[3.1 Exploring the .NET Framework 2](#_Toc242074308)

[3.1.x Key features of .NET 2](#_Toc242074309)

[1.x.x .NET and Windows integration 3](#_Toc242074310)

[3.1.1 Providing a common runtime environment 3](#_Toc242074311)

[3.1.2 A wealth of libraries 4](#_Toc242074312)

[3.2 A runtime for dynamic languages 5](#_Toc242074313)

[3.2.1 It’s all about shared types and messages 5](#_Toc242074314)

[3.2.2 Walking through an execution of an expression 6](#_Toc242074315)

[3.3 Hosting the DLR in your applications 7](#_Toc242074316)

[3.3.1 Bird’s-eye view of hosting the DLR 7](#_Toc242074317)

[3.3.2 Hosting the DLR in a .NET application 8](#_Toc242074318)

[3.4 WPF and IronRuby make a nice pair 11](#_Toc242074319)

[3.4.1 XAML: Development: now assimilating designers 11](#_Toc242074320)

[3.4.2 A simple browser example 11](#_Toc242074321)

[Heading here 12](#_Toc242074322)

[Heading here 13](#_Toc242074323)

[Heading here 14](#_Toc242074324)

[Heading here 15](#_Toc242074325)

[3.5 Join me on the Silverlight bandwagon 16](#_Toc242074326)

[3.5.1 What’s Silverlight made of? 16](#_Toc242074327)

[3.5.2 Is Silverlight for me? 17](#_Toc242074328)

[3.5.3 How do I use it? 18](#_Toc242074329)

[Heading here - Write the XAML 18](#_Toc242074330)

[Heading here - Get the XAML into IronRuby 18](#_Toc242074331)

[3.6 Developing for the web with ASP.NET 18](#_Toc242074332)

[3.6.1 Webforms: using ASP.NET the classic way 19](#_Toc242074333)

[3.6.2 MVC: using ASP.NET the new and improved way 19](#_Toc242074334)

[3.7 Summary 19](#_Toc242074335)

3

.NET puts the Iron in IronRuby

This chapter covers:

* Components of the .NET Framework
* WPF for building desktop applications
* Silverlight for building rich internet applications
* ASP.NET family for creating internet applications

When Microsoft announced the .NET framework, my colleagues were absolutely stoked because it would solve a lot of versioning issues with dlls. It was going to be something that would allow people to write libraries in one language and use them in another. Those things sounded like pretty cool features to me. I started learning VB.NET, but quickly switched to C# because it felt more natural to me. The beauty of .NET was that the libraries I had previously learned when using VB.NET wasn’t in vain. I could just use that knowledge in C# without having different interfaces to classes etc.

In chapter 1, we covered the differences in typing systems between a classical .NET language, for which we used C# as an example of a canonical static language, and the ruby language. In chapter 2, we talked about the ruby language and some of the cool things you can do in Ruby. This chapter is about the .NET Framework, how it ties in to IronRuby, and why that’s a good thing. Before we get to what the DLR does for you, we’ll have to go through what the .Net Framework actually is. We’ll also look at what WPF is and does for you. After which we’ll explore the technologies .NET offers you to develop applications for the Internet. One of these technologies, the most innovative, is Silverlight. We think Silverlight will significantly change how rich Internet applications will be developed in the future. The other two topics of discussion regarding Internet technologies are ASP.NET Webforms and ASP.NET MVC, we’ll check out how to make use of those with IronRuby as well.

Note

Most of the samples from this chapter should run both on mono and on the Microsoft .NET framework. The samples that utilize WPF probably won’t run under mono because at the time of this writing mono doesn’t have an implementation of the presentation framework in the olive project. If you don’t know what Mono is don’t worry we’ll explain it later in this chapter.

Nonetheless a short history lesson is in order to understand where we come from and where we’re going. At the beginning of this book we said that we assumed you knew what the .NET framework is but we didn’t expect you to know how your code executes. So to avoid leaving you baffled because the first part of the story is missing we’re briefly going to look at what the .NET framework does and is.

3.1 Exploring the .NET Framework

In my opinion, the .NET framework is the best invention Microsoft has made for developers in the last decade. In this section, allow me to elaborate a little on why I made that statement. We’re not going to take you back in time to explain all the problems that existed before, but rather explain what the .NET Framework can do for you.

3.1.1 Key features of .NET

The .NET Framework consists of two major parts:

* a common language runtime (CLR) engine
* the Framework Class Library

Using the .NET framework for your application development gives you the ability to deploy applications by just copying them from one location to another without the need for an installer. The .NET framework also tries solving the issues around library versioning of the dependencies for your application because they can just be copied next to the executable.

The CLR gives you a consistent programming model for all types of applications whether it’s a windows service, a command-line interface (CLI), a windows forms app or a web application. The way you program and configure the application is always consistent. This makes it a lot easier to develop a wide diversity of application types without having a steep learning curve to get into a new type of application.

The CLR greatly simplifies the windows programming model. It doesn’t have the same plumbing requirements typical win32 applications used to have like Globally Unique IDentifiers (GUIDs), the registry and so on. That means it frees your time to focus more on the task at hand instead of doing boring and often error-prone boilerplate code. Once an application has been installed and run on a pc it should always run, partly because of the reasons mentioned above, like the fact that it doesn’t suffer from versioning problems etc.

The CLR allows developers to write libraries in one language and use these libraries in another language. It also saves you from having to compile and test your application on every different processor architecture that may be used (x86, x64, IA64,…). As long as there is a .NET framework for that architecture you get the support for free.

In chapter 1 on typing we discussed that type-safety is very much what strong typing is about. The CLR ensures type-safety so that objects will always be accessed in the same way. It gives you a consistent model for error handling. If there is an error it always will throw an exception that you can handle in your code. The exception handling includes a stack-trace, making it a lot easier to look for the cause of bugs. On top of all the goodness we described above, the CLR provides excellent tools for securing your application and your code.

Microsoft wrote the .NET Framework to run on Windows. They submitted the specifications to the ECMA commission, and ECMA accepted it as a standard. This enables other companies to build their own implementation of those standards. Novell has implemented those standards in the open-source project Mono. This extends the reach of the .NET Framework to a wide range of operating systems including Linux and OS X.

There are two other features we would like to mention because they are quite superb; the first is automatic memory management. The CLR provides a garbage collector to manage the memory in your applications and that makes developing an application considerably easier. And, lastly, there is a Common Type System that all languages on the CLR use, more about that later in this chapter.

Figure 3.2 illustrates how all of the features debated above are enabled for every language that runs on top of the .NET Framework. Each of these languages compiles into a Common Intermediate Language (CIL) that gets translated by the CLR for the specific platform it’s running on..

3.1.2 .NET and Windows integration

Figure 3.1 shows a high-level overview of how the .NET framework ties into the Windows operating system.

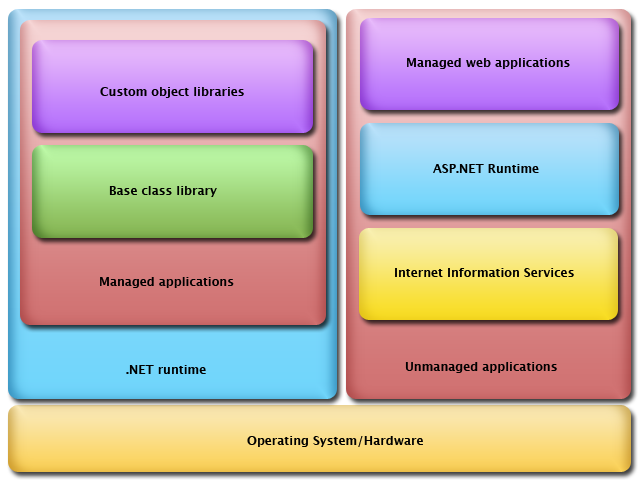


Figure 3.1: This is how the Microsoft .NET Framework integrates into the Windows operating system.

This figure shows, on the left side, how *managed* applications get to the operating system. By managed application, we meanthat the application runs on top the .NET Framework; the application is then managed by the CLR. So we have the .NET runtime, which functions as a host for those applications. The Base Class Library (BCL) provides a broad and solid foundation for basing your custom libraries on.

On the right side, the figure shows a special case for hosting the CLR in Internet Information Services (IIS) to enable ASP.NET for running managed web applications.

Next, we’ll look a little bit closer at the Runtime and the Framework Class libraries. The runtime environment functions as a virtual machine that is able to optimize the managed code for the specific platform it’s running on. For the remainder of the book, we’ll be talking about the Microsoft .NET Framework, but most of the samples should work on Mono as well.

3.1.3 Providing a common runtime environment

The common runtime environment is mostly referred to as the Common Language Runtime (CLR). The CLR is responsible for compiling the generated CIL code into machine-readable code (figure 3.2). Because the runtime is running on the actual machine it is executing code on and because the CLR does something called Just-In-Time (JIT) compilation the CIL compiler has a lot of information about the application and platform available at run-time and can thus provide highly optimized code for the current platform. In my opinion that is great, but there is a trade-off to make. It also means that before your application starts, the CLR has to be loaded for it to be able to execute your .NET code. This causes a slight delay in startup time. And the first time a method is executed there is an extra compilation step. This step also causes a very small delay.

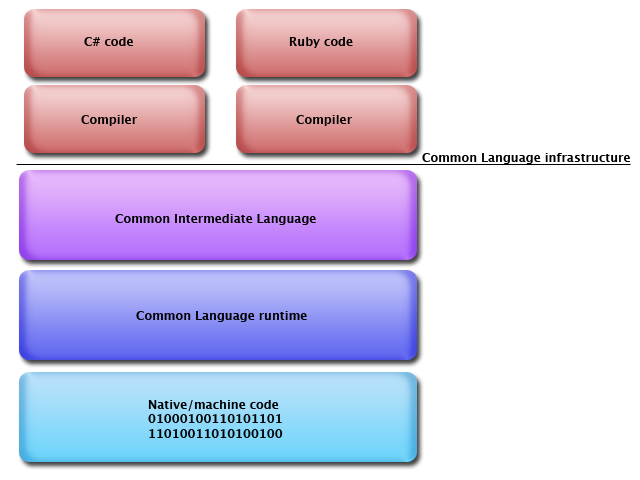


Figure 3.2: An overview of how the CLR translates your code into machine code

Figure 3.2 shows that code written in a language supported by the CLR is translated by its compiler into CIL. At that point the code is living in a language agnostic environment across the boundary of the common language infrastructure. That CIL is passed on to the CLR who will JIT compile it into native code to be executed by the system.

In addition to the features we talked about earlier the CLR also provides facilities for thread management. It hugely simplifies the multi-threading story. And Microsoft will be adding more parallel computing goodness in the version after the 3.5 version of the framework. In chapter one we discussed how type-safety (strong typing) helps in creating more robust code. We also discussed that the runtime enforces type-safety. It does that by having a Common Type System. Those types are somewhat like language agnostic types. The CTS allows for languages to be used interchangeably in the same application but in different assemblies, a type in one language is exactly the same in another one. The CTS accomplishes this by including metadata that describes the type. This allows you to interrogate objects about their type information at run-time like members, properties and so forth.

Now let’s deal with the base class libraries and how they can make your life easier in the IronRuby world. To send an email for example; you typically have to rely on a third-party library that encapsulates that process for you so you don’t have to deal with the SMTP protocol directly, this library is included in the .NET Framework. The .NET framework implements a ton of those libraries that you can use for free in IronRuby.

3.1.4 A wealth of libraries

The second part of the .NET Framework is the BCL and the Framework class libraries. These libraries make an otherwise difficult task drastically easier to implement. Some of these tasks include sending emails, enabling data access to a bunch of data sources, and building a web application or a web service. These libraries also provide facilities to build console applications, GUI applications and many others.

The scope of the base class library is much larger than the standard libraries for other languages. The base class libraries are simply brilliant because they allow you to do most common tasks on virtually any platform using a consistent API from different languages with one and the same library. In addition to being a very rich library the quality of the library is extremely high and rich in functionality. There are also a large number of 3rd party libraries available out there in the wild that will perform almost every task you need.

And because all of those libraries run on the CLR it doesn’t matter in which language the library is implemented; you can just use it all the same. To me this is the one of the best strengths of the whole .NET experience: you can program in any language you want, as long as it’s implemented on .NET. You can then ship your compiled binaries to clients and other developers and they can use them in the language they want.

We will be using the BCL in various places, whether we’re creating a WPF, Silverlight or web application we will always rely on libraries that have been predefined in the BCL. When we’ll be requesting urls on the web from our Silverlight application we’ll use WebRequest and that class is defined in the BCL.

Now that we know how the .NET framework does its thing, let’s take a closer look at the DLR, which is a runtime that runs on top of the CLR and provides facilities for dynamic languages, like IronRuby, to run on the .NET framework.

3.2 A runtime for dynamic languages

When we’re talking about essential building blocks of the .NET Framework, in the context of dynamic languages there is a third building block. In addition to the CLR and BCL there is also a Dynamic Language Runtime (DLR). The DLR is a set of services that run on top of the CLR and provide facilities like a hosting API, a shared dynamic type system, dynamic method dispatch and dynamic code generation. These services are language agnostic and are meant to make it easier for language developers to implement their language on top of the .NET framework.

The shared dynamic type system allows you to freely share code with other dynamic languages as well as with the static languages implemented on the .NET Framework. The most important bit of the previous sentence was that you can share code with any language that is running on the CLR or DLR.

Another great benefit of the DLR is that the language implementer gets a whole bunch of stuff for free like a Garbage Collector and debugging support for example. Also any optimizations that Microsoft performs in the DLR are automatically incorporated in the languages implemented on top of it. Previously we mentioned the wealth of libraries you have at your disposal, these are still available; and can also make a lot of difficult tasks really easy to perform in a dynamic language. Obviously you would have to write less code to get there but that’s a nice side effect from dynamic languages.

There are a couple of things worth knowing about the DLR, which we’ll discuss in this section. First, we’ll look at how the DLR thinks about its types and how it invokes methods on the types. Second, we’ll walk you through what happens when you create a method and invoke that method in the context of IronRuby and the DLR.

3.2.1 It’s all about shared types and messages

We’ve seen before how the CLR has a common type system. We’ve also seen that the DLR is an extension from the CLR that provides facilities for dynamic languages. By now we also know that the CTS provides metadata for its types, this makes it possible for the CLR to share its types with the DLR. This is an important fact because it will allow for very efficient interoperability for the different languages. Concrete this means that an object from the CLR is the same as object in the DLR and is completely available in the DLR.

We’ll look the interoperability story a little bit later, for now we still need to find out how methods and properties are being invoked on objects in the DLR. The DLR is built around passing messages to objects. When the IronRuby interpreter starts, it builds up a Ruby language rule set that defines how the expressions need to be executed. When you want to invoke a method you send a message to an object, the DLR will look up the rule whose name is included in the message. Once the rule is found it will be executed. Figure 3.3 shows a high-level diagram of the steps involved.

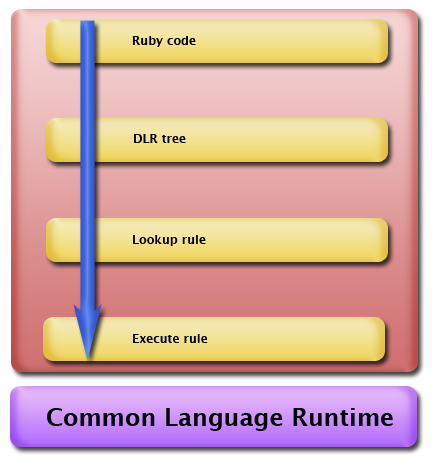


Figure 3.3: A simplified overview of the DLR’s execution model

Let’s first walk through a simplified example of what’s going on, we’ll look at how it really works in the next section. In the first chapter I said that the DLR generates IL that isn’t completely true. The DLR is actually a little smarter about it than that, because generating IL wouldn’t be as efficient as the way it is done now.

When you pass a message like puts “hello world” to the interpreter, the interpreter looks at the statement and builds an Abstract Syntax Tree (AST) for it. An AST is a simplified tree representation of a unit of source code; in the case of the DLR we call it a DLR tree. You can think of it as an internal representation of a program with all the language specific fluff removed. That DLR tree gets passed on to the DLR and the DLR looks in its rule set for a rule with name puts, when it can find the rule for puts it will tell it to execute with the parameter “hello world”. In the next part we will walk you through the execution and definition of a method in IronRuby.

3.2.2 Walking through an execution of an expression

We thought it might be interesting to know what happens behind the scenes. Having an understanding of what causes for the DLR and IronRuby to work in the way they do should help with demystifying the apparent magic that seems to be going on. We will go into a little bit more detail than the previous section, but this book isn’t about the DLR but about IronRuby. We’re only highlighting this because it will help you understand what’s involved in case you want to extend the IronRuby implementation.

We will be walking through the execution of a very simple multiplication program. It multiplies 3 and 5 and outputs the product to the console. The code for this simple program is shown in listing 3.1.

Listing 3.1: A very simple multiplication program

def multiply(x, y)

x \* y

end

puts multiply(3, 5)

# Outputs: 15

There is nothing very surprising in the code of listing 3.1. It does exactly what it’s told, multiplying 2 numbers and outputting the result. Now when we issue the command rbx Listing3.1.rb things start executing. The first thing that happens is the interpreter gets started and initializes the DLR who in turn will host our Ruby engine. Next the RubyContext, our language engine, gets initialized and set as run-time engine for the DLR host. When this is complete it will parse the console command-line arguments, which in this case is a file. At this point the actual execution can start. The hosting application checks if there are command-line arguments that need to be set first, in this case there are none so it doesn’t need to set any to configure the environment. It figures out that we passed a file path to the command line and decides to run the file.

When the execution starts we are at the top of the diagram in figure 3.3. The script engine generates a DLR tree from the source code that is provided. We won’t be showing the DLR tree that is being generated because that is out of the scope of this book. It would beg too much explanation and really warrants its own book. There is a Ruby compiler that parses the source file and transforms that parsed code into an AST, the DLR tree. Once the DLR tree has been generated it will be optimized. At this moment the application is ready to start looking up the language specific rules for executing the DLR tree. The Ruby language is implemented using those rules, so just about every operation needs a look up. This lookup process uses some smarts for resolving the requested rules. When an operation is repeated a number of times the script engine will generate the IL for it directly and cache that. At this point execution of the code is complete and the result will be shown in the console in this case.

Now that we know how IronRuby and the DLR fit together, it is time to discover how we can host the DLR in our own applications and communicate with the ruby code that we are executing.

3.3 Hosting the DLR in your applications

The DLR team has prepared a couple of hosts for you to which we’ll return later in this chapter, but first we need to talk about hosting the DLR in one of your own applications so that you’re able to extend your existing .NET applications with IronRuby if you so desire.

In addition to the Shared Dynamic Type System, the DLR also has a shared dynamic host system, which takes care of executing code in the DLR from your application. These allow you to execute your IronRuby code in a managed application, for example. They can also allow your code to interact with libraries and share types with the application that is hosting the DLR. Microsoft provides a couple of precooked hosts, including hosts for Silverlight, ASP.NET, Console applications, and so on.

This hosting infrastructure is also responsible for providing intellisense inside an IDE like Visual Studio. This integration is what should give you intellisense, a rich debugging experience, and more nice goodies.

3.3.1 Bird’s-eye view of hosting the DLR

You can host the DLR in your own applications, which means that you can also get full dynamic language support in your own application. Hosting the DLR revolves around creating one or more ScriptRuntimes and engines. It allows you to execute arbitrary strings of IronRuby code, IronRuby files; in fact it allows you to use any language that is built on the DLR like IronPython, IronLisp or managed JavaScript. From now on I will talk about IronRuby but you can just use that to refer to any of the DLR languages.

The hosting API allows you to isolate the dynamic code execution by 3 different mechanisms, each with its own sharing boundaries and nuances.

* The first mechanism and also the most heavyweight are .NET AppDomains. An AppDomain is the way the CLR provides isolation for a set of assemblies or an application. AppDomains can be individually configured and secured. They also make it possible to unload all the assemblies without affecting the rest of the code. You cannot directly communicate with objects that live in a different AppDomain instead you have to marshal the object by reference or value. The DLR allows this communication across AppDomain boundaries and makes it fairly straightforward to do so. You can almost look at AppDomains as different applications.
* A second mechanism for isolating code execution is creating different ScriptRuntimes. A ScriptRuntime is like a whole separate ruby engine with it’s own references to .NET assemblies and namespaces in those assemblies. This is more lightweight than managing different AppDomains. It also allows for different components to run in isolation within the same AppDomain so you don’t have to manage juggling potential conflicts and so forth.
* The third and last level of isolation are ScriptScopes, this is the most lightweight of the 3 mechanisms but doesn’t provide as much isolation as a ScriptRuntime does. A ScriptScope provides variable binding isolation, so you can execute code that uses the same variable names in different scopes in the same runtime and get different results because of the scope isolation. Figure 3.4 shows an overview of the possible isolation levels.

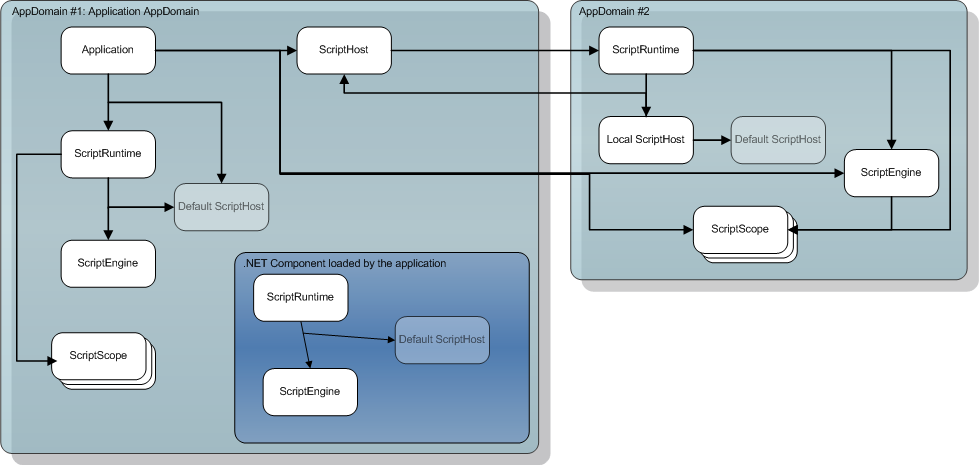


Figure 3.4 Overview of the hosting components and their isolation levels

If the above paragraphs seem just a little bit too abstract to go off and start hosting the DLR in your own applications, then that is about to be remedied. The next section takes a more practical approach to hosting the DLR. We’ll start off with a very simple example and work our way up to the more advanced hosting scenarios.

3.3.2 Hosting the DLR in a .NET application

This section discusses the practical side of hosting the DLR. We start with a very simple example and will end with an example that communicates across AppDomain boundaries. If you find that the hosting scenarios described in this section don’t cover your needs, there is a complete DLR Hosting spec available on the internet that goes beyond what will be discussed here. You can find a list of DLR resources on this page: http://blogs.msdn.com/ironpython/archive/2008/03/16/dlr-resources.aspx

Code listing 3.2 shows how to host the DLR in a console application and then have it execute some ruby code in the context of that application. This was the simplest scenario we could come up with but they will get more useful as this chapter progresses. To follow along with the code listings below you need to create a new Console application in C#. The next step is to add a reference to Microsoft.Scripting.dll. This dll can be found in the folder where you have the build of IronRuby, in our case that is C:\tools\IronRuby\build\debug as shown in chapter 1. If you want to use IronRuby you will need to add 2 more references one to IronRuby.dll and one to IronRuby.Libraries.dll. You are now ready to start using IronRuby in the C# application.

Listing 3.2: Hosting the DLR in one of your applications

ScriptRuntime runtime = IronRuby.CreateRuntime();

Console.WriteLine("Executing from file:");

runtime.ExecuteFile("hello\_world.rb");

Console.WriteLine("Press any key to close...");

Console.ReadKey();

// Outputs the following:

//

// Executing from file:

// hello, world!

The code listing 3.2 above shows how to host the IronRuby engine inside a runtime (DLR). We get the IronRuby ScriptRuntime and tell it to execute the “hello\_world.rb” file. The hello world file contains one line of code: puts ‘Hello, World!!!’. It might be worth noting that for in order to work you need to set the *Copy to Output Directory* property on the hello\_world.rb file to *Copy if newer*. If you hit F5 or run the application you will see that it executes the ruby file. That was easy enough and that is the base of what we need to know at this level. Let’s move on and see how we can do host-script-host communication as demonstrated in listing 3.3.

Listing 3.3: Passing a list of strings to a ruby script created from a string and execute that script

//TODO: update when hosting API changes

ScriptRuntime runtime = IronRuby.CreateRuntime();

ScriptEngine engine = IronRuby.GetEngine(runtime);

ScriptScope scope = engine.CreateScope();

scope.SetVariable("txt", "IronRuby is awesome!");

scope.Execute("def self.upper; txt.to\_upper; end;");

string result = scope.GetVariable<Function<string>>("upper")();

Console.WriteLine("The result is: " + result);

Console.WriteLine("");

Console.WriteLine("Press any key to close...");

Console.ReadKey();

// Outputs the following:

// The result is: IRONRUBY IS AWESOME

The listing above is slightly more complicated and demonstrates how we can control the scope in which the script will be executed within the DLR. It also shows how we can just take a string as a ruby script and execute that in with the IronRuby engine. In the first line we make sure that we have a DLR runtime in our application. The next line gets our IronRuby language engine. On the third line we create a ScriptScope that we’ll use to communicate between IronRuby and our application. At this point it becomes a little bit more interesting because we set a local variable txt with a string. And now we can finally execute some script. To do this we call the method Execute on the scope. The script we’re going to execute defines a class method that transforms the string contained by our variable txt to upper case. After we’ve executed the ruby script we get the method upper from the variable dictionary and invoke it. Console.WriteLine then prints that result.

Not super impressive but we can see some useful scenarios emerging here. It looks like we cannot only share variables but we can even share function pointers (delegates) around. That opens up the possibility where you can attach ruby code to event handlers, which might be valuable at some stage. Listing 3.4 shows how to attach ruby code to event handlers.

Listing 3.4: Attaching ruby code to C# event handlers, our string adder class

public class StringAdder

{

private readonly List<string> \_stringList;

public delegate void OnStringAddedDelegate(string addedValue); 1

public event OnStringAddedDelegate OnStringAdded; 2

public StringAdder()

{

\_stringList = new List<string>();

}

public void Add(string value)

{

\_stringList.Add(value);

OnStringAdded(value); 3

}

public override string ToString()

{

return string.Join(", ", \_stringList.ToArray());

}

}

1 Define delegate

2 Publish event handler

3 Raise event

Listing 3.5 shows a StringAdder class. This is an object we need to raise an event. We define a delegate OnStringAddedDelegate #1 that takes 1 string parameter. We also need to publish a hook so that we can subscribe to the event when it’s raised; this is OnStringAdded #2. Our class has an private \_stringList field which we’ll use to store our strings in. It also overrides ToString to show the contents of the list as a comma-separated collection. And lastly the StringAdder class defines an Add method where we add the value of a string to the list and raise the OnStringAdded event #3. Listing 3.5 shows the code that consumes the StringAdder class and attaches some ruby code to the OnStringAdded event.

Listing 3.5: Consuming the StringAdder class and attaching a ruby event handler

ScriptRuntime runtime = IronRuby.CreateRuntime();

ScriptEngine engine = IronRuby.GetEngine(runtime);

ScriptScope scope = engine.CreateScope();

StringAdder adder = new StringAdder();

scope.Execute("def self.string\_added(val = ''); puts \"The string \\\"#{val}\\\" has been added.\"; end;");

StringAdder.OnStringAddedDelegate functionPointer = scope.GetVariable<StringAdder.OnStringAddedDelegate>("string\_added");

adder.OnStringAdded += functionPointer;

Console.WriteLine("Initialisation complete. About to add some strings.\r\n");

adder.Add("IronRuby");

adder.Add("IronPython");

adder.Add("VBx");

adder.Add("Managed JavaScript");

adder.Add("IronLisp");

adder.Add("IronScheme");

adder.Add("Nua");

Console.WriteLine("\r\nSome DLR languages: " + adder);

// Outputs the following:

//

// Initialisation complete. About to add some strings.

//

// The string "IronRuby" has been added.

// The string "IronPython" has been added.

// The string "VBx" has been added.

// The string "Managed JavaScript" has been added.

// The string "IronLisp" has been added.

// The string "IronScheme" has been added.

// The string "Nua" has been added.

//

// Some DLR languages: IronRuby, IronPython, VBx, Managed JavaScript, IronLisp, IronScheme, Nua

Listing 3.5 first creates a DLR runtime; next it fetches the IronRuby language engine. We also need to create ScriptScope and an instance of the StringAdder class. We are now ready to execute some ruby script that defines a class method string\_added, which takes one parameter. The parameter defaults to an empty string if none was provided. The string\_added method outputs *The string “string value” has been added.* We then get that method from the scope as an OnStringAddedDelegate. That’s all we need from IronRuby at this stage and we attach the function pointer to the OnStringAdded event. We then tell the user we’re going to add strings. We add the strings to the adder object and in the end we show the user a list of DLR languages.

This covers the basics of hosting the DLR in your own applications and might be all you ever need from the hosting API. The next part of this chapter will provide an overview of the technologies we’ll be using later on in the book. that the following sections should give you an idea of how you might go about building a real-world application using IronRuby. The first technology we’ll cover is Windows Presentation Foundation (WPF), a framework for developing desktop applications.

3.4 WPF and IronRuby make a nice pair

Windows Presentation Foundation (WPF) is being used to create rich and stunning desktop applications. It was designed to provide a uniform and easy way of creating nice effects and having a totally customizable toolset.

WPF is vector based for the most part which means that no matter how big the screen is the images etc will always look crisp and beautiful. It is very capable of doing 3D animations and such because it uses the GPU of the video card so that doing animations doesn’t block the main CPU. It is fully hardware accelerated which means that you enjoy the same power a game does. I think that’s pretty cool stuff. And what makes it even cooler it’s really easy to use it from IronRuby.

WPF uses a markup language called *eXtensible Application Markup Language* (XAML), which allows for a bunch of tasks to be done declaratively. We’ll look into XAML a little bit closer in a minute. In addition to using the XAML format (which is by far the easiest way to do things in WPF), you can use any .NET language, including IronRuby, to build up WPF-based Graphical User Interfaces (GUIs).

Note

There is one caveat at the time of this writing: Mono doesn’t support WPF completely, but that will change in the future.

The knowledge you gain from using WPF is mostly transferrable to Silverlight. Silverlight or Moonlight on Mono understands XAML, but runs inside the browser (currently supported: Firefox, Safari, Internet Explorer). Silverlight is a really cool piece of technology and we’ll be investigating that properly later on in Chapter 4.

Explaining everything you can do with WPF is out of the scope of this book. There are several really good books on this subject, like Windows Presentation Foundation in Action from Manning, Silverlight 2 in Action from Manning or Windows Presentation Foundation Unleashed from Sams Publishing.. We will, however, discuss the possible ways of running WPF with IronRuby, and we’ll use those techniques later on in Chapter 4 when we’ll be building an application. The first thing I want to walk through with you is XAML.

3.4.1 Xaml developers reaching out to designers

You must be thinking, “Oh my gosh!!! I thought this was a book on IronRuby and so far I’ve seen a lot of samples in C# and it looks like they’re going to be in some form of XML.” You are correct to be assuming this, but we want to assure you that this book is about IronRuby, and the samples from now on will be almost all in the Ruby language. However, we want to show you some samples in other languages to demonstrate that you can use them together with IronRuby.

There are a couple of reasons why we want to talk about XAML in this book:

* First, XAML is a way for interfaces to be created by designers in Illustrator, for example, and then through a tool they can be turned into XAML. The resulting XAML can then be used immediately by you, the developer, to make the interface do some actual work. Using XAML has some distinct advantages over most imperative programming languages, but anything you can do in XAML can also be done with an imperative programming language like IronRuby (thoughnot vice versa).
* XAML is usually the most concise way to represent user interfaces and other hierarchies of objects. Because of the fact that it is XML based, one of its strongest points is probably the ease of representing highly related data. Using XAML promotes a clear separation between your front-end appearance and your backend logic, which provides easier maintainability.
* XAML is primarily used for creating and initializing objects. Because interfaces represent mostly very hierarchical data, XML is a good fit for representing that data because the nesting of tags can mimic the hierarchy easily.

That being said we want to repeat that anything you can do in XAML, you can also do in IronRuby. Next, we’ll take a closer look at some of the possible ways you can use WPF from IronRuby.

3.4.2 A simple browser example

In this section, we’ll first show an example in XAML of how you could create your own simple browser. Then we’ll take a look at a translation of that XAML to IronRuby. In Chapter x, we’ll take a closer look at how I think IronRuby and XAML will be used to develop desktop applications, which is a kind of middle ground between both worlds.

The first example we’re going to see a layout created in XAML (listing 3.6). Then we’ll load that into an application from IronRuby. The second example is using IronRuby to build up the layout from code.

The example we’ll be creating is a simple browser application with an address bar, a button to fetch a page and a frame that will display the webpage. Figure 3.5 shows our browser in action.

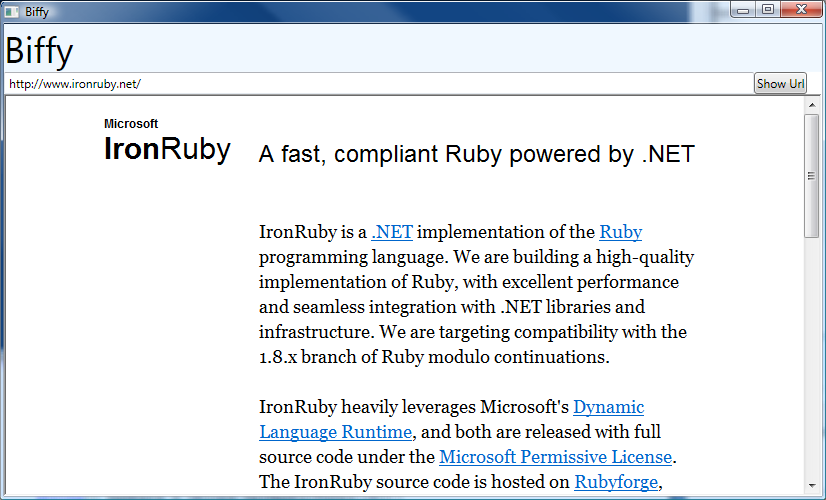


Figure 3.5: Our Browser showing the ironruby website

define the gui elements

In this example, we’ll first examine the layout created in XAML. Listing 3.6 shows us the XAML that describes the window we want to display by nesting a dock panel in the window, showing a title bar (TextBlock) and a stack panel with the actual GUI elements.

Listing 3.6: The XAML for defining the layout of the browser application

<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Biffy" Height="500" Width="826" x:Name="root" >

<DockPanel x:Name="dock\_panel">

<TextBlock FontSize="36" Background="AliceBlue" DockPanel.Dock="Top">Biffy</TextBlock>

<StackPanel Orientation="Horizontal" DockPanel.Dock="Top">

<TextBox x:Name="web\_url" Width="750">http://www.ironruby.net</TextBox>

<Button x:Name="get\_url\_button">Show Url</Button>

</StackPanel>

<Frame x:Name="web\_url\_display" Source="http://www.ironruby.net"></Frame>

</DockPanel>

</Window>

Use ruby to make it do stuff

Now, we’ll use IronRuby to load the XAML into an application. Listing 3.7 shows how we would use the previously created XAML from IronRuby.

Listing 3.7: The Ruby code to show the XAML and attach an event to the button

p "loading components, please wait..."

# get the necessary assemblies loaded for .NET 1

require 'mscorlib'

require 'System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'

require 'System.Xml, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'

require 'PresentationFramework, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'

require 'PresentationCore, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'

# include some namespaces for easy access 2

include System

include System::Windows

include System::Windows::Controls

include System::Windows::Markup

include System::Windows::Media

include System::Windows::Media::Animation

include System::Xml

# define an application 3

class XamlApplication < Application

def initialize

run yield

end

end

#run the application 4

xaml\_path = "Biffy.xaml"

XamlApplication.new do

obj = XamlReader.load XmlReader.create(xaml\_path)

obj.find\_name('get\_url\_button').click do

obj.find\_name('web\_url\_display').source = obj.find\_name('web\_url').text

end

obj

end

1 load the .Net framework libraries

2 include some helpers

3 define an application

4 run the window and attach the click event

In listing 3.7, there are 4 important sections. This is also the first time that we actually get to use .NET from the Ruby language so let’s investigate #1 a little bit closer. To get access to the basic .NET types we need to load the file mscorlib, but that is just a very limited set of classes the absolute minimum that is required to have .NET going on a machine. So we load a couple of extra assemblies where the assemblies whose names start with Presentation are WPF specific assemblies. This first step ensures that we have the necessary type information available for using WPF and XAML.

The second part #2 provides access to types that live in the specified namespaces without having to specify the fully namespaced type all over your code. So it allows you to write Application instead of System::Windows::Application.

On to part 3 then shall we. This is standard WPF programming. In #3 we create a class that inherits of Application to encapsulate our own application logic. The code that we have in that class expects a block for the constructor and that block should return a Window object which we’ll run.

The last section of the code in listing 3.4, section #4 is the code required to start our application. That is where the actual work is done. We first read in the XAML document which will return us the window object. Next we need to attach a click handler to the button that will change the url of the frame so that we can browse the internet. We gave the frame a name with the x:Name attribute and we use the find\_name method to get to the frame and set its url.

A pure ruby browser

Listing 3.8 shows how we would go about creating the exact same application, but by using the Ruby language to program the interface instead of XAML to describe it.

Listing 3.8: Creating the browser application by using only Ruby

p "loading components, please wait..."

# get the necessary assemblies loaded for .NET

require 'mscorlib'

require 'System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'

require 'System.Xml, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'

require 'PresentationFramework, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'

require 'PresentationCore, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'

# include some namespaces for easy access

include System

include System::Windows

include System::Windows::Controls

include System::Windows::Markup

include System::Windows::Media

include System::Windows::Media::Animation

include System::Xml

my\_window = Window.new

my\_window.title = 'Biffy'

my\_window.height = 500

my\_window.width = 826

my\_dock = DockPanel.new

my\_window.content = my\_dock

header = TextBlock.new

header.background = Brushes.AliceBlue

header.font\_size = 36

header.text = "Biffy"

DockPanel.set\_dock header, Dock.Top

my\_dock.children.add header

start\_url = "http://www.ironruby.net"

top\_stack = StackPanel.new

top\_stack.orientation = Orientation.Horizontal

feed\_url = TextBox.new

feed\_url.text = start\_url

feed\_url.width = 750

DockPanel.set\_dock top\_stack, Dock.Top

load\_feed\_button = Button.new

load\_feed\_button.content = "Show site"

top\_stack.children.add feed\_url

top\_stack.children.add load\_feed\_button

my\_dock.children.add top\_stack

frame = Frame.new

frame.source = Uri.new start\_url

my\_dock.children.add frame

load\_feed\_button.click do |event, args|

frame.source = Uri.new feed\_url.text

end

my\_app = Application.new

my\_app.run my\_window

By just using imperative code, we immediately lose some of the advantages we gained from XAML:

* We don’t have an overview at-a-glance of how the hierarchy of the interface works.
* How did those couple of lines of XAML become so many lines of Ruby code? Wasn’t XML supposed to be the verbose language and Ruby the beautiful language?

What’s going on here?

We are working with the direct API’s of windows presentation foundation. These have been built purposely to accommodate for XAML, in fact, the API’s are often a consequence of the fact that it uses XAML. With Ruby, of course we can do better than just using the native API’s.

Adding some ruby love

Why don’t we sprinkle some meta-programming over that code and see where we end up. Listing 3.9 shows the result. For brevity, I haven’t included the code that makes that possible here, but it is available with the code samples for this book or with the samples that you can download from <http://dynamicsilverlight.net>.

Listing 3.9: Programming a wpf interface after meta-programming

p "loading components, please wait..."

require 'wpf'

require 'wpf\_elements'

class WpfApplication < Application

def initialize()

run yield

end

end

start\_url = "http://www.ironruby.net"

title = "Biffy"

WpfApplication.new do

# build the window

obj = Wpf.build Window, :title => title, :height => 500, :width => 826, :name => "Biffy" do

add DockPanel, :name => "dock\_panel" do

add TextBlock, :text => title, :font\_size => 36, :background => :alice\_blue, :dock => :top, :name => "text\_block"

add StackPanel, :orientation => :horizontal, :dock => :top, :name => "stack\_panel" do

add TextBox, :text => start\_url, :width => 750, :name => "web\_url"

add Button, :content => "Show site", :name => "get\_url\_button"

end

add Frame, :source => start\_url, :name => 'web\_page\_display'

end

end

# attach the event handler

obj['get\_url\_button'].click do

obj["web\_page\_display"].source = obj["web\_url"].text

end

obj

end

In listing 3.9 above we do exactly the same thing, but in this case I’ve used a module that John Lam built for a presentation at RubyConf to build up the interface. I just had to put this code in to show that there is still hope left. This preserves the fact that we can see the hierarchy of the interface, and it’s slightly less verbose than the xml we generated. This has an extra advantage: the CLR doesn’t have to perform a XAML parse and compilation step before it can display the interface. If I were building a WPF app, I would probably combine the first and last approach.

This concludes our brief introduction to what WPF is and how to use it from Ruby, but there are many avenues we haven’t explored yet. The book isn’t over yet and we’ll revisit WPF in chapter x. Next, we’ll take a brief look at Silverlight.

3.5 Join me on the Silverlight bandwagon

If we would have to describe my opinion about Microsoft Silverlight in one word I would have to say: Awesome! Out of all the technologies discussed in this book, I think Silverlight takes the cake. I want to convey this feeling to you because this is where I think web development gets so much more fun and cool at the same time. But let’s first start by explaining what Silverlight actually is and then moving on to why it’s so cool.

We can’t get around the compulsory definition for this one, here it goes: Microsoft Silverlight is a cross-browser, cross-platform plug-in for delivering the next generation of .NET-based media experiences and rich interactive applications for the web.

That’s cool, but bear in mind that at the time of this writing, Silverlight runs on Windows and Mac OSX, but not on Linux. Again, we’ll mention Mono for people that are running Linux. Mono has an application called Moonlight, which is roughly the same thing. Silverlight 2.0 and Moonlight should be completely compatible with each other. I think this is a pretty sweet deal because you can develop your applications and be sure they will work in IE, Safari and Firefox without even having to worry about any inconsistencies that exist in the different browser implementations. Another cool aspect is that you get to program both the server-side and the client-side with the same programming language: IronRuby.

So let’s take a closer look at what’s in Silverlight to find out a little bit more about its architecture and different components.

Note

Just to be clear: we are talking about Silverlight 2.0because Silverlight 1.0 does purely JavaScript and HTML.

3.5.1 What’s Silverlight made of?

The development platform the Silverlight plugin provides has 2 major parts:

* Core presentation framework—provides components and services to build your UI. It contains controls for user input, media playback, and so on. In addition to the controls, it also provides a DOM API for the Canvas and XAML support for creating your interface.
* .NET Framework for Silverlight— a subset of the .NET framework that contains a bunch of libraries for common tasks, the DLR and the CLR.

The ability to use a language like IronRuby to program both the server side and the client side is one of the most anticipated features for me personally, followed closely by the ability to program on top of the CLR and have everything expressed with WPF controls. Although we consider all the components to be fairly important, the ones we mentioned above are those that motivated us to take a really close look at what Silverlight could mean for us. We’ve listed all the components of Silverlight in table 3.1.

Table 3.1: The components of Silverlight

|  |  |
| --- | --- |
| Core Presentation Framework |  |
| Input | Handle input from hardware devices |
| UI Rendering | Renders vector and bitmap graphics, animations and text |
| Media | Features playback and management of audio and video files |
| Controls | Provides controls to rapidly built applications |
| DRM | Enables DRM on media assets |
| XAML | Provides a parser for XAML markup |
|  |  |
| .NET Framework For Silverlight |  |
| WPF | A subset of the full blown WPF framework for the desktop |
| Data | Support for LINQ and XLinq, to ease working with data |
| BCL | Provides essential programming functions |
| Networking | Simpler access to remote services and data |
| CLR | Common Language Runtime for memory management etc. |
| DLR | Support for Dynamic languages on the CLR |
|  |  |
| Additional API Features |  |
| Isolated Storage | Enables safe local storage and caching of data for a particular user |
| Asynchronous programming | The work is done on a background thread keeping the UI responsive |
| File management | Provides a safe File Open dialog box to make uploading easier |
| HTML – managed code interaction | Enables DOM manipulation through a .NET API |
| JSON support | Allows access to ASP.NET web services |
| POX services | Enables calls to simple XML based web services |
| XML Libraries | XMLReader and XMLWriter to simplify working with XML Data |

We think this is a pretty compelling list of features for web development; it solves some of the problems that the larger applications and sites are struggling with, like isolated storage for restricted access to the local file system of the person viewing your site.

You can just program Silverlight using IronRuby, which is a big bonus if you ask me. This means that both the client and server side can be programmed in the same language. What’s more is that you can mix and match languages. An IronRuby file, for example, can require a vbx file and access all the types and methods etc created by that file; very handy under the best tool for the job motto.

3.5.2 Is Silverlight for me?

If you’re doing web development you should definitely care about Silverlight. It allows you to target the major browsers without having to worry about how things will look in each of them. It will always look the same. The same goes for providing a richer interaction with the user. In recent years, web applications have used HTML and JavaScript through a technique called AJAX for providing those experiences, often in combination with Adobe Flash. There are a number of problems with the current approach of developing those rich internet applications:

* It makes development pretty complex because, if you want to stand your ground as a web developer, you need to be able to program in at least 5 languages: HTML, JavaScript, CSS, SQL, Server side language (Ruby, Python, C#,…).
* Every browser implementation has its own ideas around how html and JavaScript should be implemented, which often makes an application run really well in one browser but not that good or not at all in another one.
* Often the JavaScript engines are slow so there is a limit as to how much functionality you can put in your application before it gets hairy. JavaScript is single threaded and sometimes you just need more. Flash can be used to fill that gap, but that adds yet another language (ActionScript) to your requirements.

In short, we could use some innovation in this area. Silverlight could just be it.

3.5.3 How do I use it?

Now that we know what Silverlight is and why it’s poised to be a big thing soon, maybe it’s time to check out how to use this thing. As an example, we’re going to build a client that will consume a flickr feed and display the pictures in an animated slideshow.

Providing the layout

In listing 3.10 we have the simplest example of a Silverlight application we could think of: our good old friend, Hello World for Silverlight.

Listing 3.10 XAML for hello world in Silverlight

<Canvas x:Class="System.Windows.Controls.Canvas"

xmlns="http://schemas.microsoft.com/client/2007"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Name="parent\_canvas">

<TextBlock x:Name="textblock" FontSize="30">Hello world from XAML </TextBlock>

</Canvas>

As you can see, we’re not practicing rocket science here. We place a Canvas control as our root element for our Silverlight app, and in that canvas, we put a Textblock that, surprisingly enough, displays “Hello world from XAML”.

Adding behahvior with ruby

Next, we need to write some code (listing 3.8) to use the xaml in listing 3.7 in IronRuby, and to change the text of the textblock.

Listing 3.11 The Ruby needed to drive the XAML

include System::Windows

include System::Windows::Controls

xaml = Application.current.load\_root\_visual Canvas.new, "app.xaml"

xaml.find\_name('textblock').Text = 'Hello world from IronRuby'

In the listing 3.11, we first include the namespaces we are going to use. Application lives in System::Windows and Canvas lives in System::Windows::Controls. Next, we load the XAML as our visual root control in Silverlight, which we follow by finding the TextBlock instance and setting its text to “Hello world from IronRuby”.

We believe that Silverlight will be a significant player on the web in a couple of years and that the strength will come from combining it with html based pages. In chapter x, we will be using Silverlight to consume a couple of API’s and displaying their data.

In the next section, we’ll see how to create web pages with ASP.NET, the combined services for the web from the .NET Framework.

3.6 Developing for the web with ASP.NET

ASP.NET is a set of components and services that are used for doing web development with the .NET Framework. Silverlight alone isn’t enough to develop web applications. We still have to face generating HTML.

The .NET Framework includes two components that accommodate this task:

* Webforms
* an MVC Handler

In this book, we will use the tools provided by the Rails framework side by side with the MVC approach for IronRuby. The code samples also include an implementation that uses Webforms. We prefer to use an MVC type model when we’re creating websites, but realize that some people will want to see how the code looks in a classic ASP.NET website. Let’s take a closer look at the differences between these approaches.

3.6.1 Webforms: using ASP.NET the classic way

From the start, the .NET Framework has supported using Webforms for developing web applications. Webforms were specifically designed to ease the transition for windows developers to the web. It uses controls and a concept of a page life-cycle in combination with ViewState and ControlState to make the web appear stateful.

You don’t have to use any of the mentioned components if you don’t want to but then it does get a bit tedious when you have complex forms. I have been developing web applications for about 10 years now and I like the webforms way of doing things if you have a small application where you don’t need a lot of AJAX interactivity. For more complex applications I look for an MVC framework like Castle/MonoRail, Spring.NET or ASP.NET MVC.

Let’s look at an example on how we can bind a grid to show us the content of an XML document. The document only holds a list of names of people and their age. You can do this completely declaratively but we want to demonstrate how to use some of the controls from code.

TODO CODE LISTING FOR AN ASPX PAGE WHEN THERE IS SUPPORT FOR IT

And that’s how you would go about creating web pages with ASP.NET and IronRuby. That’s already pretty cool, but there is an alternative to webforms. This alternative is called ASP.NET MVC and that’s what we’ll be talking about next.

3.6.2 MVC: using ASP.NET the new and improved way

Very recently, Microsoft has been building an MVC (Model-View-Controller) handler for ASP.NET. Using an MVC approach to web development makes sense and improves testability. I think it’s always good to have better and easier ways to test your work. It also forces you to have a very clean separation of concerns. By that we mean maintaining your state in your models, often that state is persisted to the database. Views are representations or the components that display the data. And the Controller part takes care of handling user input, passing that on to the models and then it decides which view it’s going to render with the data it collected from the models.

I personally have been developing applications with this methodology on .NET for the past 3 years and really like it. I definitely like the ASP.NET MVC Framework because it’s highly extensible. You have full control over how you want the MVC Framework to behave. And it integrates seamlessly with existing ASP.NET applications. You still get to use controls if you want to but there is no more ViewState and you don’t have to use them. There are already a couple of alternative view engines for the ASP.NET MVC Framework. These can be found in the mvccontrib project on codeplex.

Now that we know what the MVC Framework is and how it’s a very nice way of implementing web applications. Let’s convert our previous webform page into an MVC page.

[[content underdevelopment]]

3.7 Summary

This chapter took us very briefly through what the .NET Framework is and what it can mean to you when you’re developing IronRuby applications. I tried to highlight the technologies we will be using from now on in the book with some basic examples as to how the code might look. Again for a more in depth coverage of each one of the subjects there are a number of really good books. Trying to explain everything is out of the scope of this book.

We’ve seen how the .NET Framework allows you to run code written in different languages on different platforms. How the CLR translates that code into CIL that gets compiled into native code optimized for the platform it is running on. We also saw how the DLR plays a crucial role in the implementation of Dynamic Languages on the CLR. We now also know that a type created in IronRuby is exactly the same as a type created in IronPython or C#. This is important for sharing code around.

Next, we took a closer look at the new technologies we’ll be using in this book to develop an application. Those technologies are specific to the .NET Framework and should make it a lot easier to develop really compelling applications on all kinds of platforms.

We investigated WPF and what its benefits are, like hardware acceleration, separation of UI components of backend logic etc.

We then took a closer look at Silverlight and how Silverlight takes a subset of WPF and brings it to the web. These are really powerful technologies and I think we’ll see some great things that are built with those tools.

And the last subject we discussed was ASP.NET. ASP.NET is the integration of .NET with a webserver like IIS or apache. We saw how we can develop ASP.NET Applications using 2 different methodologies—Webforms and MVC—with a short example using each of them.

In the next chapter, we’ll take a deeper look at WPF by using it to build a basic Twitter client. We will see what is involved in consuming such an API with IronRuby.