Table of Contents

[1. Objectives 3](#_Toc296851443)

[2. Intended audience 4](#_Toc296851444)

[3. Getting started 5](#_Toc296851445)

[3.1. Initial prequirements 5](#_Toc296851446)

[3.2. Creating new project 6](#_Toc296851447)

[3.3. Implementing the IPlayEngine interface 11](#_Toc296851448)

[4. Implementing basic routines for an modern music service. 13](#_Toc296851449)

[4.1. Creating the playback object 14](#_Toc296851450)

[4.2. Understanding core concepts of music queries 15](#_Toc296851451)

[4.2.1 Queries (URL) by metadata 15](#_Toc296851452)

[4.2.2. Query by service 16](#_Toc296851453)

[4.3. Implementing music queries for comusic model and basic playback workarounds 17](#_Toc296851454)

[4.3.1. Implementing the namespace definition 17](#_Toc296851455)

[4.3.2. Implementing the query namespace 17](#_Toc296851456)

[4.3.3. Handling load query 18](#_Toc296851457)

[4.3.4. Handling stop, pause and position of the song 18](#_Toc296851458)

[4.3.5. Handling position changes 21](#_Toc296851459)

[4.4. Handling media lookup 22](#_Toc296851460)

[4.4.1. Handling the find method 23](#_Toc296851461)

[4.4.2. Handling RawFind() Method 24](#_Toc296851462)

[4.5. Implementing authentication routines for the service control 25](#_Toc296851463)

[4.5.1. Handling user authentication 25](#_Toc296851464)

[5. Deploying media services 28](#_Toc296851465)

[6. References 30](#_Toc296851466)

# Objectives

This manual provides an step-by-step walkthrough of creating an sample media provider (also called IPlayEngine) for mediachrome to demonstrate how 3rd party developers can integrate their own music services for use with MediaChrome API.

# Intended audience

This manual assumes the developer has an fundamental understanding of the fundamentals of computer programming, application development and .NET framework. We use C# as it were the primary language the author use when developing MediaChrome, but plugins can also be developed with any CIL compatible language.

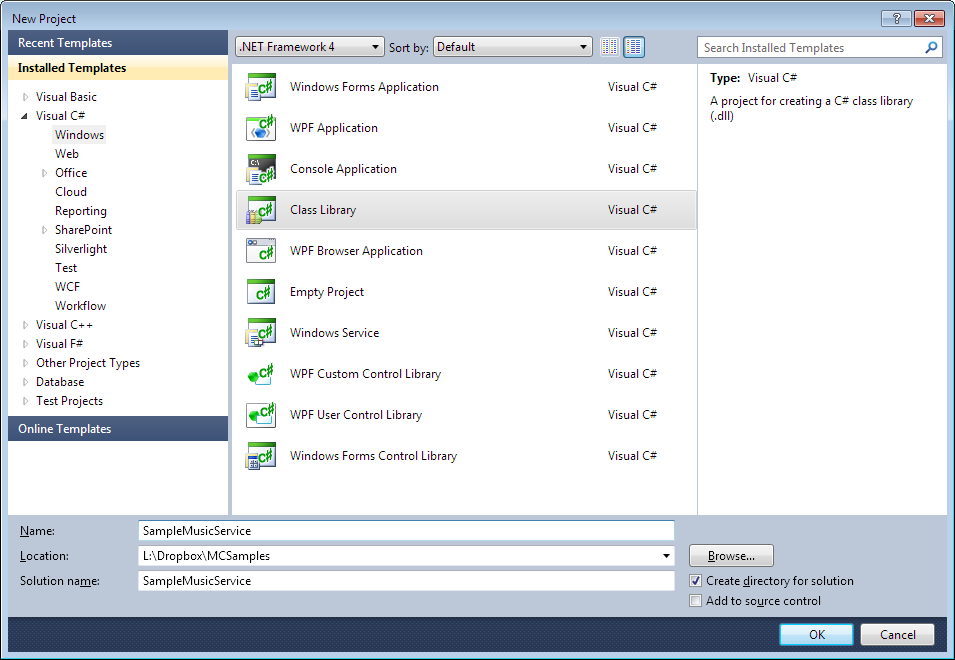
# Getting started

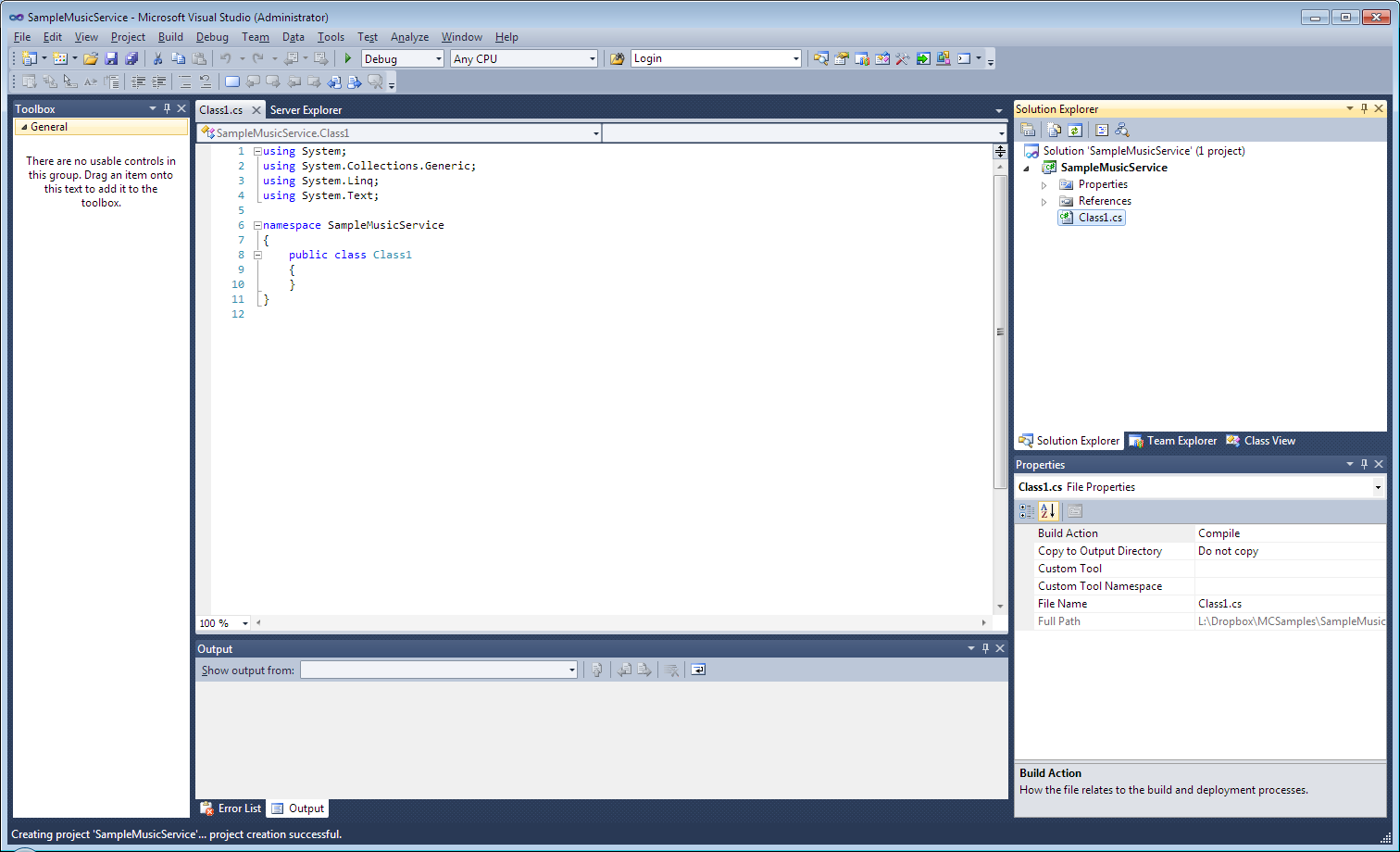
## Initial prequirements

To create an MediaChrome plugin, you should first download an assembly called ’MediaChrome’, which will be handed out to developers. It will soon be available for public download from www.mediachrome.me .

## Creating new project

Create an new empty project on File >> New Project. Choose ’Class Library’ in the CLI language, (ours is Visual C#) in the template view. We call the project **SampleMusicService** and is stored in our folder ”MCSamples ” (MediaChrome Samples).

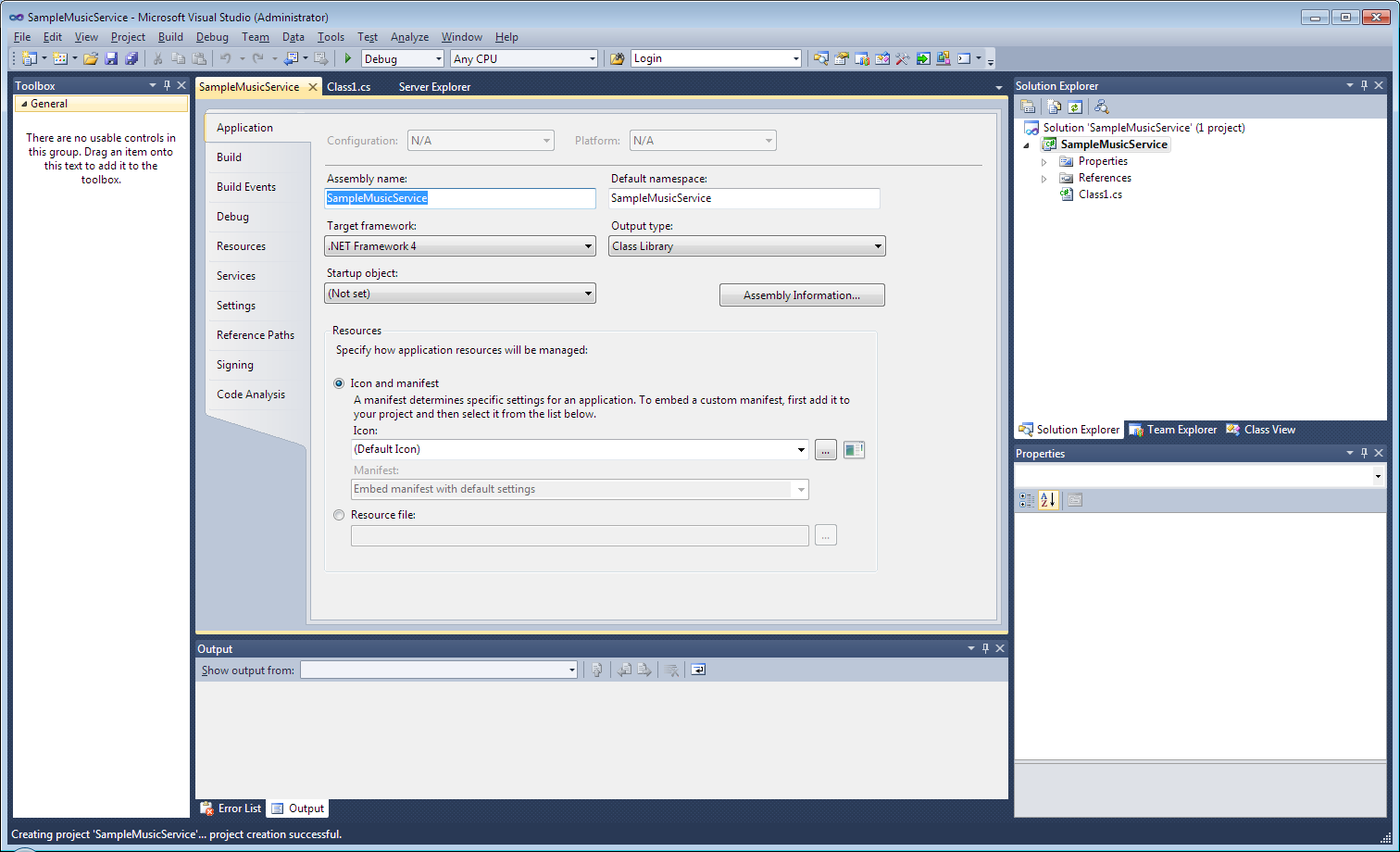


Press OK. Visual studio creates now an class library. You will soon see this:

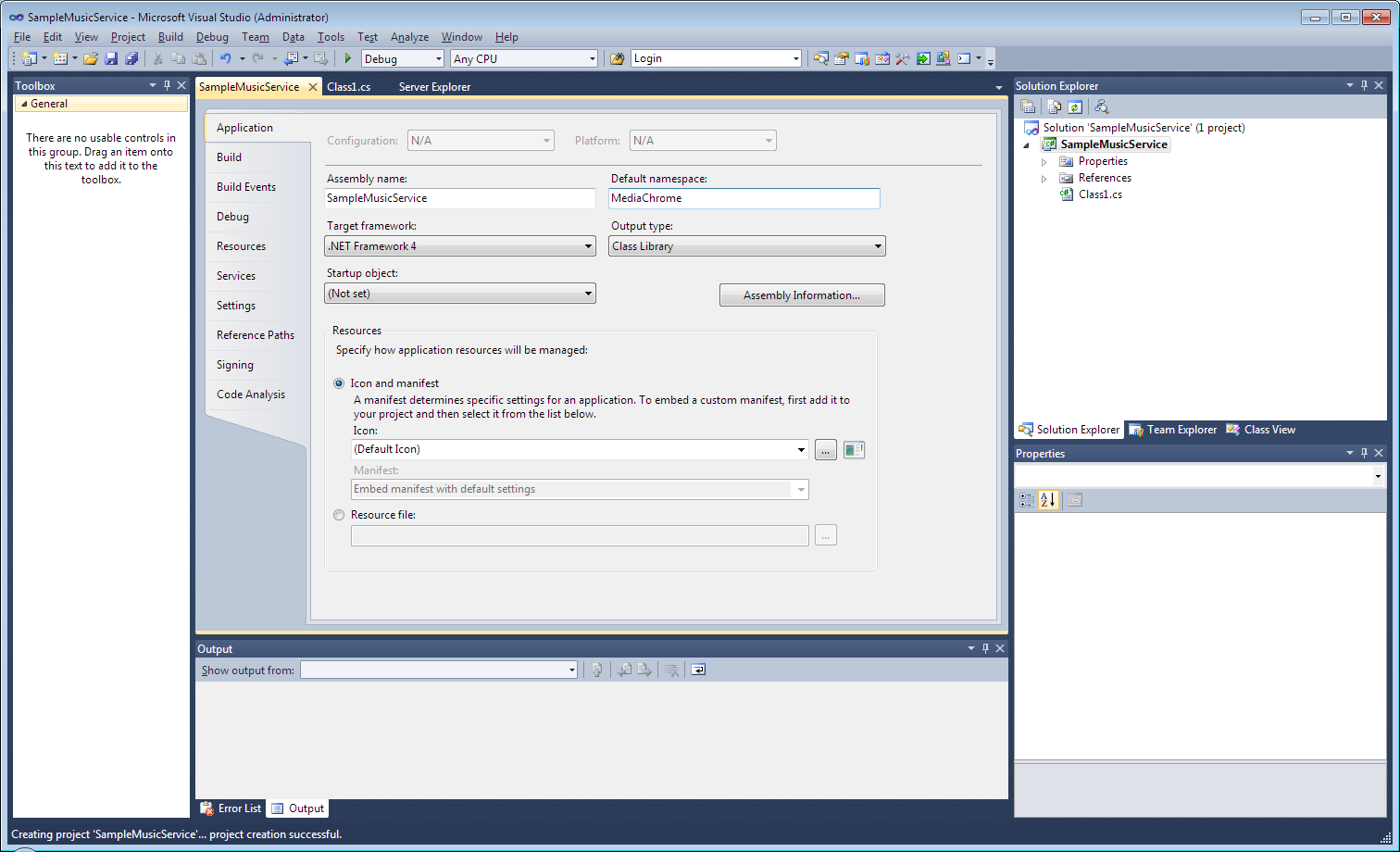
The first thing we’re going to do, to prevent further complications is to rename the namespace of the project to ’MediaChrome’. This is mandatory, since all MediaChrome plugin must have ’MediaChrome’ as initial namespace.

Before we rename the namespace in the code editor of the class file, we must change the root namespace of the application. Right-click on the project icon, **SampleMusicService** and select ’Properties’.

In this window, the field labeled ’Default namespace:’ to ’MediaChrome’. Press CTRL+S to save the changes.

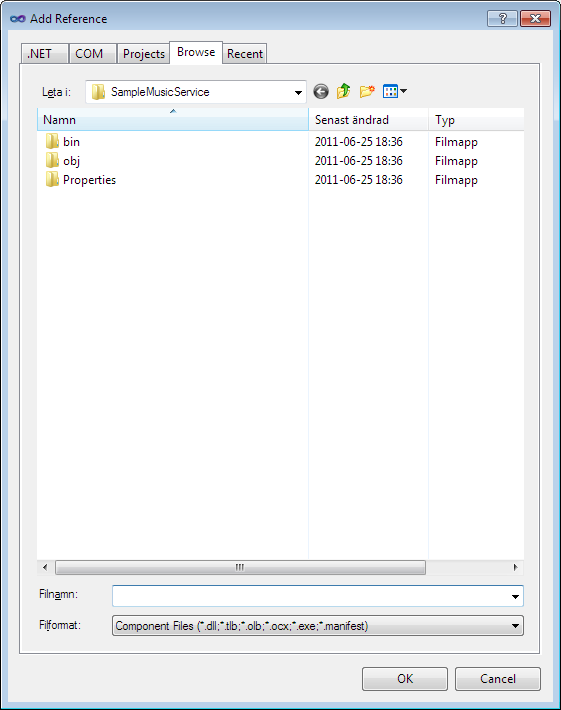


It should appear

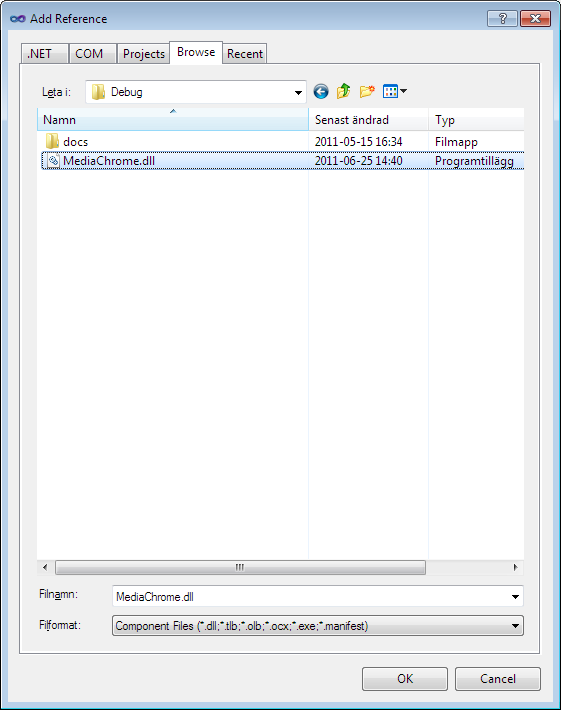


Then save (CTRL+S) and leave the settings by closing the page (pressing X).

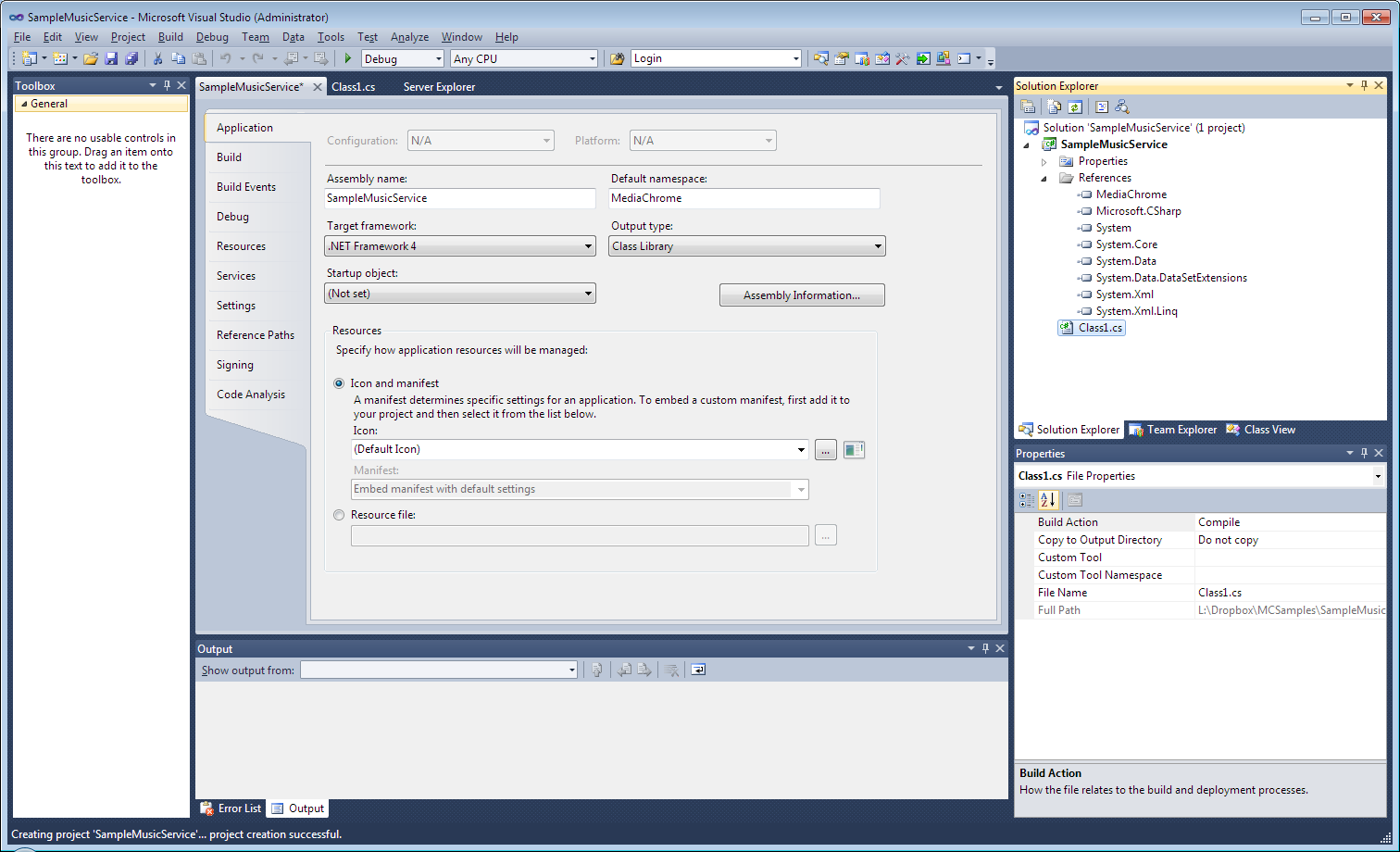
Now, right click on the application folder ’References’ in the Solution Explorer, and select ’Add Reference’.



Click the ’Browse’ tab and locate the MediaChrome dll assembly.



Click ’OK’.



Now, you see MediaChrome is referenced. This is all we need to do before begin our coding.

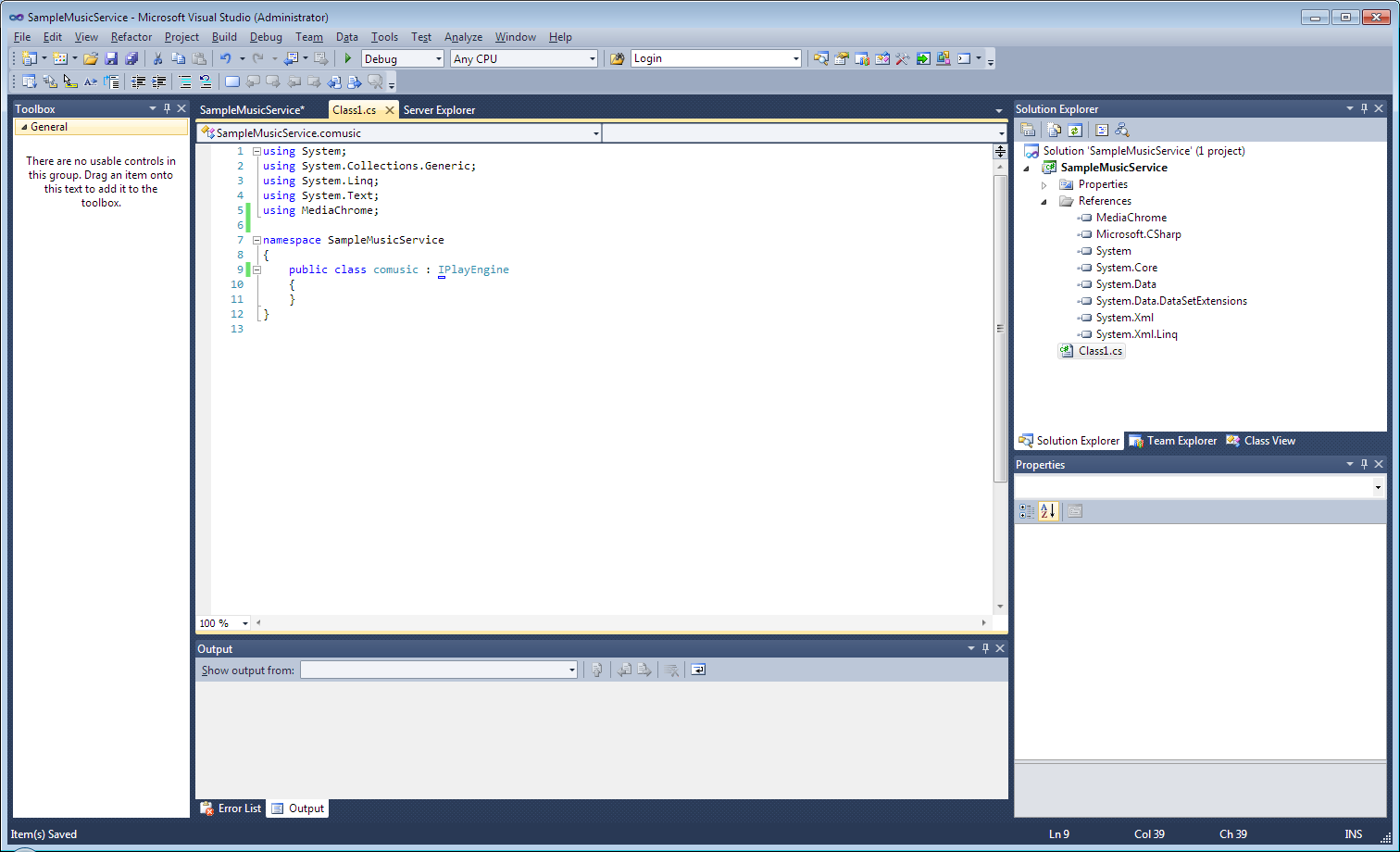
## Implementing the IPlayEngine interface

Now, we need to implement the IPlayEngine interface. As we now have an reference to MediaChrome’s assemblies, we now can access the objects MediaChrome offer us.

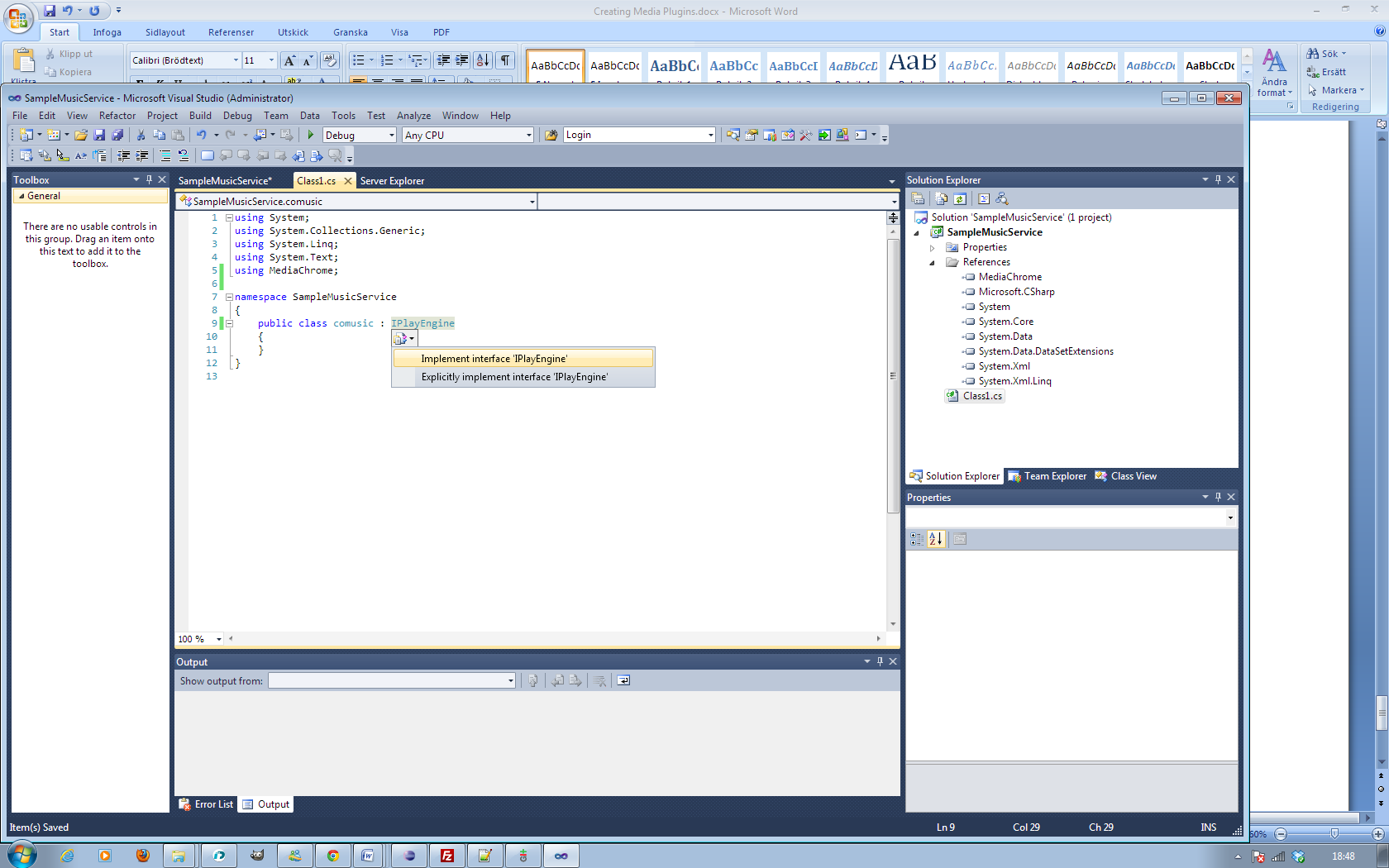
We must now define an **namespace** for the media provider, which must be unique to each other and only consist of an set of a-zA-z characters, with no number preceeding the first characters. It should be completely in lowercase. For our example, we define our service as **comusic** and thus use **comusic** as our primary namespace. It also works as an unique alphanumeric key for each plugin.

Add **MediaChrome** to the list of “usings”. It must be done, even if our namespace is the same.

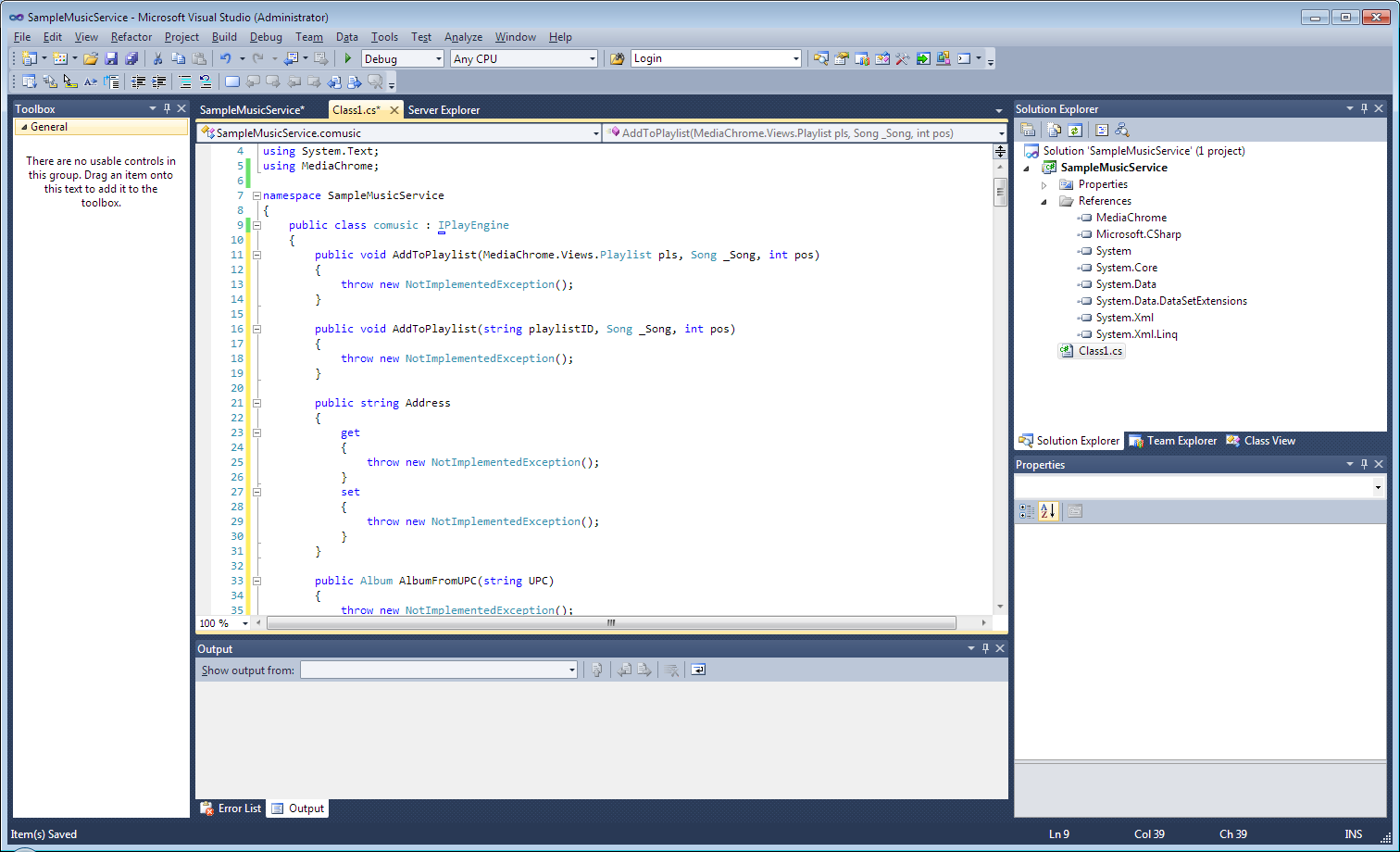
So now, rename the class name to just **comusic**. Then declare the class are implementing IPlayEngine so it looks following:



Then, implement the interface by raising the menu in the little blue tiny thing and select ‘Implement interface’ IPlayEngine.



Visual Studio now automatically adds stubs to all the methods at the interface. Note that the developer must override (also called implement) every method in an interface. It will now look like the next shot:



Now we can begin implement the plugin as we need.

**Be aware!** If any method will retain the NotImplementedException, horrible bugs may happen for the end user, as upon invocation of any of these methods, MediaChrome will crash.

# Implementing basic routines for an modern music service.

The goal of this service is adapter the services basic routines to be handled from **MediaChrome’s** shell. All music service allows people to find, discover and stream music. But you must also be able to handle user authentication.

Basic methods an music service should implement is following:

Start and stop playing (streaming) of an particular song.

Find songs by track/artist/album queries,

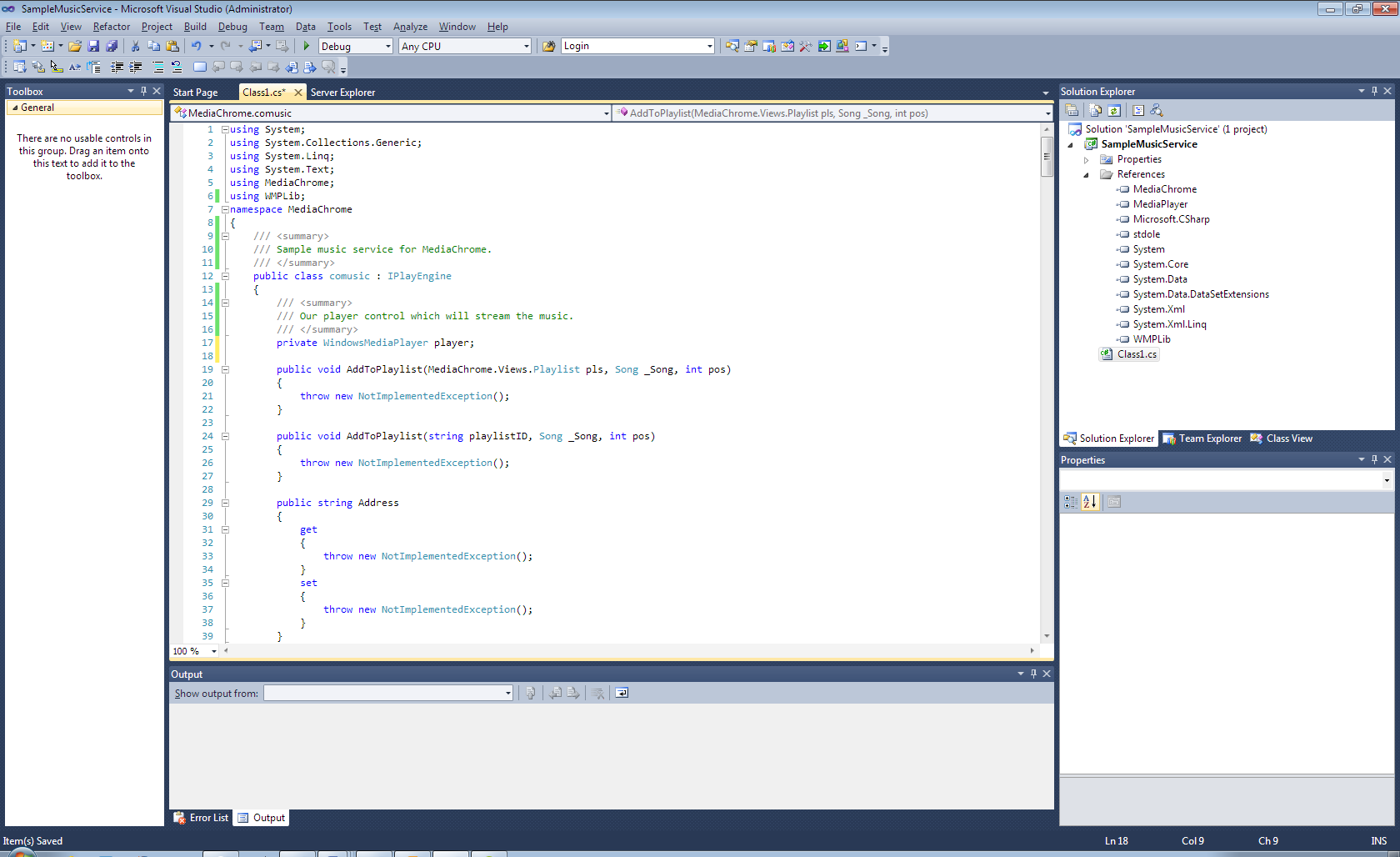
Find info about an special artist.

As we have no real service to deal with, we create an simple add-on which is capable of streaming mp3 on your own computer with an instance of WindowsMediaPlayer, but also lookup music from an database stored on the computer. We’ll also create an simple authentication model to show how handle user authentication.

## Creating the playback object

We must first create a **private** instance of the IWindowsMediaPlayer control, which handles our playback internally. This will not be accessible from MediaChrome’s shell unless the commands adapted by the routines in the IPlayEngine. Right-click on the **References** folder and open the add reference dialog as you did in chapter *3.1 Getting started,* select the ‘COM’ tab and find the WindowsMediaPlayer component Sort the component by the name header by ascend, and scroll until you find the component named ‘Windows Media Player’ versioned 1.0. If there are multiple choices, choose them all who apply. Click OK, and go back to the code editor.

Now, add an using named **WMPLib i**n the usings list in the beginning of the class code, then declare an private instance of the Windows Media Player Component:



This will be the **core** of the component. Now we’re going to handle the loading of track, but first we must have some theory lesion about track queries:

## Understanding core concepts of music queries

MediaChrome uses a distinct query model for consuming media and identify objects. The core element to know is the **URI subsystem.** Every track in MediaChrome is queried in two ways, **by metadata** or **by service pointer.** Here is an overview of the two concepts:

### Queries (URL) by metadata

The first one, is an general query, which ask the MediaMatch feature in MediaChrome to find the song with the matching mediaData to any available and installed service in MediaChrome. The query model is following:

Also written as

music://<artist>/<album>/<title>

#### Boxed HTTP substitute

As this query can be complex to handle in some operating system, applications and web browsers, we have created an **boxed** model of the query, which the **music://** statement is substituted to [**http://music/**](http://music/)**.** It’s an adaption for http only subsystems and is like the difference between [**http://open.spotify.com/**](http://open.spotify.com/)and **spotify:** query starting conversations.

Also written as

http://music/<artist>/<album>/<title>

### Query by service

The service specific query model is designated to this URL system:

The address part is an specific query string used by the service iself and does not need to follow any specific guideline by MediaChrome, but must be URL-friendly.

In our example **comusic,** we load the tracks from files on our computer. This means that decide the address part should point to an MP3 file. Consider following query:

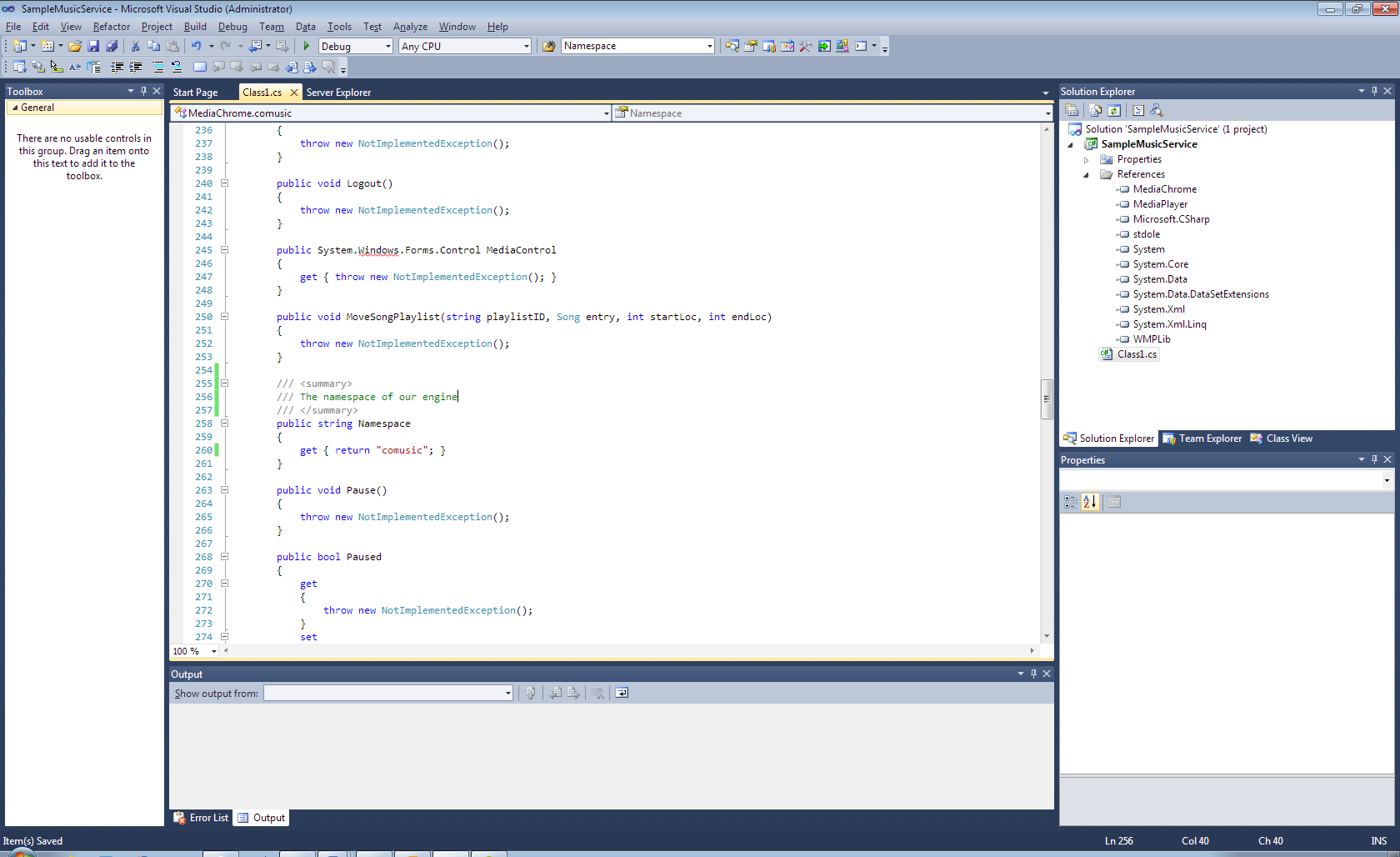
Comuic:C:\mymp3s\sample.mp3

That was all. Note the address is an common URI in the Windows file system.

## Implementing music queries for comusic model and basic playback workarounds

### Implementing the namespace definition

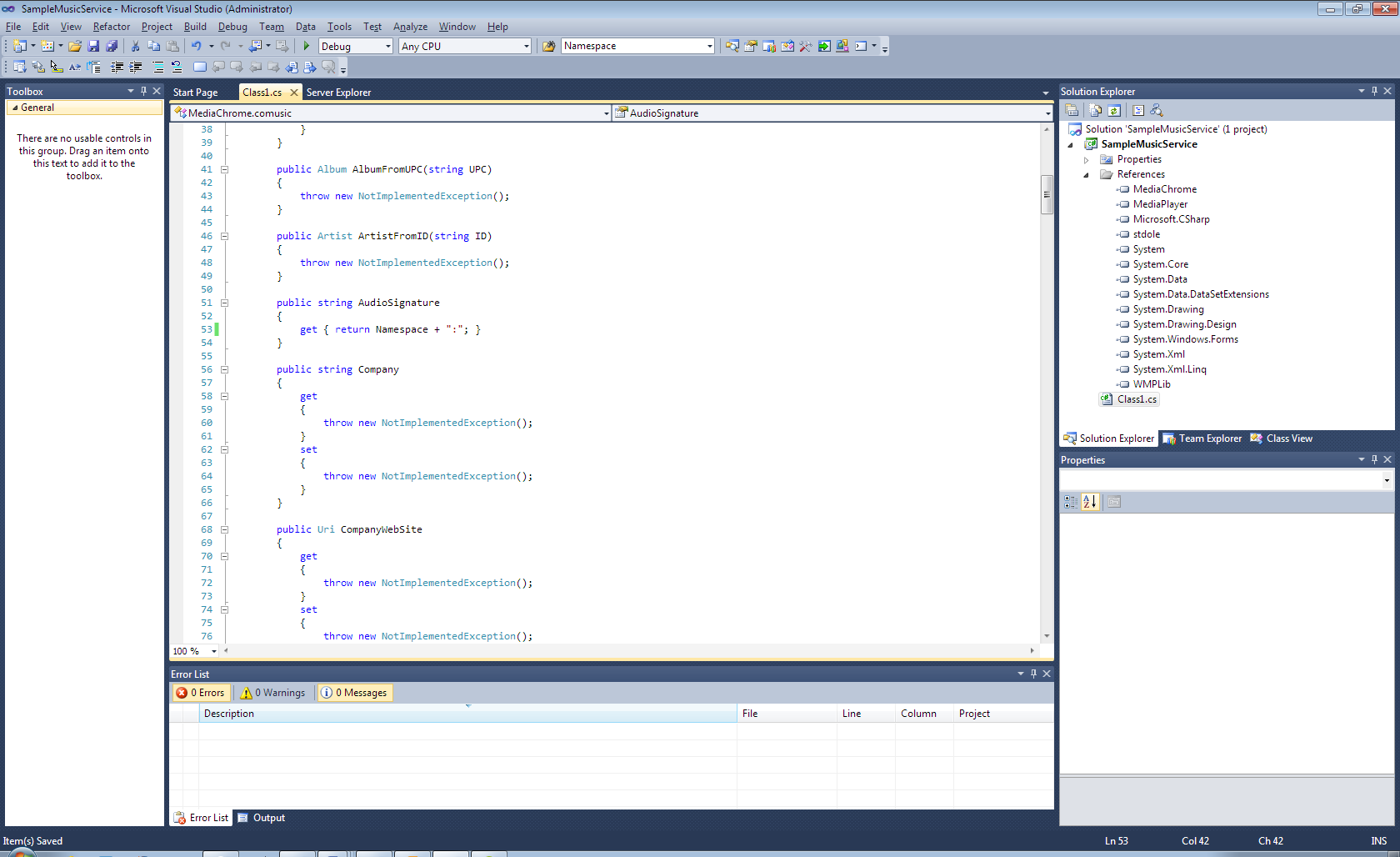
We will now implement comusic. First we should decide the namespace of the engine by implementing an read only property called “Namespace”. It should not be modified anywhere in the program flow. It must return an static value.



It returns an constant. Worth to say it again, it should **never** be modified through the run, otherwise weird things would happen with MediaChrome.

### Implementing the query namespace

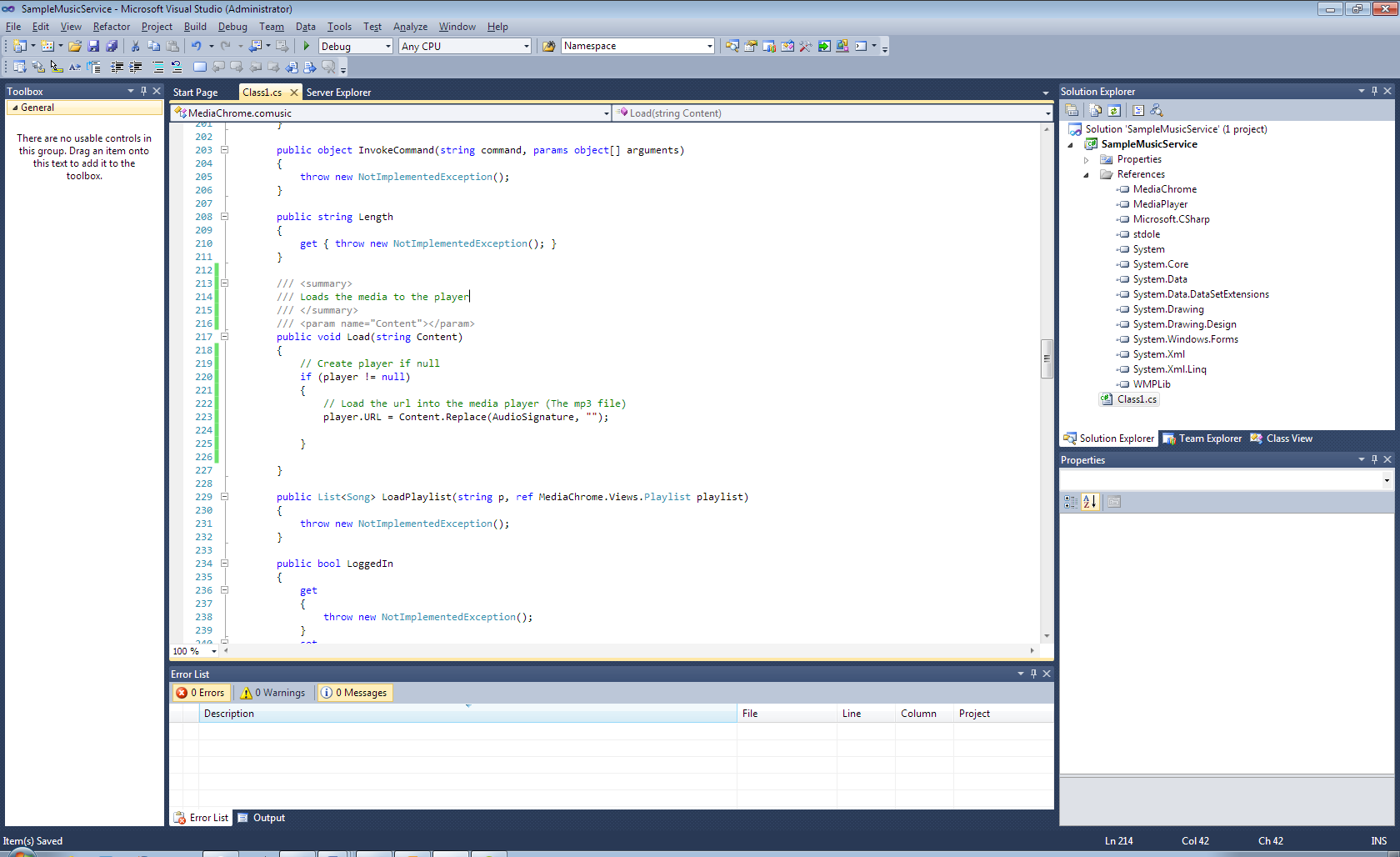
We must declare the query namespace too, and this is the same as the namespace but successes with a ‘:’ colon. This is done by implementing the return signature of the read-only property called **AudioSignature**.



Please notice that we have referenced the **Namespace** property instead of hard-coding the signature. The reason for this is to **preserve consistency,** if we for some reason need to change the namespace, we would spare the amount of work.

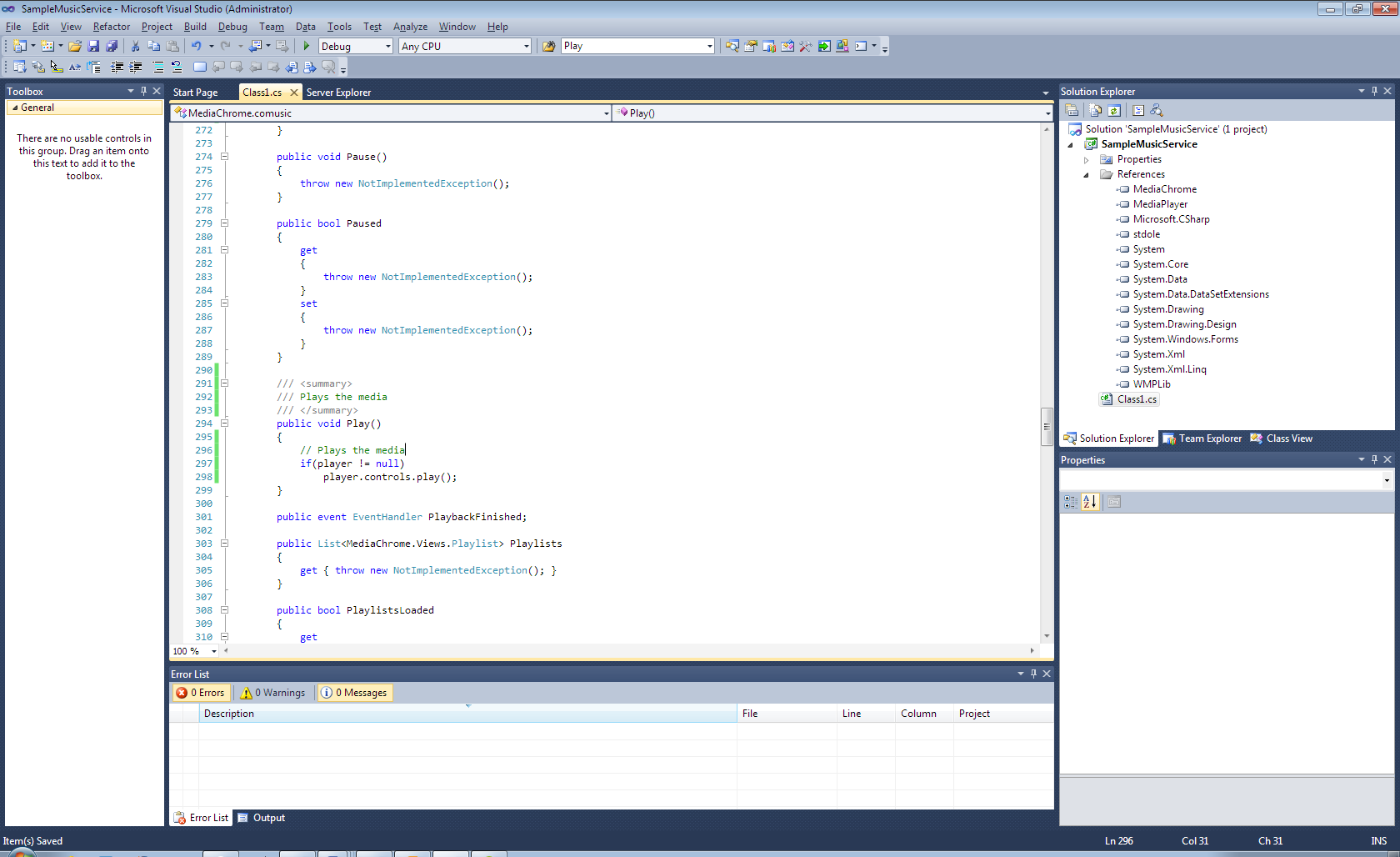
### Handling load query

The **load** method is responsible for **prepare the playback** of the the media resource pointed by the **load** string. We format the address and forward it to the internal WindowsMediaPlayer component.



You should notice that the **Load** method is not called as an initialize method of IPlayEngine, all initialization should be handled in the default constructor, i.e. public <className>() and that the void Load is called every time new media is requested for playback.

The initialization of the media is commonly success with the play method, which is the next one we implement. This is very simple:

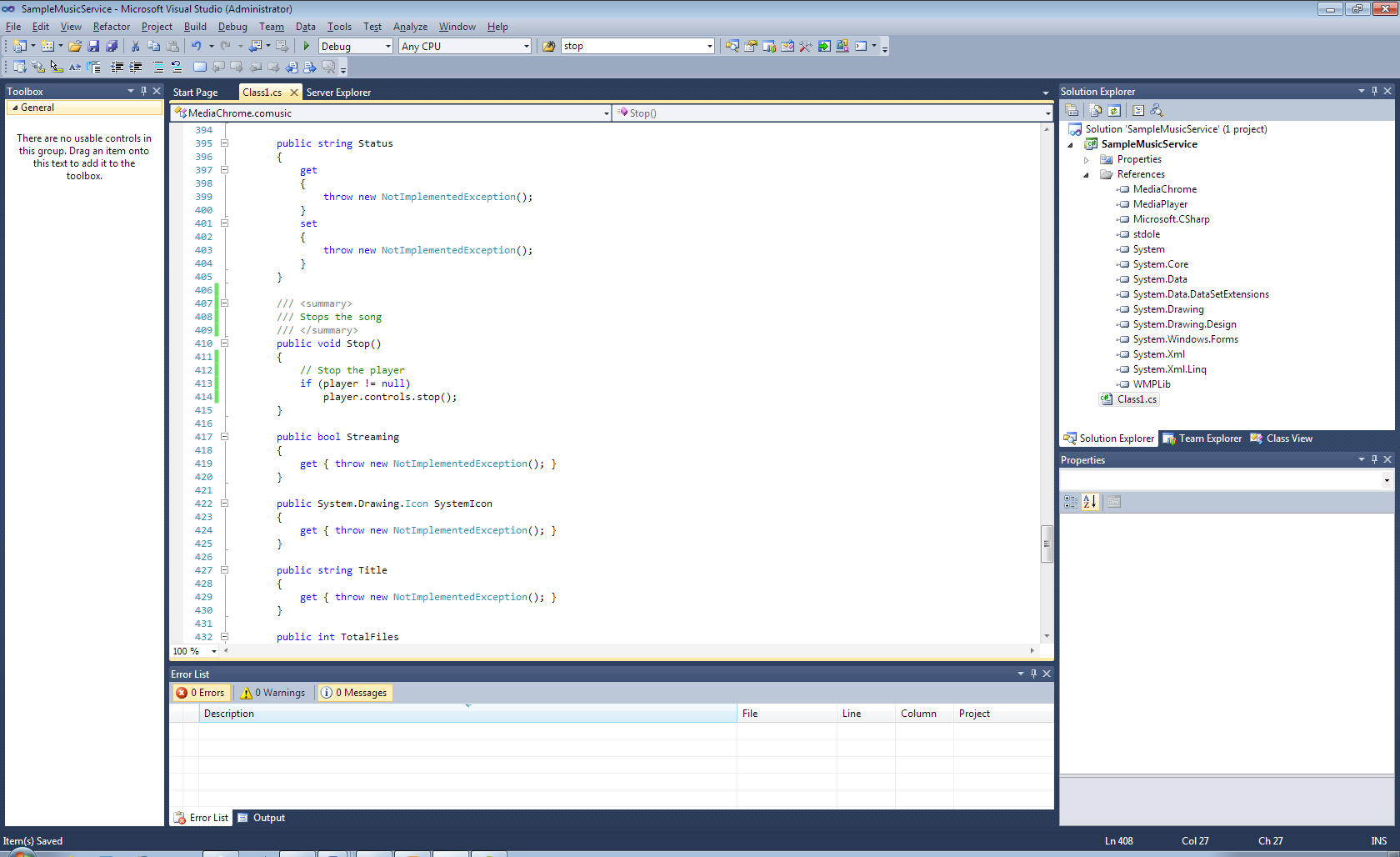


You should also make an distinct call to the “PlaybackStarted” event to ensure proper functionality. More on this later.

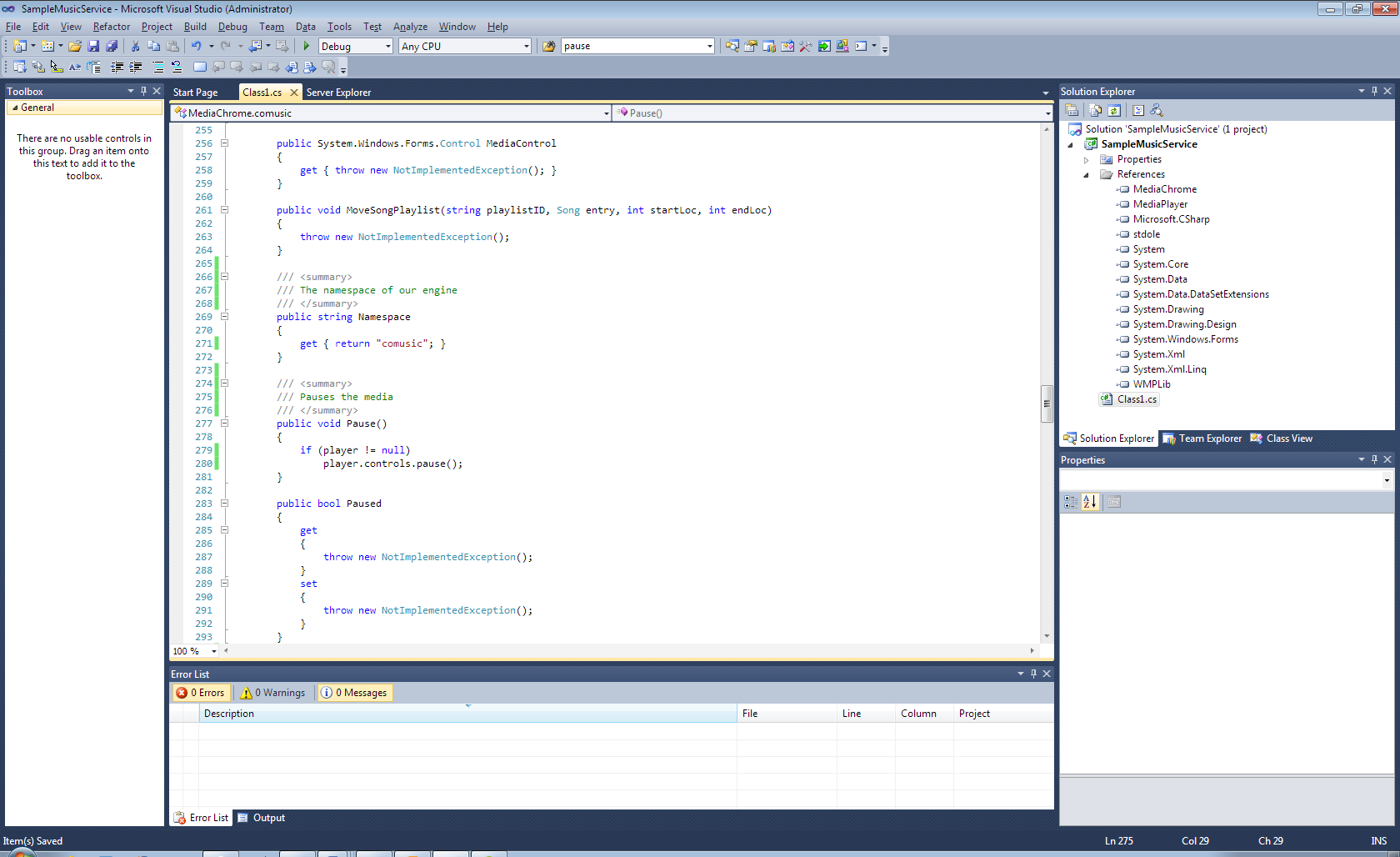
### Handling stop, pause and position of the song

Like all media player, you will need to handle common operations when playing media files such **stop, pause, seek** and retrieve the current position. This is trivial to do in this case:

For the stop operation, you implement the **stop** method:

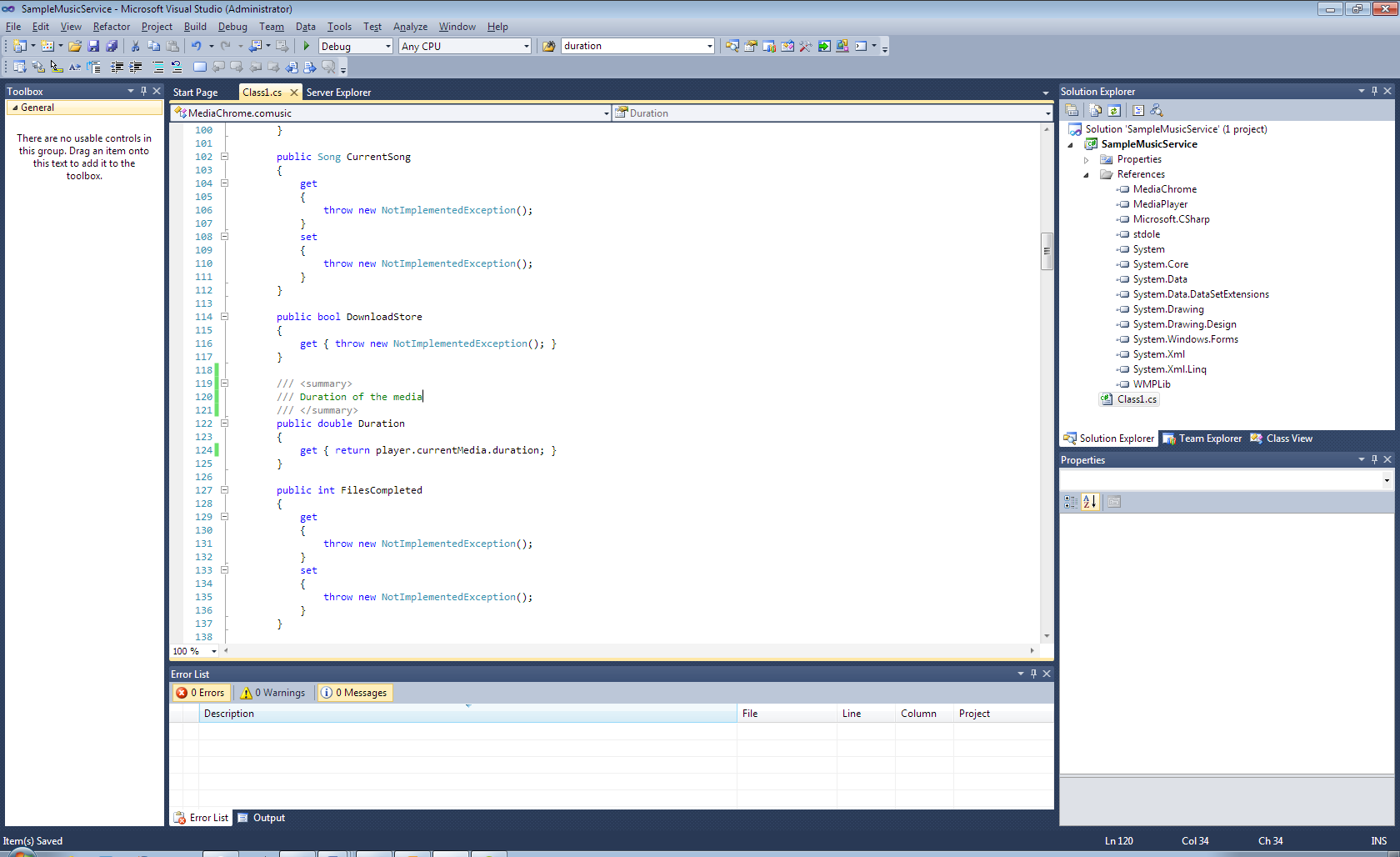


**Pause**



#### Position

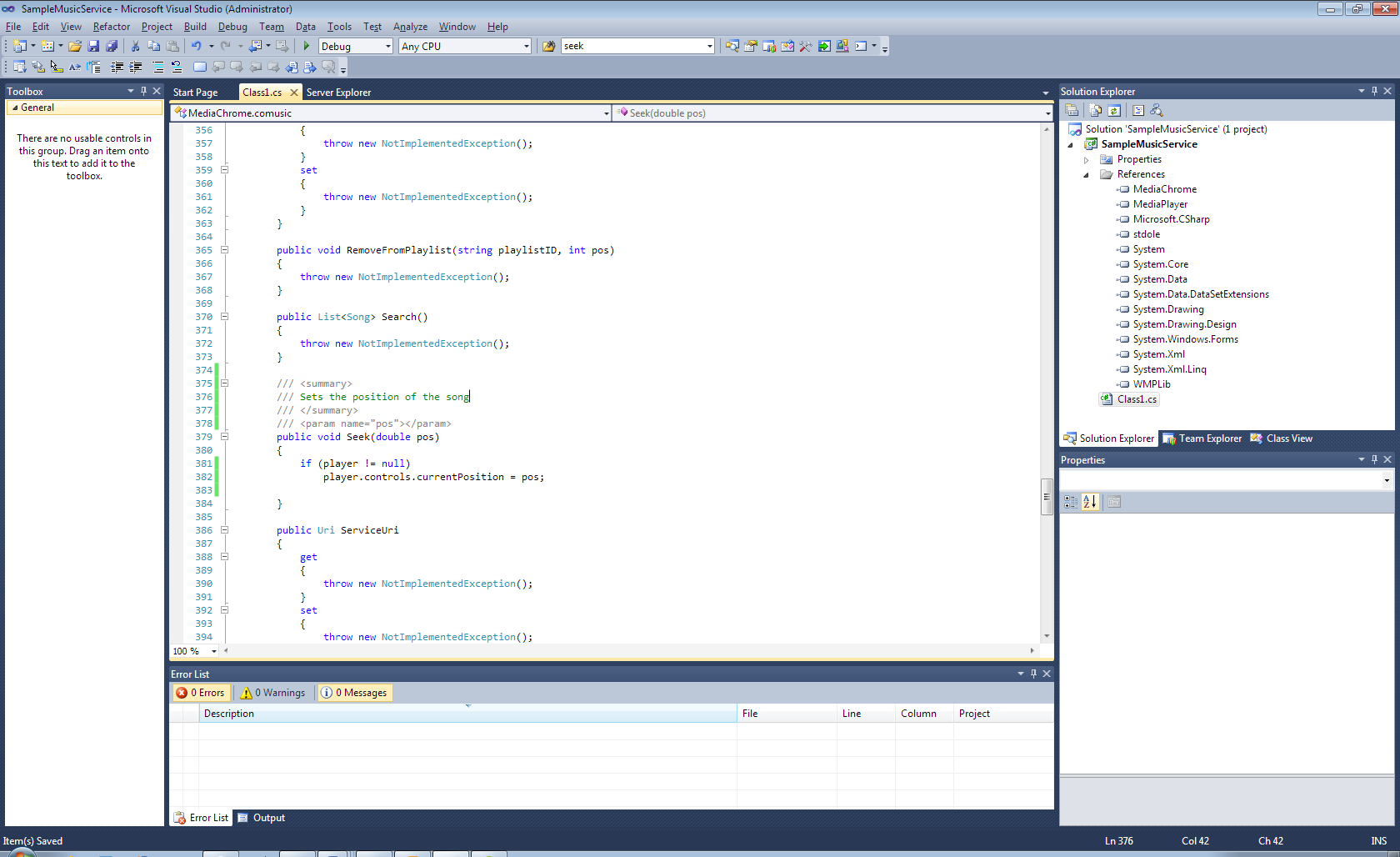
The position requires some words. The **duration** property of the engine, should define the length of the current song in **seconds**, not time span. The data type of the property is **double** and should return the position from the internal player, converted to a double integer.



When working out the design, we used WindowsMediaPlayer component, so this is one of the reason the data type were double. Additionally, we think this is a suiting model, since the double fits a larger time span.

**Seeking**

Seeking in media is achieved by a method called ‘Seek’. It is worked out in this way:



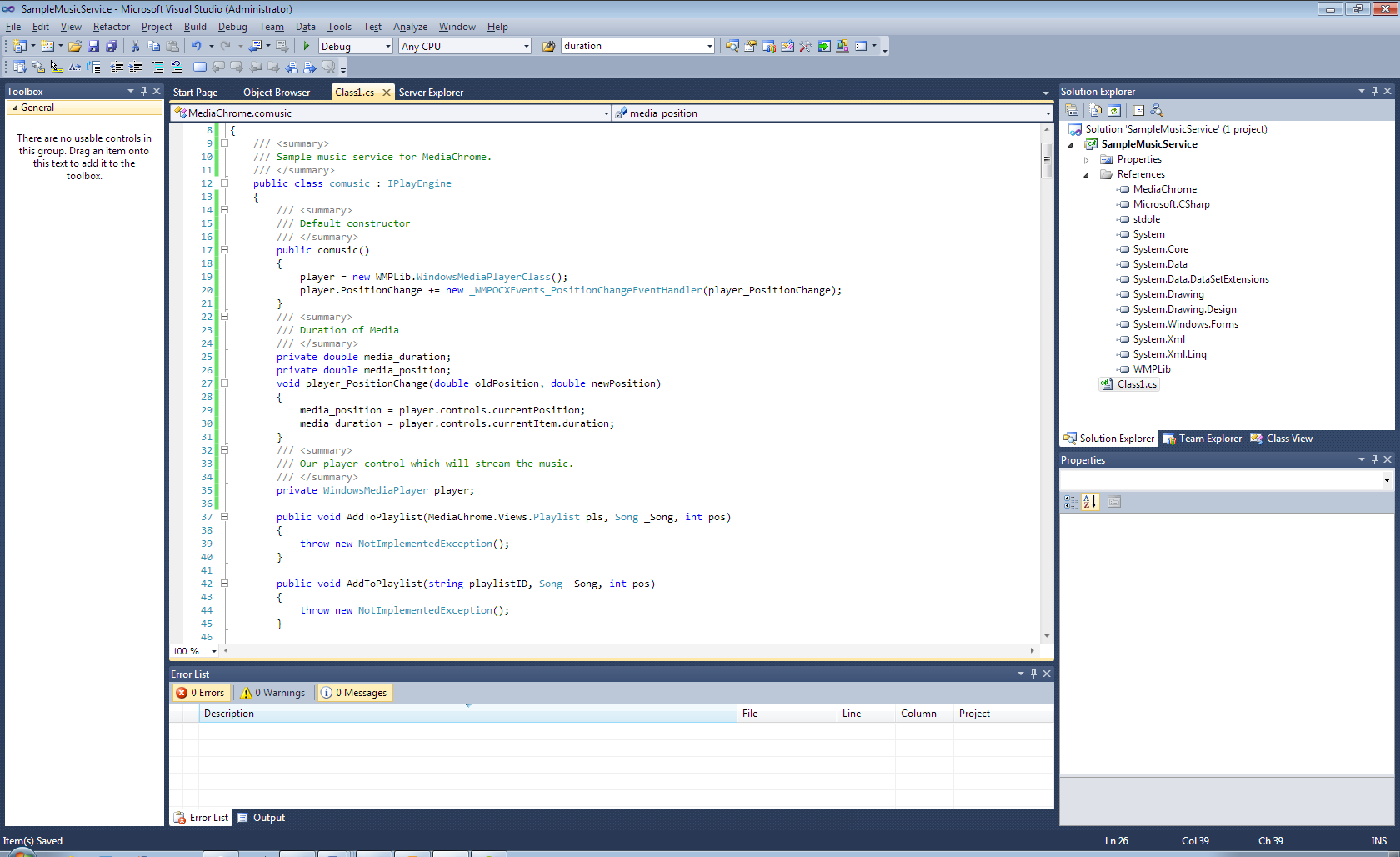
### Handling position changes

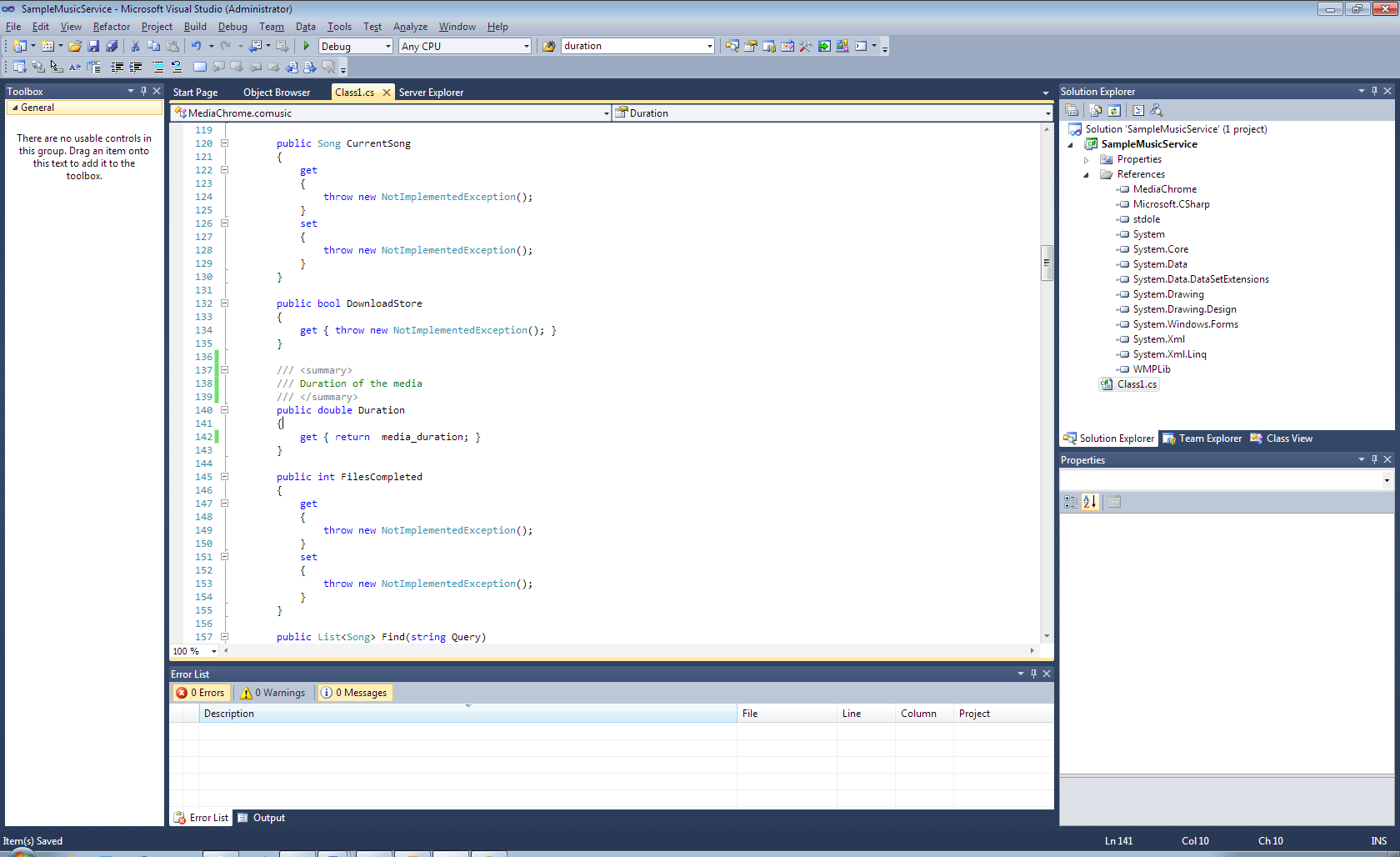
We’ll also now forced to add the initialized code as we add the first broadcast (event) handler.

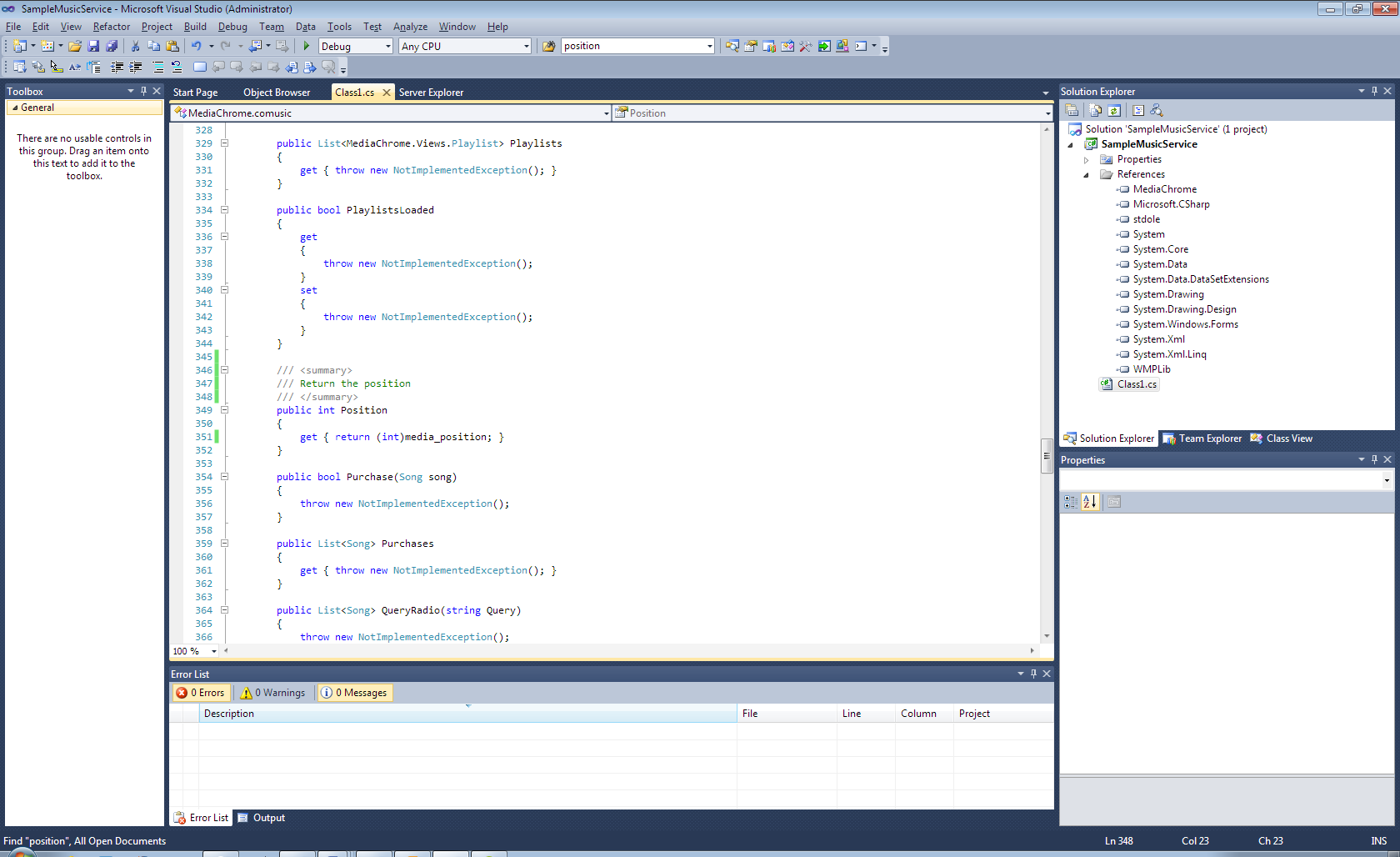
Please note that we should not return the duration and position directly from the internal player, since we aren’t sure if the object are instanced properly all times. In this objective, we create two private doubles media\_position, media\_duration which will proxy the position to the getters Position and Duration, and let the native events in the mediaPlayer class take responsible for updating the position.

You must be sure notifications from the internal media controller notifications about changes in the media operations, such position, starting, stopping is handled by the engine this case, we must implement the player objects

The initiator of the object







## Handling media lookup

One of the most important features in MediaChrome is the ability to lookup music across multiple services installed in the MediaChrome shell, which all boxed in IPlayEngine derived class. Now we’re going to tutor you about the various methods which will be responsible for this job:

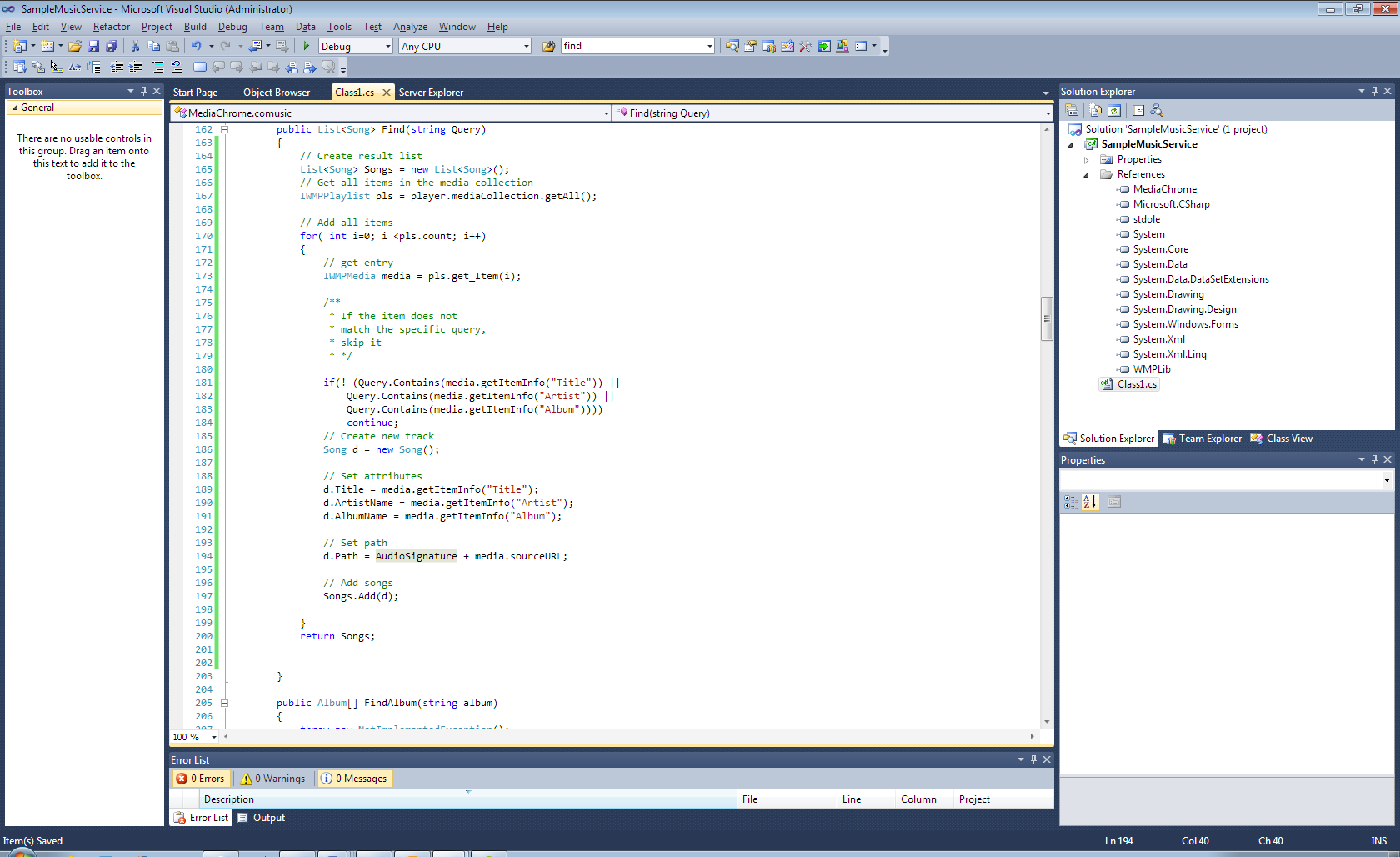
Please note that many functions here are only unimplemented stubs, as this is an merely demonstration of how the search routines are working. For real examples, you can study the source code to some open source plugins or the built-in MP3 engine. The objective examinates an situation where the service database consists of an local SQLite database, managing mp3 on the local harddrive.

### Handling the find method

**IMPORTANT!** This method is executing on an separate thread apart from the windows STATThread, which means that you **cannot do operations that implies on objects protected by cross thread behavior.** This means that you can not do any control modification or call methods on Control components.

When handling the find method, you will be provided with an search query. The intended result of the search function is an set of **Song** instances, which should have correct properties attached. To minimic a music service, we pretend matching the file from the windows media player library.

Consider this search outline:



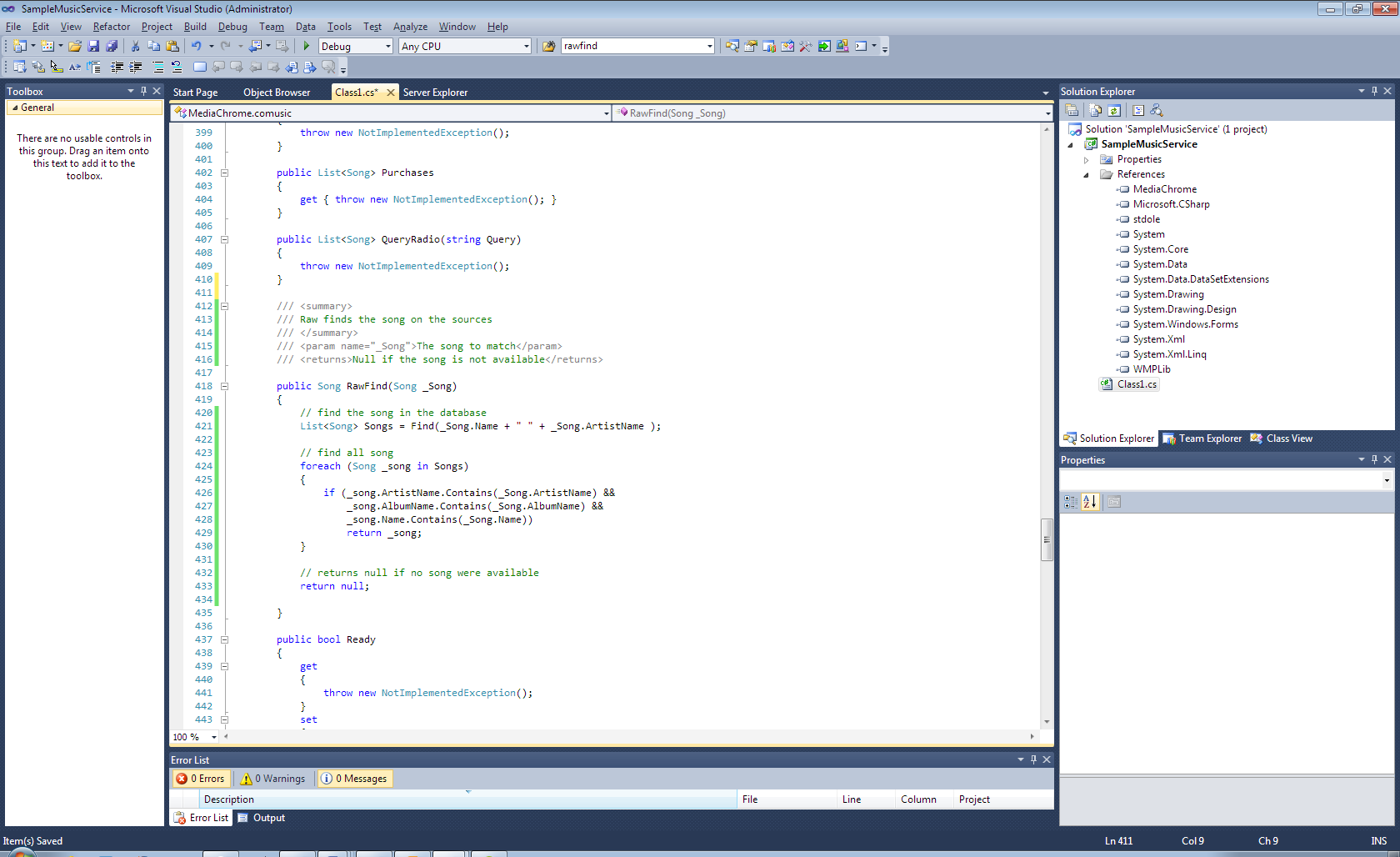
See the Song class reference in the reference table to see which properties you can apply to it.

### Handling RawFind() Method

And now, we’re going to handle the most important feature of the engine which is mandatory for making MediaMatch to work: RawFind().

This method will be called on each service when MediaChrome shell is seeking for available media.

Like the previous find method, RawFind() usually runs on an separate thread.



This is the core operation of the MusicMatch feature.

## Implementing authentication routines for the service control

In order to provide the best experience of MediaChrome and it’s plugins, you should be very polished when coming to implement the state properties and methods, like **LogIn()**, **Logout(), LoggedIn** and some other. These will help MediaChrome to handle the state of the services correctly. Although you can implement a good error handling in the engine, implementing all these routines will help MediaChrome to understand the states of the media sources and prevent weird problems. For example, it will spare resources if MediaMatch skip matching an track against an service, whose requires an account you haven’t logged in to, to enhance the speed.

### Handling user authentication

User authentication is important in every shared application aspect. In every ASP.NET book, the author says some wise words about login. This is something we’re going to do here. User authentication is fundamental here too.

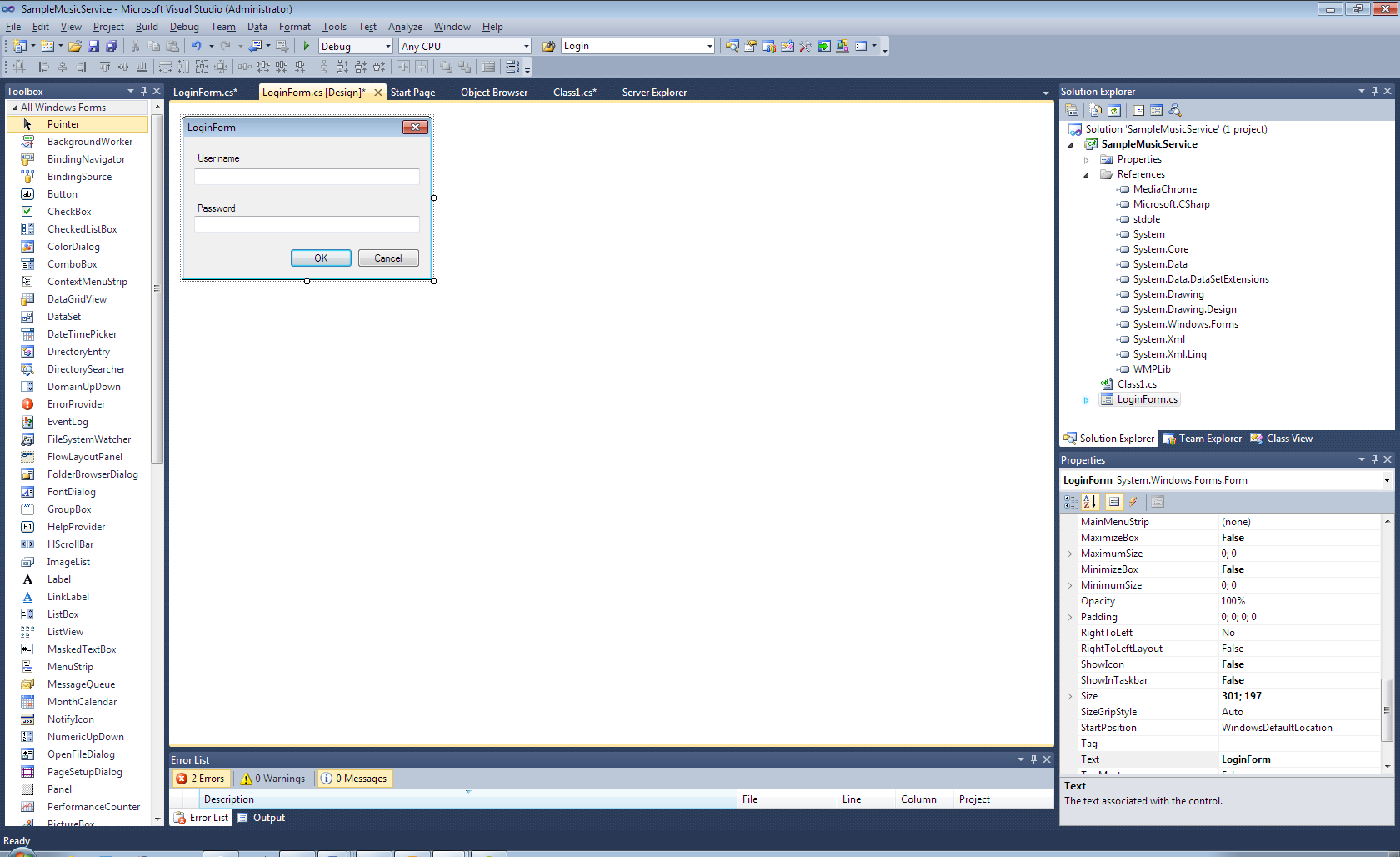
As media chrome plugins are adapters for various kind of music streaming services, it are possible many of these has routines for user authentication. For example, paid services like Spotify or WiMP requires an user to be authenticated before delivering playback to users. The tech routine of the login process is worked around for each service, and you should understand them well before going on. The intended implementation of the methods described in this section is to adapter media chrome media engine to the routines defined in the service object. Since we are working with an local playing only, we can’t unfortunally demonstrate how an such apporoach would be worked around in the reality. But we can off course show the logic about how the authentication works.

#### Creating an simple login

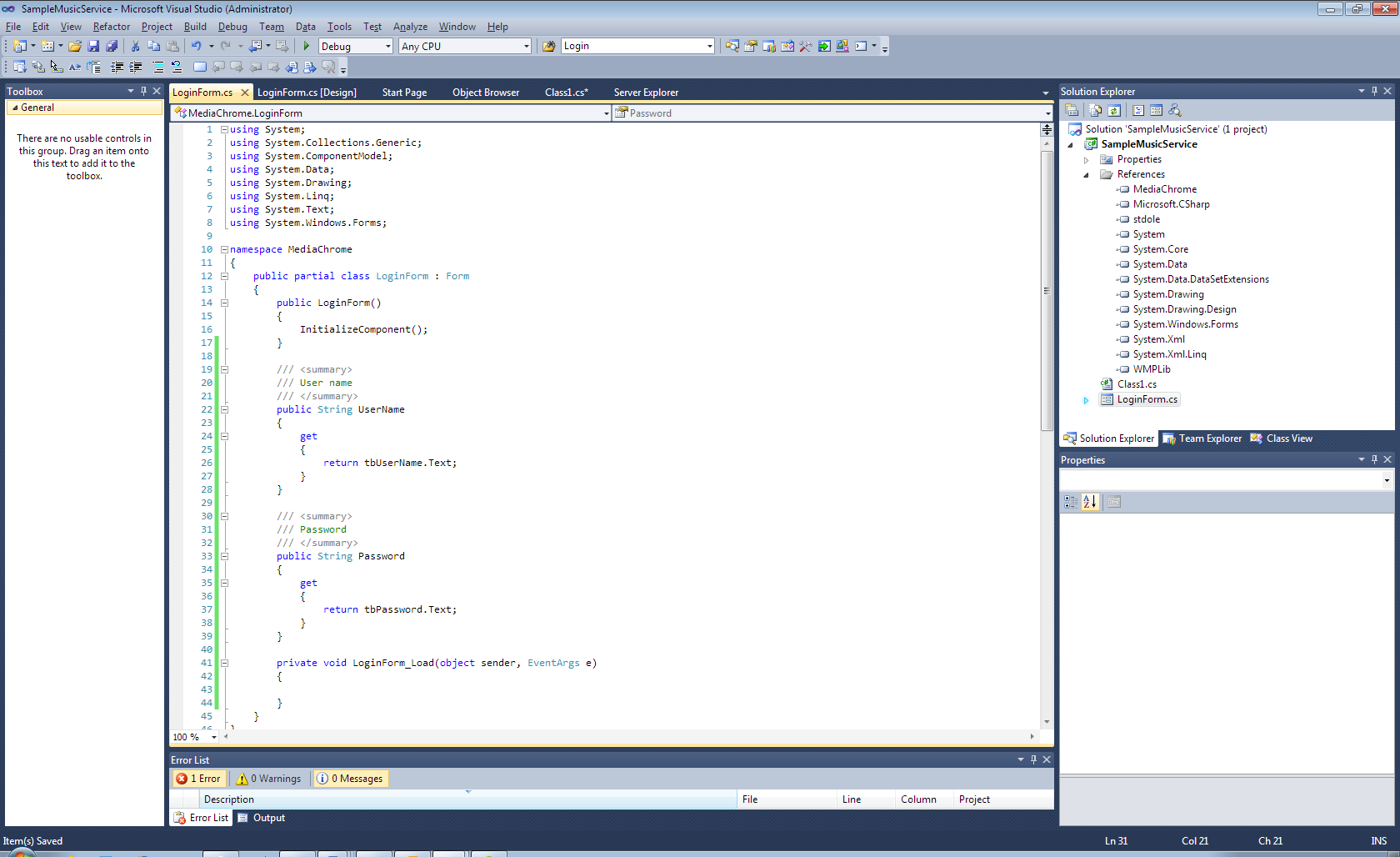
This objective shows how to create an simple login form. We do assume you’re already familiar with WindowsForms programming, as it are outside this document coverage.

But we’re going to act like if this was an online service to demonstrate. Let us create an login window. All plugins has their own login routines. Right-click on project **SampleMusicService**, select add > Windows Form… Name the form “LoginForm.cs”. Then create the form so it has two **TextBoxes,** tbUser and tbPassword, two buttons, tbOK and tbCancel. Clicking on tbOK should exit the form with the DialogResult OK and cancel with Cancel.

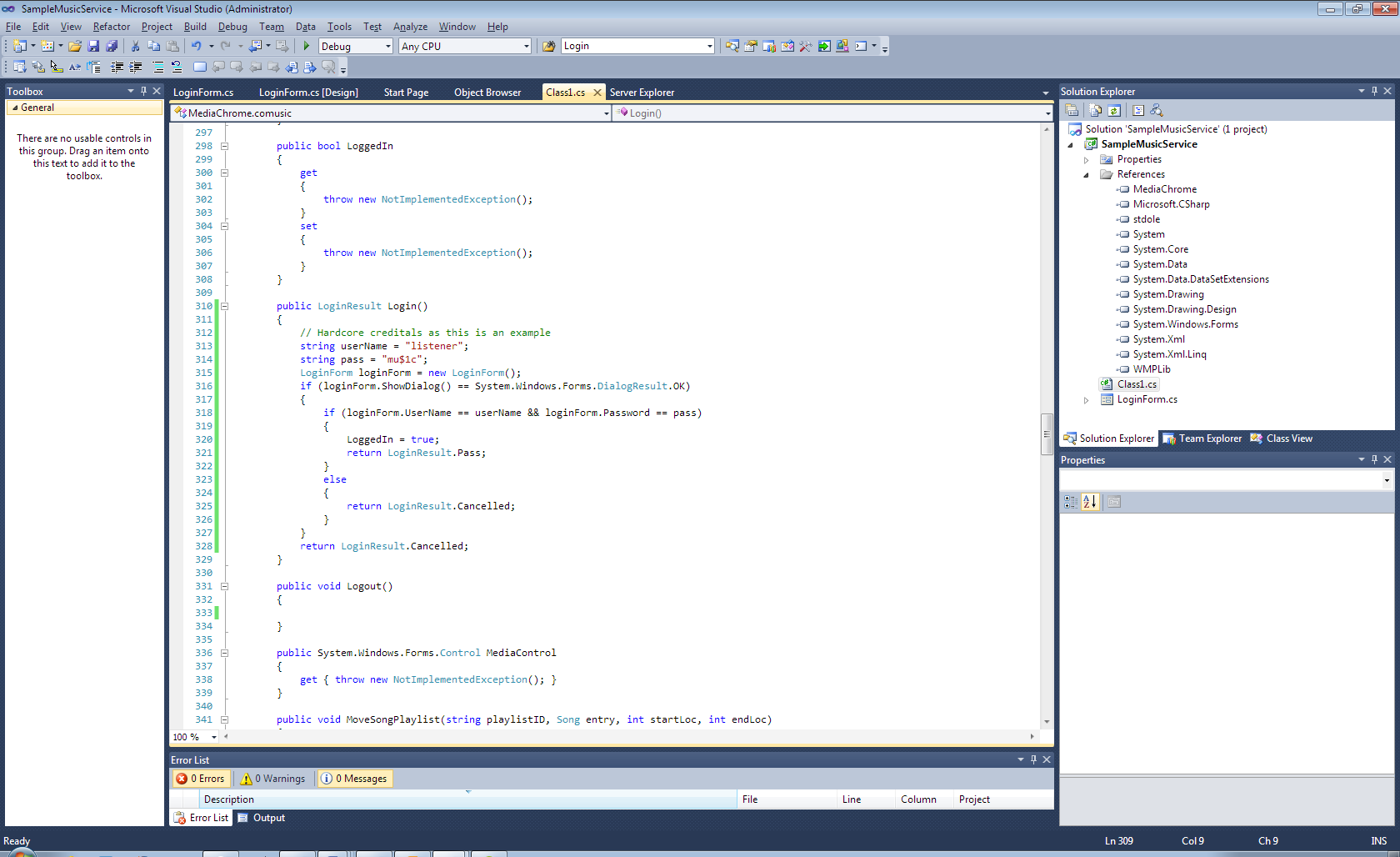
This is an example of how it could be:



Double click on the form and add this code after the initialize:



Now, we should implement the login routine. The login returns an value of the **LoginResult** enum. As this is an demonstration, we create an hardcore requirement for an successful login, the user name is “listener” and password “mu$1c”. If the incorrect credentials aren’t provided, the method returns **LoginResult.Fail** which tells MediaChrome that the login were incorrect. **LoginResult.Cancelled** should be returned if the login process were cancelled by the user, or **LoginResult.Pass** if the login sucessed. Before exiting the scope, you must set the property **LoggedIn** to **true** so MediaChrome knows that the engine has an authenticated session.



Now we must implement the logout routine as well:



# Deploying media services

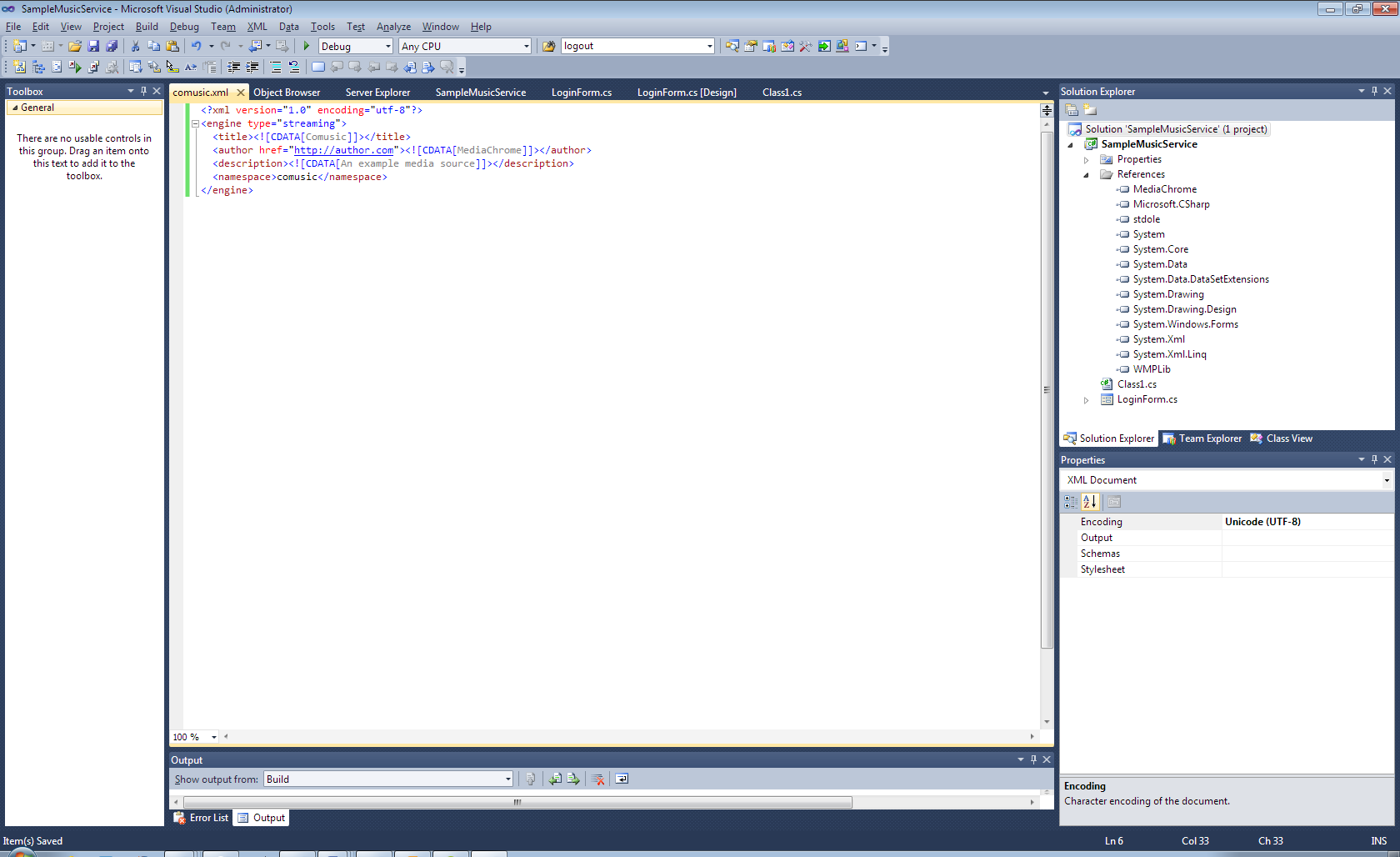
To deploy an media service, zip all the contents including sub-folders in the bin directory. First, we must make sure the assembly name is the same as the namespace, so MediaChrome can locate the dll.



Then click Build > Build Solution. When the build is ready, right click on the solution and select ‘Open in explorer’, and browse into the bin directory.

Inside the bin directory, right click and create an new text file. Then open it in your text editor, and save it under an new name ‘comusic.xml’ (double-quotes **should be** included) in the file menu. When the document is saved under an new name, we should create the description XML file.

We can also drag the file into the visual studio window, which we’ll do here.



For more detailed reference of this xml manifest, refer to the reference documentation in the same folder.

The folder should now have this content.



You’ll also need a png image, comusic.png (<namespace>.png) in the same directory for the little tiny icon to be shown.

Then drag the bin folder into the corresponding FTP location, the root of the repository of the engines. Rename the bin folder to the namespace.

# References