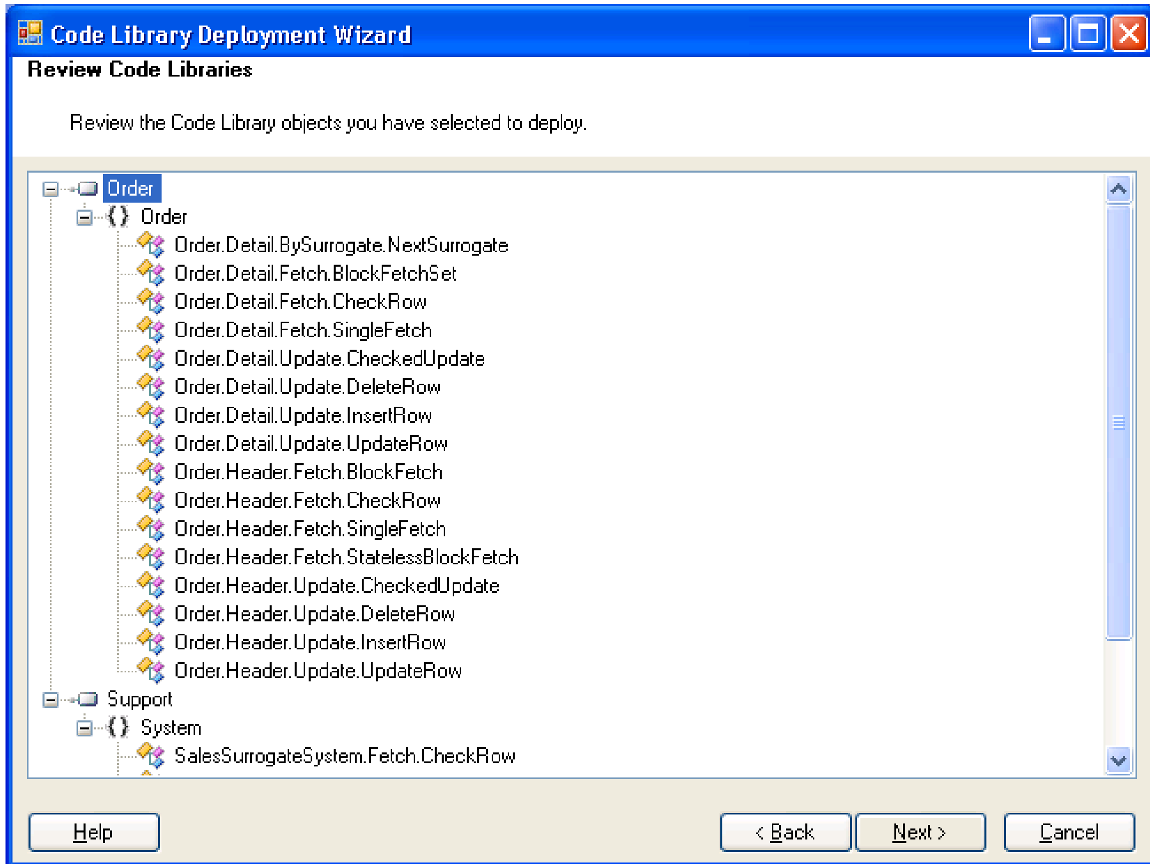
The next screen shows the options that the Wizard will use when creating the assemblies. These options are defaulted from the Generate and Build options from the current local model, but can be overridden if required.



If you set the Preview Output option to True (the default is False), then the next screen allows you to review the Code Libraries you have selected for deployment. You can drill down into the libraries and see their packages and functions that will be contained within them.

.NET Support and Code Libraries

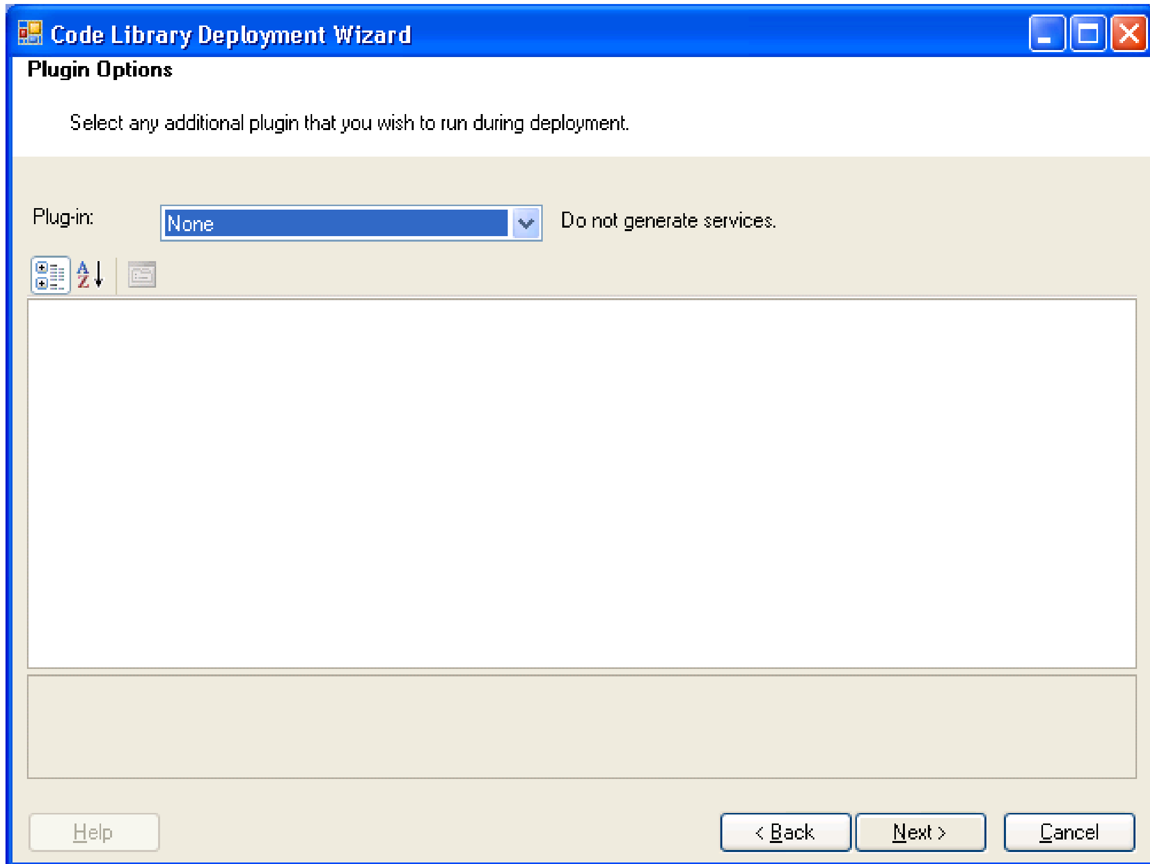
Sample Model



Selecting ‘Next’ opens the ‘Plugin Options’ screen that does not require any changes.

.NET Support and Code Libraries

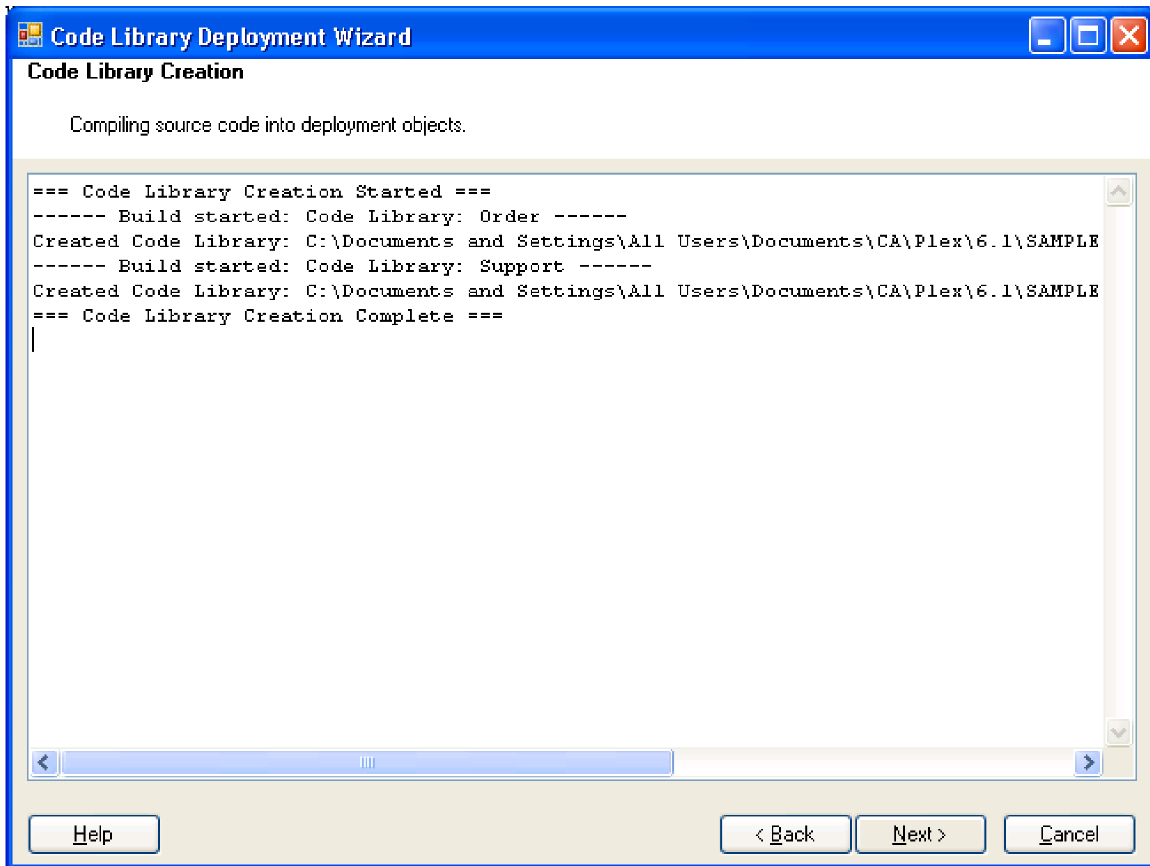
Sample Model



Selecting 'Next' will start the assembly build process.

The next screen shows the output of the build process; you should see that your assemblies are created without any errors or warnings.

You can only select 'Next' when the build has completed.

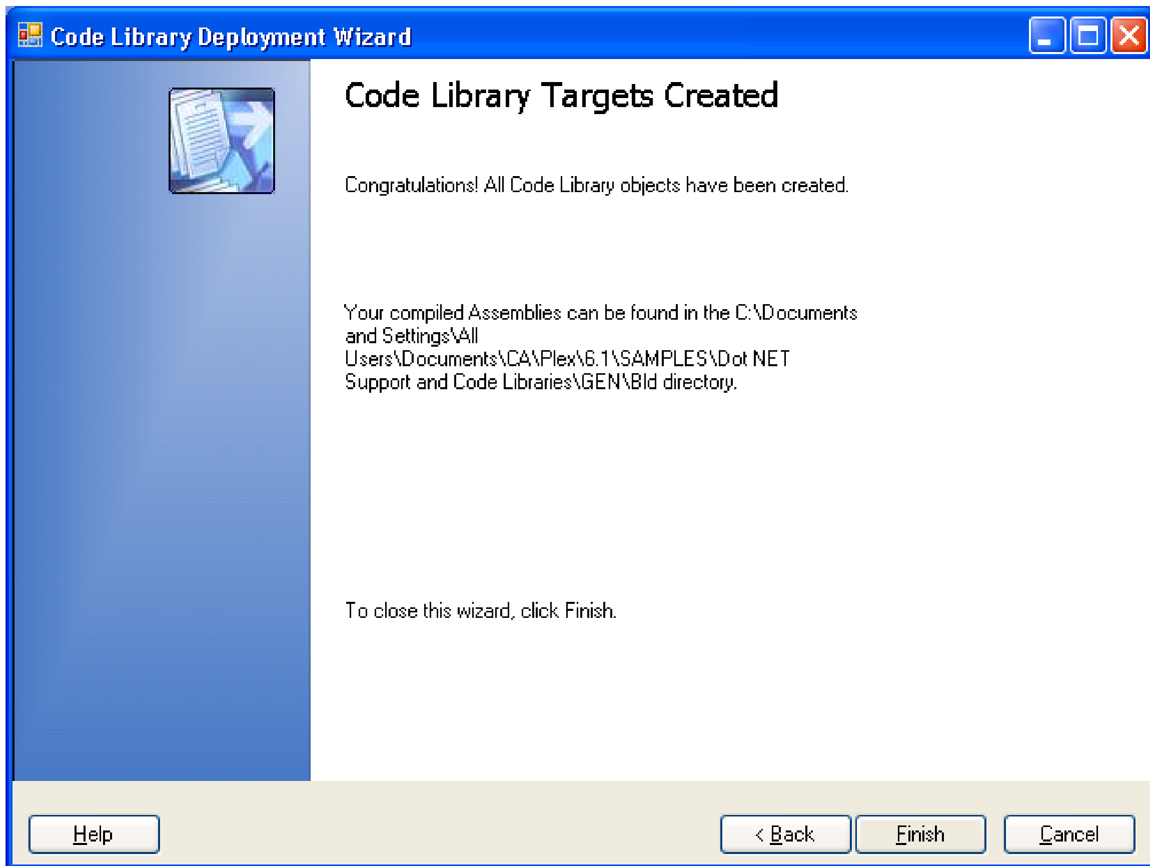


Congratulations - You should have successfully created an assembly that contains the required Plex functions.

Select ‘Finish’ to close the Code Library Deployment Wizard.

.NET Support and Code Libraries

Sample Model



But what actually got created? And how do we use them?

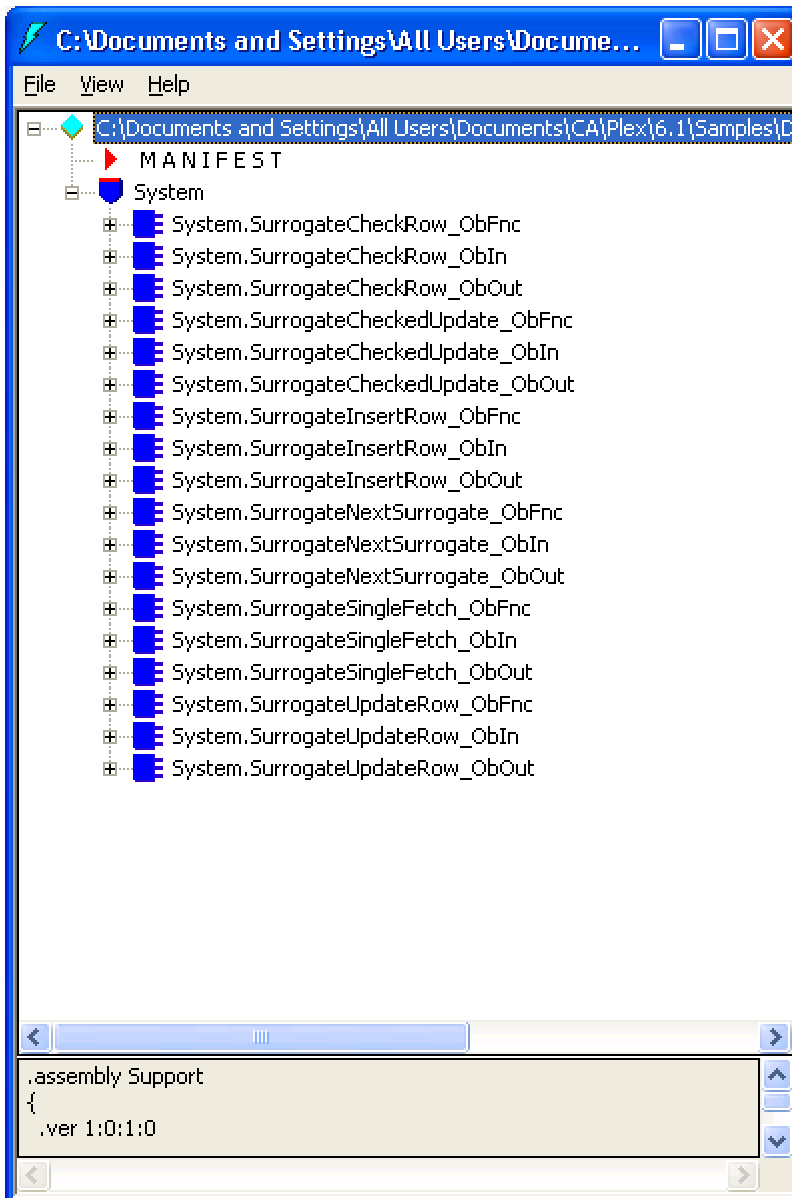
The directory in which the Code Library Assemblies reside is based on the target directory that specified in the build options (step 3 of the wizard). The actual directory will depend on whether you selected Release or Debug as your build option.

These are named after the **CDL File name NME** property you specified when you modeled your Code Library object.

.NET Support and Code Libraries

Sample Model

You can see the contents of the assembly using the Il Dissassembler tool shipped with the .NET Framework.



In order to use the assemblies with the Plex runtime, you need to specify the names of the assemblies in the **AssemblyList** entry in the configuration file for the .NET Runtime you are using.

Multiple entries can be made in this field, and at runtime, it is searched in a similar manner to a PATH variable or a Java Class Path. Also, in a similar manner to the PATH and Java Class Path entries, multiple entries are separated by semicolons.

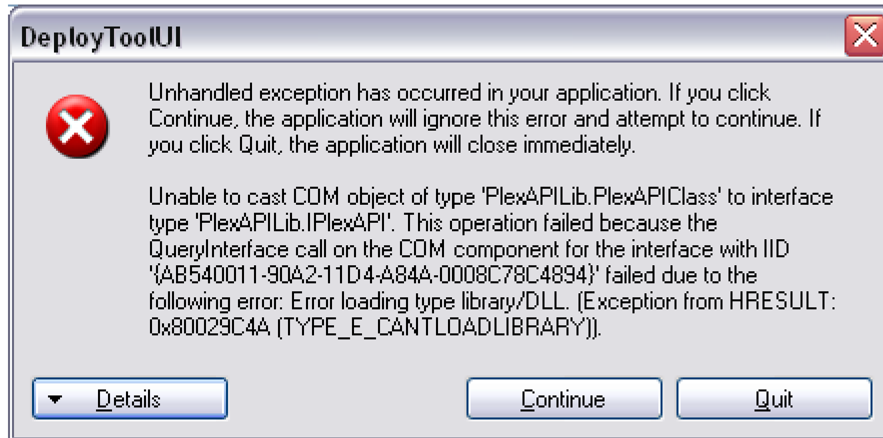
.NET Support and Code Libraries

Sample Model

General	
AssemblyList	T Support and Code Libraries\Gen\Bld\Support.dll;C:\Pro…
Culture	en-US
Preferences	

Troubleshooting

(1) **Problem:** You receive the following error when launching the Code Library Deployment Wizard:



Solution: Register the PlexAPI COM interface. To do this, go to a command line, change into your Plex installation directory and enter the following command:

```
Plex.exe /RegServer
```