6

ASP.NET MVC: Putting you in control

* Know about filters
* Know about major ASP.NET components
* Know about the MVC pattern

When I started developing on the .NET platform I came from ASP with Javascript and basically went from a stateless model of programming to a model that pretended to be stateful, ASP.NET WebForms. Over the years I’ve seen some pretty horrible consequences from that model like a view state that contains 25MB worth of data for example, which get sent back and forth over the wire. So I welcomed a more web-friendly way of building .NET web applications with open arms.

I had been in anticipation of the first release of ASP.NET MVC when I first heard about it and also contributed the XSLT view engine to the ASP.NET MVC contrib framework about 2 days after the first community technology preview was released. This view engine now exists as part of the mvc contrib. project and can be found on github. Although many people complained that there weren’t many features in the ASP.NET framework that could be found in other ones, I actually liked it because it didn’t force me down a path that I did or did not want to take.

This chapter of the book will present a sample application, this sample application is nowhere near production ready code and functions as a starting point on how you might go about integrating IronRuby in your own projects. Many of the topics we’re going to cover will come back when we get to the chapter about IronRuby on Rails. The most prevalent topics when talking about any MVC implementation for the web are: routing, filters, models, views and controllers. We’ll discuss all of these shortly. Because the ASP.NET MVC framework is built on top of the ASP.NET technology we’ll start by looking at the most important and relevant components of the ASP.NET technology.

6.1 Anatomy of the ASP.NET technology stack

The .NET framework is huge, heck even ASP.NET in itself is a fairly large framework with facilities to do many of the most common tasks you need to perform when developing web applications. I won’t explain the low-level plumbing like what happens when a request arrives at IIS but we will look at some of the basic components that are key components of every web application.

6.1.1 Key extension points in an ASP.NET application life cycle

Every ASP.NET application is built on the same pipeline. That is a request always follows the same path giving you common points to extend an application. Figure 6.1 illustrates those extension points. It’s at these point you can subscribe to certain events in the lifetime of a request to add your own logic to it.

I’d like to use this illustration to explain where ASP.NET MVC is situated in that lifecycle. When a request arrives at the web server and ASP.NET takes over everything is the same for ASP.NET MVC as for WebForms except that a different handler is used. Webforms typically used an IHttpHandler implementation called PageHandler, which knows how to operate on a certain base class (Page) and process the logic contained in a classic WebForm. ASP.NET MVC, like WebForms, hooks into ASP.NET by using a Handler. And from then on the ASP.NET MVC framework starts executing.

Before a handler executes the request passes through a series of modules, one such module is responsible for providing the FormsAuthentication infrastructure. Another one is responsible for managing Session state. The request then reaches the ASP.NET MVC handler, a controller gets instantiated and it starts processing the request. When the controller is finished a rendered view is pushed down the pipeline and passes through a set of modules before being sent to the browser.

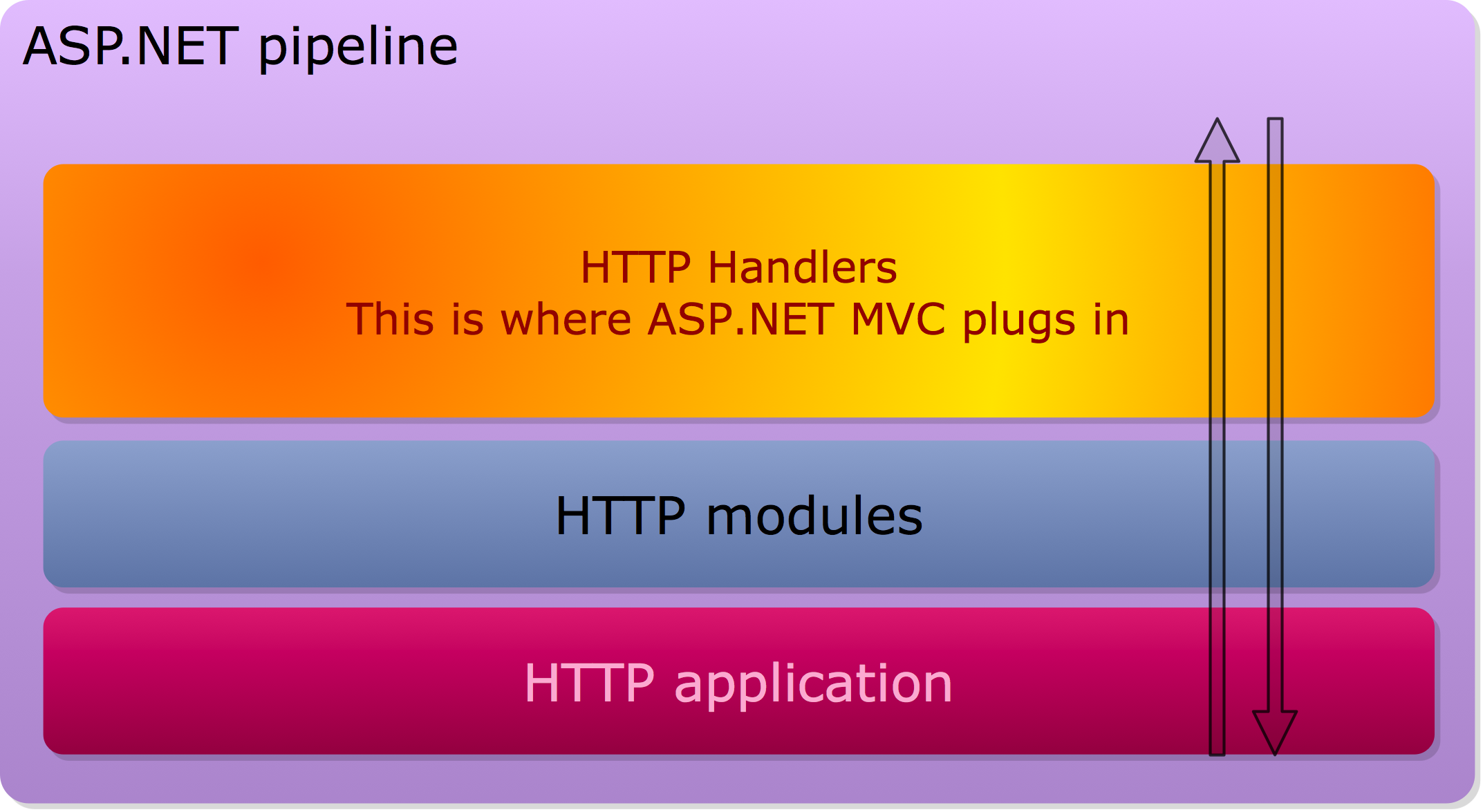


Figure 6.1: Key extension points for an ASP.NET application

Now that the request pipeline and how ASP.NET MVC hooks into the ASP.NET infrastructure is clear to us, we might have a look at some of the key components in an ASP.NET MVC application. The controller, model and view and how they fit together.

6.1.2: Key components in and ASP.NET MVC application

At this point it might be worth taking a look at the major components that are being used by the handler and that make up the letters M, V and C in the pattern name. Let’s first introduce all the components and then later look at how they fit together in the context of a request. Figure 6.2 illustrates the components and how the control flows between each of them. You will see that controller is a crucial pivotal point in your application.

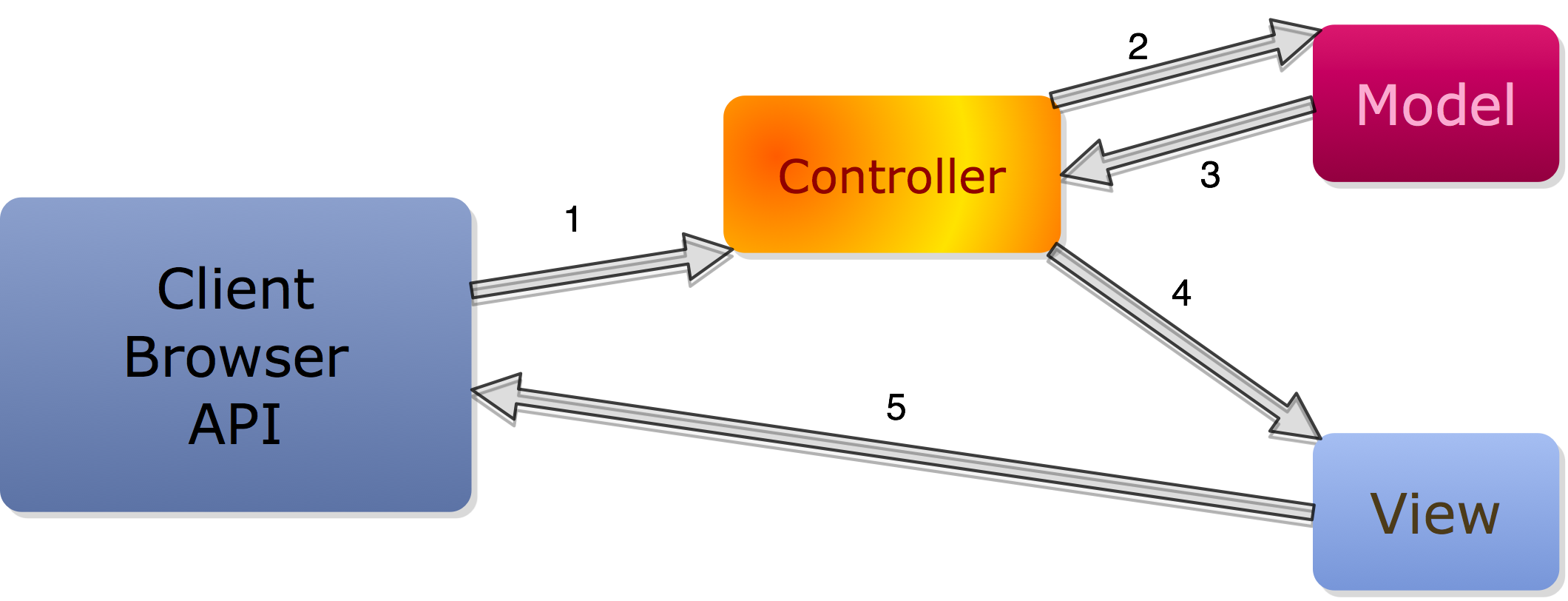


Figure 6.2: Key components of an ASP.NET MVC application

Routing

When a request arrives at the server, the URL gets mapped to a controller and it’s appropriate action. This process is called routing and the default route that is implemented maps the first part of the path to a controller name and the second part to an action name. So an URL <http://somecoolsite.com/products/list> would map to the controller ProductsController and the action on that controller would be the List method implemented on that controller. After routing the request passes into the controller

Controller

The controller is an orchestrator and takes the input from the client, this can be in a query string, request headers, form variables, … and calls various models to process that input. When the controller has gathered the output it needs it directs its attention to the view and passes the gathered output to the view to render it.

All logic that is responsible for calling out and getting output back for the view belongs in the controller. A controller takes orders, delegates them and delivers them back to you, much like a waiter in a restaurant.

Model

The model is really an object that contains some form of state for your application. This doesn’t mean that the state has to be persisted to the database but it does mean that it does something with data and generally returns a result for one or more properties. This is where the processing belongs nowhere else. Any logic that isn’t directed at orchestrating models to get output for the view belongs in the model, but a little bit more about that later.

View

The view is actually just a means to display a given set of data. This component typically serves to generate html or JavaScript/JSON. So a view is generally built using some kind of templating engine that allows you to create html or JavaScript.

As a general rule of thumb I use: if the logic can’t be expressed by simple if/else statements and for loops it doesn’t belong in the view and as such should be moved to a model preparing it to send to the view.

Control flow

Ok now that we have a grip on what makes up the different components of an MVC application, it’s probably a good time to explain the numbers on figure 6.2.

1. The client sends a request to the web server where it arrives in the handler. The handler functions as a front controller and handles the routing to collect the necessary information about which controller should be instantiated by the controller factory. The handler then invokes the appropriate action, which it also got from the routing handler.
2. The controller orchestrates one or more models and passes input to these models
3. The models return their values to the controller who prepares them to be sent to the view.
4. The controller is finished preparing and aggregating values and is now ready to decide which view it wants to render and in which form. After having done so it passes the necessary data to the view and renders the template.
5. The controller is finished rendering, perhaps does some post-processing and passes the output back to the client.

MVC in real life

As it so happens you probably deal almost daily with some kind of MVC pattern in real life. To get to the full explanation the restaurant would probably be fairly fancy.

You arrive at the restaurant and the maître d’ will show you to your seat. In this case the maître d’ is the front controller, which takes care of the routing. Hopefully he will also send a waiter to your table (the controller). This waiter takes your order (input) and goes off to deliver your order to the kitchen. The kitchen is full of models aka. cooks, they prepare your platters and put them back on the counter. The waiter keeps an eye on that counter and when he sees one of his orders there, he goes to pick it up, perhaps takes a pepper crusher (post-processing) with him and delivers the food (view) to your table.

The last topic of discussion in this section is about some of the key objects available to you in the ASP.NET MVC application. Most of them are available in the controller and some of them are reachable in the view too.

6.1.3: Some of the key objects in an ASP.NET MVC application

Just about every web framework you’ll work with will wrap the environment variables provided by the request and web server in some kind of more friendly objects. ASP.NET is no different in that regard. We’ll look at the stock ASP.NET objects and will touch on how ASP.NET MVC uses them in the next section.

HttpApplication

This class is the entry point for your web application and as such encapsulates some application level variables. This class is implemented as a Singleton and calls a magic method Application\_Start once at the start of an application. This class also provides a bunch of methods you can override to hook into the application life cycle.

HttpServerUtility

This utility provides a set of helper methods that make it easier to work with the web server. One of those methods is mapping an absolute URL path to a path on the file system of the server.

HttpContext

This class encapsulates the entire context of the request and response, it also contains the server utility object as well as the session state object. If you want to know something about the request like headers, variables… this would probably be a good starting point to find it.

HttpRequest

I guess you have pretty good idea of what is going to be in this object, but just in case you don’t; this class encapsulates the environment variables that make up a request. Among its properties one finds the query string, form post data, the request URI and such. The controller and the handler use this object most frequently.

HttpResponse

And the last one I want to discuss at this point is the HttpResponse object. This object provides methods to write to the response stream that will be sent to the client. It also has facilities for you to set headers for the response. The view engine uses this object most often.

These components are essential to just about any web application and even if you don’t use them directly; it pays off to understand that the internals of your framework of choice probably use them to some degree. Now that we know some of the major components of ASP.NET you’re probably asking yourself why would I want to use ASP.NET MVC?

6.2 Why, oh why ASP.NET MVC?

Many developers in the .NET sphere that I talked with before ASP.NET MVC 1.0 had been released didn’t know why they would want to use ASP.NET MVC over standard ASP.NET WebForms. The second problem they face is that most of the MSDN guidance to date had been written from the point of view of an n-tiered application where you have ie.: a data and a business logic tier as well as a presentation tier. When we talk about the MVC architecture the tiers still apply but are distributed differently, this is not better or worse it’s just different. For example your data logic will most definitely be contained in the models as well as most of your business logic. Some of that business logic may spill into the controller but you should be careful with that.

While ASP.NET WebForms have been used to create some pretty large web applications successfully it’s not without its pain points. We’ll look at the areas where ASP.NET alleviates some of these pain points and how they are solved. The aim of this section is not to say how webforms are bad that discussion has been held on the Internet and is for you to decide.

6.2.1 Unit testing

ASP.NET MVC wraps most of the basic objects of a web application in classes so they can be stubbed or mocked out easily. This has a big impact on how you can develop your application in a test-driven way. You can now test the application from unit tests without requiring a webserver to provide you with the necessary objects. We’ve highlighted the importance of unit testing at the beginning of this book. Whatever makes it easier for me to feel warm and fuzzy I’ll take!

Unit testing is also a lot easier because of the clear separation of concerns enforced by the MVC pattern.

6.2.2 UI driven development

There is a trend in web development that uses a UI driven approach to the development process. They start by developing the interface with mocked data and once everybody is happy with the UI they start writing the code that is necessary to make the interface work.

This is probably the more efficient way of building an application, because during the design phase the only thing that needs to be changed is html and perhaps some mock data. These things are much easier to change than say a database schema or the way billing works. This works particularly well in MVC because the mock data can be created in the controller and the html can be done in the view templates and reused later.

6.2.3 Accommodation of AJAX web applications

ASP.NET MVC makes it very easy to respond with different output formats (JSON, HTML, XML, …) to a request depending on certain variables etc. We’ll look at an example of this later in this chapter. This means that the degree of potential code reuse is pretty high because you can use the exact same code and data but give it any number of different representations.

6.2.4 Flexibility

An overall benefit of using MVC is that you get a lot more control over how your application is built and how it does certain things rather than using a bunch of black box components that are outside of your control. An MVC application makes it easier to create loosely coupled applications and create flexible designs. Then there are some other benefits that we should look at.

ASP.NET MVC in particular lets you change the view engine. It allows you to replace the controller factory. It basically allows you to replace any one of its components by a custom implementation, making this a platform you can truly shape to be your own.

These are some of the advantages that ASP.NET MVC has to offer. At the time of this writing there is no WebForms story for IronRuby but there will be one shortly after IronRuby gets released. So I wanted to inform you about the choices ahead without the ability to provide samples for the WebForms version. Of course there are some downsides to ASP.NET MVC as well, which is what we’ll look at next.

6.3 All roses? Any of them thorny?

The previous section highlighted some of the benefits of using ASP.NET MVC but to give a balanced explanation I ought it important to also look at some of the downsides. Depending on your POV or your objective some of the advantages of ASP.NET MVC are in fact disadvantages. This part aims at putting some of these items forward.

6.3.1 Controls or lack thereof

Most of the Microsoft/.NET technologies involve event-driven programming. Almost everything in the UI triggers an event that gets handled by the code-behind. This is a good solution if you’re working in a stateful environment. To ease the transition from desktop to web development you may opt to choose for a solution that also supports this pattern of event-driven programming.

Something that is also common in most GUI technologies provided by the .NET stack is that they work with widgets in the form of controls that expose certain events to so you can program against them. These controls are absent in the ASP.NET MVC framework, a lot of the lack of controls gets made up by the possibility to write helper methods to generate html for you in a reusable way and across projects. Or by using the excellent jquery ui, ajax control toolkit, ….

6.3.2 RAD story or code volume

One of such advantages is the fact that you have more control. More control means that you probably have to do some plumbing. Depending on your goal it might be a good idea to go for something that requires virtually no coding. Maybe you need something really quick that you can throw away later. Perhaps you’re application doesn’t do anything except show data from a database. Or maybe you just don’t want to write any code. In any of the aforementioned cases the WebForms alternative or dynamic data might be a good solution for you. We’ll look at another case next, the case in which you have an existing codebase written in ASP.NET WebForms.

6.3.3 Existing codebases

You or your company may already have made a serious investment in the ASP.NET WebForms platform in the last few years. Perhaps you want to migrate but can’t justify an entire rewrite of the application. New functionality and pages can be added to the ASP.NET WebForms application but using ASP.NET MVC controllers and views because you can use both side by side. You may even agree to move existing pages to the MVC model gradually by moving them over whenever they need to be touched. There is a special case that we haven’t discussed yet and that is the case in which you already have existing ASP.NET MVC components but they are written in C#.

6.3.4 Existing CLR ASP.NET MVC codebases

The way IronRuby MVC is implemented makes it compatible with existing components created in another language like C#. That means you can reuse all of your existing logic as is and you can add new logic using the wonderful Ruby language. This has proven to be an advantage when working with a CLR OR/M tool like we’ll be doing in this chapter.

Phew this again concludes the theoretic base for this chapter and we can now move on to a more practical exploration of the ASP.NET MVC framework with IronRuby. During the remainder of this chapter we’ll build a very simple chat application that updates the conversation using AJAX.

6.3.5 ASP.NET MVC isn’t a full stack

One of the critiques on ASP.NET MVC is that it isn’t a full stack MVC framework. While that is a valid argument for me the fact that it isn’t a full stack and the wide variety of tools you can choose from within the .NET community or the Ruby community to complete your stack make this a non-issue because instead of using tools I’m totally not familiar with I can just use what I know to work well and get the best of both worlds

6.4 A practical exploration with MockChat

So far this chapter has been primarily theoretical, it would also be presumptuous to say that this is all you’re going to need to become an expert at ASP.NET MVC. There are many excellent books about ASP.NET MVC whose content is also valid for IronRuby (but the samples will most likely be in C#).

In this exploration we’re not going to follow the interface driven approach but instead will follow this order Model – Controller - View when talking about the MVC implementation. Why? Otherwise we’d have to jump around in our code too much, which would make this exploration pretty hard to follow. So instead we’re going to take a much more sequential approach to writing this application. Let’s start by getting things set up.

6.4.1 The setup

Most MVC applications use a database to persist the state of most of their models. Our application is no different in that regard. We also use a database to persist our state. In this case we’re going to use a sqlite database. Sqlite is a very lightweight database system that holds up well when you don’t have too many users or data to manage (in the regions of 2GB). This makes it almost perfect to use during development as there is no setup involved all you need is the library to talk with the database file.

Once you have a database you probably are going to want to map the database tables to your model objects. Typically one would use an OR/M to handle that mapping for you. There are many good reasons for using an OR/M and not rolling your own mapping layer but that is out of the scope of this book. The OR/M tool we’re going to use is LightSpeed, there is a free version of their tool available which is more than sufficient for what we’re going to need from it in this chapter.

One of the reasons I picked LightSpeed for this chapter was the fact that with little or no coding I get validation, lucence full-text search, it has a good visual designer and it is really simple to use. In addition LightSpeed allows me to either create my models from the database or use their designer to create the models and then have it generate the tables for me in the database. Another benefit is that LightSpeed has built in support for full text searching through Lucene. These features make it a pleasure to work with for me. I should probably also disclose that the guys that wrote this software are personal friends of mine, so there might be some bias involved too. Aside from the benefits I mainly picked LightSpeed because it will present us with some interesting challenges later on. LightSpeed uses obfuscation in its assemblies, it does type overloads and uses ILMerge to concatenate a bunch of assemblies, including native ones, into 1. As far as .NET assemblies go this is about the hairiest situation you can find them in.

These tools cover the models end. On the views end we’re going to make use of jQuery as our javascript library of choice. Again we won’t cover any ground on how to work with jQuery because there are some other books that will do a far better job at that. In addition to the base library from jquery we’ll also make use of 2 jquery plugins to make our life a little easier with regards to making some ajax calls.

The tools you’re going to need for this setup are scattered over the net. Let’s compile a list of where to get, install and configure them.

Getting all the tools you’ll need

The first tool on our list is of course ASP.NET MVC itself. We will use ASP.NET MVC 1.0 in this sample as that is the official release at the time of this writing. You can get the download package in the Microsoft download center (<http://www.microsoft.com/downloads/details.aspx?FamilyID=53289097-73ce-43bf-b6a6-35e00103cb4b&displaylang=en>). Make sure your visual studio is closed and run the setup.

Next up LightSpeed. We’re going to download this tool from the mindscape website (<http://assets.mindscape.co.nz/Downloads/LightSpeedExpress.msi>). When you’ve finished downloading the setup, again make sure all your Visual Studio sessions have ended and run the setup.

And last but not least JQuery and some plugins for jquery. You can download jquery on google code <http://jqueryjs.googlecode.com/files/jquery-1.3.2.min.js> but it is included in the Visual Studio template so you don’t need to download it. We’re going to use the jquery.form and jquery.editinplace plugins. You can download the edit in place plugin from as well from google code <http://jquery-in-place-editor.googlecode.com/files/jquery-in-place-editor-v1.0.1.zip>. Lastly the form plugin isn’t on google code but on github (<http://github.com/malsup/form>). But we’re just interested in the JavaScript file so you can use the direct download link to the library: <http://malsup.com/jquery/form/jquery.form.js?2.33>. We don’t have a location to put these javascript libraries yet so for now they can remain in their download directories until we’ve got an ASP.NET MVC solution, which we’ll create in the next section.

6.4.2 Creating the solution

So far so good we’ve got all the tools needed for this chapter installed on the machine. The first thing we need to do to start programming our chat application is create an ASP.NET MVC enabled solution in Visual Studio. The people that use MonoDevelop can create an empty asp.net mvc application.

The most manual way to do this is by creating a new CLR ASP.NET MVC application with visual studio, and the remove the views, controllers etc from the solution. You then add references to IronRuby.dll, IronRuby.Libraries.dll, IronRuby.Libraries.Yaml.dll and System.Web.Mvc.IronRuby.dll. You can then change the base class of the global.asax to be a RubyMvcApplication.

<%@ Application Inherits="System.Web.Mvc.IronRuby.Core.RubyMvcApplication" Language="C#" %>

If all of this sounds like a lot of hassle to you, the IronRubyMVC project also includes a number of visual studio templates, one of those is a project template that has done all of that for you.

Now that we’ve got a solution we can add the files that are specific to our application. The first thing we’ll do is: add are more references. We need to reference System.Data.SQLite.dll and Mindscape.LightSpeed.dll

SQLite and Mindscape caveats

SQLite comes in 2 flavours a 32-bit version and a 64-bit version. The native 32-bit dll is included in the MindScape.LightSpeed.dll but that won’t work on a 64-bit system. So if you’re running on a 64-bit OS or if you want to use a newer sqlite version than version 1.0.61 (the included version in LightSpeed 2) you’ll need to set up an assembly redirect in the web.config file as shown below.

<runtime>

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

<dependentAssembly>

<assemblyIdentity name="System.Data.SQLite"

publicKeyToken="db937bc2d44ff139"

culture="neutral" />

<bindingRedirect oldVersion="1.0.61.0"

newVersion="1.0.65.0"/>

</dependentAssembly>

</assemblyBinding>

</runtime>

Now we can also add the jquery files that are patiently waiting in the download folder. As layout for this application I’ve used a free template I downloaded from some random template site, I chose this one because it is red and it’s simple. This concludes all the preparations we needed to go through before we can actually start building an application. The first file we’re going to add to the chat application is a routes.rb file.

6.4.3 Routing in a nutshell

Routing is one of the corner stones of any MVC application because it glues a path expression to a controller and its actions. As stated earlier this is a front controller for a complete description on the front controller you can check Martin Fowler’s Pattterns of Enterprise Applications book.

The Front Controller consolidates all request handling by channeling requests through a single handler object. This object can carry out common behavior, which can be modified at runtime with decorators. The handler then dispatches to command objects for behavior particular to a request.

Martin Fowler - <http://martinfowler.com/eaaCatalog/frontController.html>

Now how does this work in ASP.NET MVC? In the CLR model of programming ASP.NET MVC you would define your routes in the Global.asax file when the application starts. The default routes for a C# asp.net mvc application are shown in listing 6.1.

Listing 6.1: Default routes in a C# ASP.NET MVC application

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

"Default", // Route name

"{controller}/{action}/{id}", // URL with parameters

new { controller = "Home", action = "Index", id = "" } // Parameter defaults

);

In the IronRuby version we do a check for a file called routes.rb in your application root directory. When that file is found it will be loaded right after the IronRubyMVC engine has been started. IronRubyMVC exposes a global variable called $routes to your application which you can use to register routes with in an application. If we were to translate the default routes shown in listing 7.1 to Ruby we would end up with listing 6.2.

Listing 6.2: Default routes in an IronRubyMVC application

$routes.ignore\_route("{resource}.axd/{\*pathInfo}")

$routes.map\_route("default", "{controller}/{action}/{id}", {:controller => 'home', :action => 'index', :id => ''})

Lots of applications have enough with just the default routes but you can add more should you want more customization. When you create a route registration you give it a name, a template for a url and the defaults for the variables you use in your url path. Variables are denoted with {} in a template. With our first route set up we can move on to creating the action that is defined as default match /home/index and to do that we’re going to create a controller.

6.4.4 The first controller

Our chat application will have a landing page where we explain what this application is all about etc. Some might even call it the home page. We don’t need any processing yet so the code for this controller and its index action is really simple and shown in Listing 7.3.

Listing 6.3: The Home controller and its index action

class HomeController < Controller

def index

view '', 'layout'

end

end

Yeah that’s right! Surprisingly enough a controller derives from the Controller base class. This index action doesn’t do anything but render a view. At this moment you have to provide 2 parameters if you want to render the view with a layout (which is the same as the MasterPage counterpart in the C# MVC application). In this example we’re using an Embedded RuBy (ERB) template to render our views but you can still use the Webforms view engine if you want. The view is included with the code samples of this book but there is nothing going on there. It might be worth noting that views are by default located in the Views directory and then they are put in a directory with the controller name (Home) in this case. In this directory I created a file called index.html.erb where I put some html.

To run this application you can just hit F5 in visual studio or use a rake task to start the webserver. Because we use ASP.NET MVC, we need the web server to know how to deal with .NET so we can’t just use webrick (which comes with the default IronRuby install). I’ve included a rake task and a small script to start the development web server for asp.net with the samples for this chapter. If you visit the page you should see something like figure 6.3.

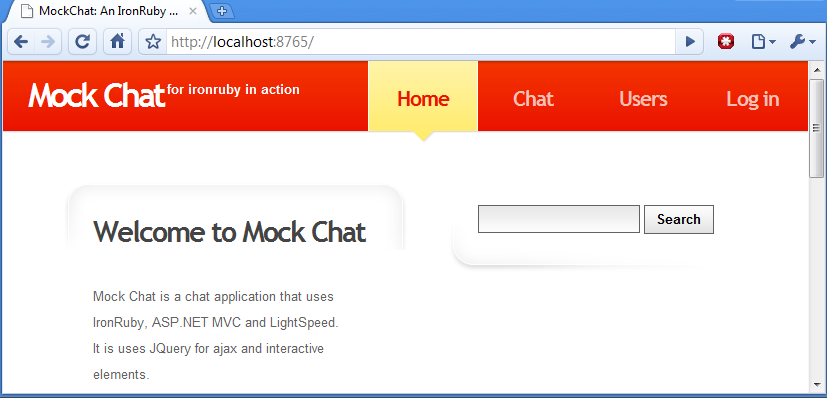


Figure 6.3: The home page of our chat application

The fact that this page works is mostly to confirm that the IronRubyMVC stack is working, because we didn’t really write any code yet. We can now move on to the more useful bits of this application. We want to store stuff in a database and perhaps create some utility models, let’s do those things now.

6.4.4 Configuring our models

People will need to sign in before they can use our application that means we’re going to have to store user credentials. We also want to keep a history of the chats so we also need to store chat messages.

I’ve also included rooms and chat sessions in the model. A chat session is basically a subject wit the messages under that subject and a room is the collection of subjects and active users. Figure 6.4 shows the model when I designed them with the LightSpeed designer.

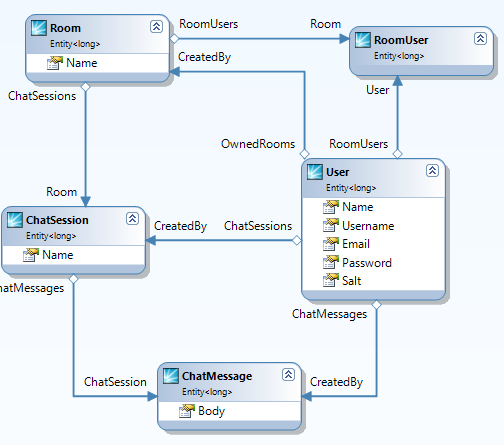


Figure 6.4: The models for the chat application.

We won’t implement all of the stuff that is supported by this model design, I left that open for you to play with. As you can see we have a User model that is linked to ChatMessage, ChatSession and Room. We have a RoomUser join table to keep the online users for a room. A Room also holds a collection of ChatSessions and each ChatSession holds a collection of ChatMessages.

By default the LightSpeed designer doesn’t generate the through associations. The joined rooms for a user and active users for a room are the 2 sides of a through association, so we’re going to define those on our C# model classes. Defining them there so that in case another CLR application needs to have access to the models they can access everything the ruby application can. Listing 6.4 shows the C# code for the User class

Listing 6.4: Adding through associations

public partial class User

{

private readonly ThroughAssociation<RoomUser, Room> \_joinedRooms;

public User()

{

\_joinedRooms = new ThroughAssociation<RoomUser, Room>(\_roomUsers);

}

public IList<Room> JoinedRooms

{

get

{

return Get(\_joinedRooms);

}

}

}

The listing above shows how to define a shortcut property that reaches through the RoomUser class and returns a collection of Room objects. The code for the Room class is about the same so I won’t repeat it, you can find it in the code samples that accompany this chapter. LightSpeed supports a repository approach, which we’ll use to group all the data access methods.

The MockChat repository

The repository pattern is a great way to group the queries for your application or databases together so you can avoid duplication (http://martinfowler.com/eaaCatalog/repository.html). In this case I’ve done this in Ruby, which was the path of most resistance when working with a CLR based OR/M. This book is after all about leveraging the CLR with IronRuby. However, if this were a production application and I needed to use LightSpeed as OR/M for this application because it is a shared library then I’d probably define those queries in the C# MockChatRepository. Linq is a really concise way of defining database queries and LightSpeed has great support for Linq but IronRuby doesn’t at this point. There are more interop problems as we’ll soon see together with possible solutions to these problems.

First we’re going to have to do a little bit of C# groundwork so we can use LightSpeed a little bit easier from Ruby. We’re going to expose the current unit of work as an instance method. Because we’ll open up the Repository class later on in Ruby and the current UnitOfWorkScope property is protected we’re making it public so it can be accessed from a different assembly. Listing 6.5 shows the C# code for the MockChatRepository. We’ll implement its actual behavior later on with Ruby.

Listing 6.5: The C# part of the MockChatRepository

public class MockChatRepository : RepositoryBase<MockChatUnitOfWork>

{

public MockChatRepository(

UnitOfWorkScopeBase<MockChatUnitOfWork> unitOfWorkScope

) : base(unitOfWorkScope)

{

}

public IUnitOfWork Uow()

{

return UnitOfWorkScope.Current;

}

}

This class subclasses the RepositoryBase class that LightSpeed provides. The MockChatUnitOfWork class has been generated by the LightSpeed designer, and it contains IQueryable properties for all the models in our design. This unit of work also groups a bunch of operations together and when told to flush it will execute them in the database.

Why can’t I get to the protected members of a class?

We’ll discuss this in more depth in the chapter about testing. In this case the MockChatRepository class is defined in a CLR assembly. When we define a ruby class that opens up the CLR class this ruby class doesn’t have access to the protected members of that class because it lives in a different assembly than the MockChatRepository class.

So far the CLR doesn’t know about partial properties or partial fields so there is no way for IronRuby to change the implementation of that property. We can access the current UnitOfWorkScope property if we subclass the MockChatRepository class.

All these rules are discussed in more detail in the chapter about testing CLR assemblies with IronRuby.

With the C# side out of the way I can assure you that from now on it’s going to be all ruby until the end of this chapter. The code you’re about to read are all the queries that we’re going to use in our application. This is about as far as I could use the raw .NET API to describe queries but more about that in a minute. Listing 6.6 shows the full ruby implementation of the MockChatRepository.

Listing 6.6: First take on a ruby repository

module MockChat

module Models

LSEntity = Mindscape::LightSpeed::Entity.non\_generic\_type 1

class MockChatRepository

include Lightspeed::Finder 2

def users

uow.method(:find).of(User).call 3

end

def messages

query = Query.new(ChatMessage.to\_clr\_type)

query.order = Order.by(a("CreatedOn")).descending 4

uow.method(:find).overload(Query).call(query)

end

def find\_user\_by\_username(username)

uow.method(:find\_one).of(User).call a("Username") == username 5

end

def find\_user\_by\_id(id)

uow.method(:find\_one).of(User).call id

end

def find\_last\_30\_messages

query = Query.new(ChatMessage.to\_clr\_type)

query.populate\_from\_hash :page => Page.at(0, 30), 6

:order => Order.by(a("CreatedOn")).descending

uow.method(:find).overload(Query).call query

end

def find\_current\_subject

query = Query.new(ChatSession.to\_clr\_type)

query.populate\_from\_hash :order => Order.by(a("CreatedOn")).descending

uow.method(:find).call(query).first

end

def find\_current\_room

uow.method(:find\_one).of(Room).call(1)

end

def save(entity)

uow.add(entity) if entity.entity\_state == EntityState.new 7

uow.save\_changes

end

def remove(entity)

uow.remove(entity)

uow.save\_changes

end

end

end

end

1. Get the non-generic type
2. Add a lighspeed finder module
3. Call an overloaded method
4. Add an order clause
5. Add a simple where clause
6. Add a paging clause
7. Add new entities to the Unit Of Work

In the code just put forward there are a bunch of things that deserve some more explanation. The first one is related to getting types. The type Mindscape.LightSpeed.Entity has both a generic and a non-generic implementation so when we want to interrogate it for its clr\_type we’ll be in for a surprise, Mindscape::LightSpeed::Entity.to\_clr\_type actually raises an Error saying it can’t find the method to\_clr\_type. This can be confusing because if anything LightSpeed is a CLR assembly and its types are CLR types. Because it is a generic overloaded type IronRuby groups these types in a type group (Microsoft::Scripting::Actions::TypeGroup), and because we’re interested in some static methods on the entity type we get the non\_generic\_type and assign it to a shortcut constant (#1) so it’s easier to access later on. You can select different generic overloads by using the of method or get\_type\_for\_arity(number).

The next bit of interest is the inclusion of the Lighspeed::Finder module (#2). This is a module that will make querying with LightSpeed a little bit nicer and rubyesque. We’ll look at this module a little bit later in more depth. For now it’s enough to know that it provides query improvements.

The next one we may have touched on earlier in the book but it has changed a little bit in the last couple of months. In the CLR it’s perfectly valid to overload methods with different parameters allowing you to call the same method, giving you API consistency, and retaining static typing validity. Ruby doesn’t know how to deal with overloaded methods so there is a bit of a work-around for that (#3). IronRuby allows you to pick overloads by first grabbing the method and telling it which overload you want by giving it the types you want to use, which in this case is the one with the non-generic Query parameter.

LightSpeed has a Query object, which is used to define all the parameters for the query at hand. One of those parameters is the Order clause (#4). Another one is the Where clause (#5) and it also allows you to specify a page size (#6). LightSpeed is built around short-lived units of work. That means if you create a new object it starts out detached from the database but as soon as you add it to the context (#7), it will get enlisted as an insert the next time you save changes to the database. Dealing with a unit of work isn’t complete yet we’ll finish it in the section about controllers.

There is one last thing I’d like to say about listing 6.6. That code is ugly as sin, it drastically triggers all kinds of alarm bells and completely uproots the premise of ruby being such a pretty language. Ruby and the Finder module to the rescue! Unfortunately I don’t have enough space to cover how to get to the end point but I do want to highlight some problems that are still open and how I solved them. The one problem I’d like to bring to your attention is the fact that LightSpeed uses implicit operator overloads (&& and ||) and as it so happens you’re not allowed to override those in Ruby. There are a few more operations you’re not allowed to override in Ruby like != and !~. You can learn more about operator precedence in Ruby here: <http://phrogz.net/ProgrammingRuby/language.html>. This is a bit of a problem for many .NET API’s. So to get around that, I basically reimplemented the LightSpeed query builder in IronRuby with overloads for Ruby operators and that is the code contained in the LightSpeed::Finder module. The code is available for you to investigate in the lib/lightspeed\_support.rb file included in the samples for this book. So how does that look after putting in those 200 or so lines in the lightspeed\_support file? Listing 6.7 shows you the answer. I’ve added one method that won’t be used but shows an overloaded operator.

Listing 6.7: The prettier MockChatRepository

module MockChat

module Models

LSEntity = Mindscape::LightSpeed::Entity.to\_a.first

class MockChatRepository

include Lightspeed::Finder

def users

uow.method(:find).of(User).call

end

def messages

find\_many(User, :order => Order.by(a("CreatedOn")).descending)

end

def find\_users\_by\_name

find\_many(User, :where => a("Username") == username & a("Email") != nil) 1

end

def find\_user\_by\_username(username)

find\_one(User, :where => a("Username") == username)

end

def find\_user\_by\_id(id)

find\_one(User, :where => a("Id") == id)

end

def find\_last\_30\_messages

find\_many(ChatMessage, **:page => 0..30** 2  
 :order => Order.by(a("CreatedOn").descending))

end

def find\_current\_subject

find\_many(ChatSession, :order => Order.by(a("CreatedOn").descending))

end

def find\_current\_room

find\_one(Room, :where => a("Id")==1)

end

def save(entity)

uow.add(entity) if entity.entity\_state == EntityState.new

uow.save\_changes

end

def remove(entity)

uow.remove(entity)

uow.save\_changes

end

end

end

end

1. The new way of writing a where clause
2. Use of the ruby Range type for paging

Much better, this new version of the code makes me feel more at home in the Ruby language again. We still need to pass the model class to the find method and the order by syntax can still be improved on but I think it’s clear that with a little effort you can turn a verbose statically typed API into a more wrist-friendly ruby API (#1). We’re also leveraging the Ruby language a lot better, making use of the Range class (#2) to tell the finder where it needs to page. We’re going to use services to glue the repository methods to the models and add behavior.

Servicing the models

Rather than implementing the data access methods on the classes itself we’re going to use services to manipulate the models and basically treat the models as dumb data containers. This has as a benefit that it’s easier to test and it separates structural code better from behavioral code. I’m a proponent of many small classes instead of less but bigger classes; classes tend to grow over time anyway. In the interest of staying largely within the limited space I have to talk about this subject we will only look at the most interesting of the 2 services the UserService, shown in listing 6.7.

Listing 6.7: The UserService implemenation

require 'digest/sha2'

require 'openssl'

require 'mock\_chat\_repository'

class UserService

attr\_reader :repo

def initialize(uow\_scope)

@repo = MockChatRepository.new(uow\_scope) 1

end

def min\_password\_length

6

end

def get\_all

repo.users

end

def get\_one(id)

repo.find\_user\_by\_id(id)

end

def get\_one\_by\_username(username)

repo.find\_user\_by\_username username 2

end

def save(user)

set\_password user, user.password if user.entity\_state == EntityState.new 3

repo.save user

end

def remove(id)

user = repo.get\_user\_by\_id id

repo.remove user unless user.nil?

end

def validate\_user(user\_name, password)

user = get\_one\_by\_username user\_name

!user.nil? && user.password == create\_hash(password, user.salt)

end

def change\_password(user\_name, old\_password, new\_password)

current\_user = get\_one\_by\_username user\_name

if current\_user.password == create\_hash(old\_password, current\_user.salt)

set\_password current\_user, new\_password

save current\_user

else

current\_user.errors.add "Password", "There was a problem changing your 6  
password"

end

end

private

def set\_password(user, passwd)

user.salt = create\_salt

user.password = create\_hash passwd, user.salt

end

def create\_salt

[OpenSSL::Random.random\_bytes(64)].pack("m\*").chomp 4

end

def create\_hash(password, salt)

Digest::SHA512.hexdigest("#{password}:#{salt}") 5

end

end

1. Initialize the repository
2. Call a method on the repository
3. Set a new password
4. Create a random salt
5. Create a hash of the password and the salt
6. Add validation error

The UserService deals with managing User models, and as is the case for every service, it shouldn’t know about the fact that it’s running in the context of the web or ASP.NET MVC for that matter. Respecting this rule makes your model layer reusable for other types of projects for example.

The first thing we do in this class is initialize the repository with the UnitOfWorkScope we get passed in the constructor (#1). We then have a few methods that just forward calls to methods implemented on the repository (#2), showing the flow from service, ultimately by the controller, to the database. The method save is more interesting though. When the model instance represents a new entity it will create a password (#3). To create the new password we need a salt and it’s probably a good idea to make this salt a secure random. The OpenSSL library, a part of the ruby standard library, makes that really easy (#4). We ask the OpenSSL library to generate 64 random bytes for us, which we then pack into a base64-encoded string. The method pack is defined on the Array class (<http://www.ruby-doc.org/core/classes/Array.html#M002222>). And once encoded we’re removing the last newline character with the chomp method.

We won’t store encrypted or plain text passwords but will rather store a hash of the password and use a hashing approach to validate the password in the application. This means that there is no way for us to retrieve the password we can only verify if a provided string is the same. Ruby has a digest library that makes it a breeze to create such a hashed password (#6). In this case we’re creating a 512-bit SHA hash and get its hex-encoded value back, which we’ll store in the database.

The UserService also has a change\_password method and inside this method it needs to validate if the old password is valid before changing to the provided new password. This is a validation that hasn’t been covered by the validations defined with the LightSpeed designer earlier. When this validation fails we have to add the error manually (#6) to the Errors property that exists on every LightSpeed model. We’ll do something useful with those errors later.

I’d like to end this part with a word about the architecture of our models and services layer. I fully realize that this architecture is highly over-engineered for an application this simple and small. I wanted to stay close to the way people develop software in enterprises and use the same terminology. If we were to add a bunch of factories and managers we’d be all set for an “enterprise” chat application. We’ve got a fairly useful back-end now, so the time has come to expose our backend to the web, starting with the controllers.

6.4.5 A controller duet

We’ve made it to the controllers part, so far our application code mainly was about LightSpeed but from now on it will be all about ASP.NET MVC. We’re just going to take a birds-eye view of the ASP.NET MVC framework, if you want to know more about ASP.NET MVC in general I can recommend the ASP.NET MVC in Action book by Jeffrey Palermo. For most of what we’re about to see there exist C# counterparts that you could use instead or reuse from an existing application.

Why don’t we finish up the LightSpeed stuff in the controllers first before moving on to the actual implementation of a controller. As we’ve seen earlier LightSpeed makes you pass those UnitOfWorkScopes around. We’re going to make sure that when a request ends the UnitOfWork will be disposed of and the entities it contains are detached from their context. This stuff is about LightSpeed but applies just as much to the Entity Framework, Linq, NHibernate,… This is often overlooked with all kinds of ugly hacks and workarounds used instead. So we want something that executes on every request right before it gets sent to the browser but after the result has been generated. We can leverage Filters for this kind of work.

Filters

Filters are a common building block of many MVC framework but every framework implements them slightly differently. The ASP.NET MVC framework has 4 types of filters. Table 6.1 has an overview of the available filters and their hooks; they’re also listed in the order that they are executed.

|  |  |  |
| --- | --- | --- |
| Filter name | Hooks | Description |
| AuthorizeFilter | OnAuthorization | Runs before any actions are executed and executes logic that checks if a user is authenticated and takes appropriate actions on the result. |
| ErrorFilter | OnException | Can run whenever an error occurs. It can contain different actions to be performed when an error occurs in the code. |
| ActionFilter | OnActionExecuting | Executes right before an action gets invoked, giving you another chance to decide if you really want to run that action or to prepare some data for the action that is to be executed. |
|  | OnActionExecuted | Executes immediately after an action has been invoked, giving you a chance to manipulate the result of the action before it gets sent further down the pipeline. |
| ResultFilter | OnResultExecuting | Executes right before the view gets rendered, giving you another opportunity to manipulate the view data etc. |
|  | OnResultExecuted | Executes after the view is rendered and is about to be sent through the pipeline to the client. This is the last place you can execute code with filters. |

Table 6.1: Filters in ASP.NET MVC

In a C# application you would apply those by using attributes to decorate methods or classes with such filters. The Ruby language however doesn’t know about attributes at all so we’re going to invoke class methods to register those filters in a controller. A side effect is that we can be more flexible about how filters are registered. Suddenly stuff like you have to define a class for every filter you want to use isn’t obligatory anymore. Let’s look at a ResultFilter (listing 6.8) that is being used to dispose the unit of work and what the different options are to implement that in IronRubyMVC.

Listing 6.8: Defining a class to use as a filter

class LightspeedFilter < ResultFilter

def on\_result\_executing(context)

# put before result filtering code here

end

def on\_result\_executed(context)

context.controller.uow\_scope.dispose

end

end

The code above really only implements one method. The on\_result\_executed method, which gets passed in a context object that gives you access to a.o: the controller instance that this filter is registered to. When a filter is implemented in this way you can also reuse it in C# applications but you would have to figure out a way to instantiate the filter from your C# code. The IronRubyMVC filter implementations subclass CLR based classes, so as far as the CLR is concerned they are actually statically typed classes.

This way of working is still the same as when you would go about it if you were to use a CLR language as your language of choice. Things start to diverge when you look at how these filters are registered in the controller.

I created a base class to derive my controllers from. This base class is called ApplicationController (inspired by Rails) and contains the methods and properties that are common for all controllers in my application (listing 6.9).

Listing 6.9: The application controller base class

require 'lightspeed\_support'

require 'lightspeed\_filter'

require 'user\_service'

class ApplicationController < Controller

filter HandleErrorAttribute 1

filter LightspeedFilter 2

include Lightspeed::ControllerHelpers 3

attr\_accessor :user\_service

def user\_service

@user\_service ||= UserService.new(uow\_scope) 4

end

def current\_user

return nil unless is\_authenticated?

user\_service.get\_one\_by\_username user.identity.name

end

def is\_authenticated?

user.identity.is\_authenticated

end

end

1. Register HandleError filter
2. Register LightSpeed filter
3. Include LightSpeed controller helpers
4. Create a user service instance

In this base class for all controllers in our application we’ve got some methods that provide access to some commonly used data like the user service, current user and whether that user is authenticated or not. In addition to those common methods we’re also registering a filter provided by the ASP.NET MVC framework to handle errors (#1) and you can clearly see that in a CLR world that is an attribute by the class name of the filter. We’re registering this scoped to all actions of the controller. When that filter is done we’re registering the LightSpeedFilter (#2) we created in listing 6.8 in much the same way as we did with the HandleErrrorAttribute. This takes care of the filters we need to register in the application controller. We then include a module Lightspeed::ControllerHelpers (#3). This module gives us access to the UnitOfWorkScope we’ve been using in the models and services implementations as shown in listing 6.10. The user\_service getter method uses the uow\_scope method defined in the filter to initialize the UserService instance.

Listing 6.10: The LightSpeed controller Helpers

unless defined? $context

ph\_path = defined?(System::Web::HttpContext) ? System::Web::HttpContext.current.request.physical\_application\_path : Dir.pwd

#setup context

$context = LightSpeedContext.of(MockChatUnitOfWork).new

$context.connection\_string = "Data Source=#{ph\_path}\\App\_Data\\mockchat\_dev.sqlite3"

$context.data\_provider = DataProvider.SQLite3

$context.identity\_method = IdentityMethod.identity\_column

$context.pluralize\_table\_names = true

$context.logger = TraceLogger.new

end

module Lightspeed

module ControllerHelpers

def uow\_scope

@uow\_scope ||= PerRequestUnitOfWorkScope.of(MockChatUnitOfWork).new($context)

end

def collect\_errors(entity, prefix="")

entity.errors.each do |error|

prop = prefix.nil? || prefix.empty? ? error.property\_name.underscore : "#{prefix}.#{error.property\_name.underscore}"

model\_state.add\_model\_error prop, error.error\_message

end

end

end

end

When we require the lightspeed\_support.rb file it will execute some code that sets up the LightSpeedContext with a connection string and some other configuration and stores that in the global $context variable. We then create a module called ControllerHelpers which contains a lazily initialized PerRequestUnitOfWorkScope with that context. We also define a collect\_errors method which we’ll use a little bit later on to convert the Errors collection on a model to a ModelState error so ASP.NET MVC knows how to deal with them too. This is important if we want to take advantage of the validation helpers that are included in the ASP.NET MVC framework later on in our views.

Modules/mixins or inheritance?

If you’re asking yourself why did he not use a module or mixin instead of subclassing? Quite frankly I don’t know and didn’t think it would matter much. I tend to try to answer the question this class ‘is a’ … or this class ‘is also a’ … As soon as ‘is also a’ starts to sound like a better fit, I’ll go for modules. However I guess that is a bit of a judgment call and depends on personal programming style. And it isn’t the only criterion I use to determine which would be most appropriate in the case I’m working on at that moment.

Modules/Mixins are great to group methods together and reuse those across classes but conceptually it’s a little different from subclassing although both have similar outcomes and Modules are often talked about as if they enable multiple inheritance. You can see that that is a little bit confusing as soon as you start asking the implementing class if it actually is a certain module it will return false. You can ask if it includes a module though. So it’s a subtle difference but a difference nonetheless.

Before we can look at the UsersController there is another, slightly more specialized, base controller that I created which ensures that all requests made to actions in a deriving controller are authorized. It’s a specialization of the ApplicationController and its code is shown in listing 6.11.

Listing 6.11: The AuthenticatedControllerBase

class AuthenticatedControllerBase < ApplicationController

filter AuthorizeAttribute

before\_action :controller do |context|

raise System::InvalidOperationException("Windows authentication is not supported.") if context.http\_context.user.identity.is\_a?(System::Security::Principal)

end

end

In this controller we register an ASP.NET MVC intrinsic filter called AuthorizeAttribute, which makes sure that any requests made to a controller are authenticated because it’s scoped to a complete controller again. If you wanted to scope a filter to a particular method you can pass it the name of an action as first argument:

filter :change\_password, AuthorizeAttribute

This code will scope the filter to the change\_password action but all other actions are allowed to be unauthenticated. The second filter declaration however is completely new and presents us with an interesting syntax to define filters in controllers by using a block instead of registering a class. IronRubyMVC defines 8 of these shortcut methods (shown in table 8.2). Instead of passing a block you can also pass it a method name to instead of the block.

|  |  |
| --- | --- |
| Method name: | Description |
| before\_action | Executes right before an action is invoked |
| after\_action | Executes right after an action is invoked |
| around\_action | Executes right before AND after an action is invoked |
| authorized\_action | Executes like an AuthorizeFilter |
| exception\_action | Executes when an error is encountered |
| before\_result | Executes right before the view is rendered |
| after\_result | Executes right after the view is rendered |
| around\_result | Executes right before AND after a view is rendered |

At this point you really do know all the filters that are available and how you use them. You also know enough of the base now to read the code for the controller so it all makes sense.

Implementing a controller

In this part of the chapter we’ll implement the UsersController. It brings together everything we’ve seen earlier to provide a CRUD (Create, Read, Update, Delete) interface for managing users in our application. This controller allows me to demonstrate validation, passing data to the view and taking input. The code for this controller is shown in listing 6.12.

Listing 6.12: The UsersController implementation

class UsersController < AuthenticatedControllerBase

def index

@users = user\_service.get\_all

view '', 'layout' 1

end

def new

@user = User.new

view 'new', 'layout'

end

def create

@user = User.new

save\_user :new

end

def edit

@params\_id = params[:id] 2

@user = user\_service.get\_one params[:id].to\_i

view 'edit', 'layout'

end

def update

@user = user\_service.get\_one params[:id].to\_i

save\_user :edit

end

def destroy

user\_service.remove params[:id].to\_i

redirect\_to\_action 'index'

end

private

def save\_user(act)

self.method(:update\_model).of(User).call(@user, "user") 3

if @user.is\_valid 4

user\_service.save @user

redirect\_to\_action('index', 'users')

else

collect\_errors @user 5

view act.to\_s, 'layout'

end

end

end

1. Render default view
2. Take simple input
3. Bind complex model to input
4. Validate model
5. Relay validation errors

Most of the code is pretty straightforward in that it uses the user\_service defined on the application controller from listing 6.9. In the index action we use the user\_service to retrieve all users and render the default view (#1) for that action. When you pass an empty string as first parameter to the view method it will look for a view with the same name as the action.

The next item on our list is taking simple input, by simple I mean primitives like string, integer etc. In the edit action we need to get the user by id. Our models use integers as their primary keys so that’s one of those primitives. And because the default route defines the id parameter as a participant in the route we can use a prettier URL like /users/edit/4 instead of /users/edit?id=4. The point is that from the perspective of the web server there is a query string parameter called id. And we get the value from that id parameter (#2) in the edit action.

When we post a form with the fields that define a user then it would be great if we didn’t have to set all the properties on the entity manually but we could bind them in some way to that input. ASP.NET MVC provides a method to do that. And we use that to update an instance of a user with the values we retrieve from the form collection in the post variables (#3). Also in the save method we check if the model is valid (#4) and if it isn’t we need to convert the LightSpeed validation errors to ModelState errors (#5). If you’re wondering what the ModelState is. It’s a property on the ViewDataDictionary that the view and the html helpers use to share validation information. You can think of it as the context for validations.

This ends the discussion about the UsersController but there is one last thing I want to share with you because it’s slightly different from the CLR ASP.NET MVC implementation. In a CLR project you can use IsAjaxRequest, but because it’s used so often we’ve provided a more ruby like method of accessing that. Also in the CLR version you can use overloads and attributes to differentiate between GET, POST, HEAD requests and different numbers of parameters. In the IronRuby version you can’t overload methods so there are a few methods available on a controller to help you achieve the same thing. We extended the HttpRequestBase class to hold flags for those http methods. Table 6.3 shows the methods that are available with a short description.

|  |  |  |
| --- | --- | --- |
| HTTP method | Controller method | Description |
| HEAD | request.head? | Returns true when this is a HEAD request |
| GET | request.get? | Returns true when this is a GET request |
| POST | request.post? | Returns true when this is a POST request |
| PUT | request.put? | Returns true when this is a PUT request |
| DELETE | request.delete? | Returns true when this is a DELETE request |
| Header['X-Requested-With’] | request.ajax? | Returns true when this is an AJAX request |

Table 6.3: controller methods

Instead of providing overloads you can use a switch or if statement in a method to achieve the same result. The easier and prettier solution is probably to just use different actions.

Linking the controller with the view

In ASP.NET MVC a controller action can return a string or an ActionResult. You can create your own action results too. By default ASP.NET MVC 1.0 supports about 9 different action results, table 6.4 shows them with their controller method and a short description.

|  |  |  |
| --- | --- | --- |
| Action Result | Controller method | Description |
| ContentResult | content | Allows you to send custom arbitrary content types and data |
| EmptyResult |  | Return value when an action has to return nil |
| FileResult | file | Allows you to return files as binary data written to the response stream |
| JavasScriptResult | java\_script | Allows you to return bits of javascript to the client to be executed |
| JsonResult | json | Allows you to return objects represented as JSON. |
| PartialViewResult | partial\_view | If you don’t want to specify a layout (master page) for your view and just want to return a snippet of HTML. |
| RedirectResult | redirect | Useful when you want to redirect to another url |
| RedirectToRouteResult | redirect\_to\_action,  redirect\_to\_route | Redirects also but to a different action in the same controller or to an action on a different controller. |
| ViewResult | view | Allows you to render views (html) with a layout/masterpage |

Table 6.4: The action result methods.

Using these controller methods help the controller decide how to represent the data and it clearly shows you that the controller is the one deciding this. By default the ASP.NET MVC framework will look in the Views directory to load the views but you can override this in the web.config file. You can also create your own ActionResult classes to provide return a format that is used only by your company or something. When you use the methods view or partial\_view in the controller you indicate that you want to return html. When you return html you can use templates to generate that html filled with the values from the data you’ve made available to the view. This does leave one question open: How does the view get the data you prepare in the controller?

If you’ve used ASP.NET MVC before you’ll know that both the controller and the view have a ViewData property where you can store objects you want your view to be able to reach. The accepted better practice in the CLR version has you creating view models that provide statically typed access to that view data (it’s statically typed, you will more often hear strong typed, but we know better after reading chapter 1).

The bad news is that in ruby there is no static typing, the good news is that ruby is dynamically typed (keeping it still strong). That means that for the IronRuby version you can bypass the dictionary and use instance variables to expose data to the view. In the IronRuby MVC version all the instance variables you declare in the controller are made available in the view as methods on the view data dictionary (it will actually use the ViewDataDictionary to transport the data to the view though). You can still use a view model approach if you want to and you can also still use the view data dictionary but you just have one more option to achieve the same goal.

This ends our discussion on controllers in ASP.NET MVC and we can now start the last leg of our exploration, next we’ll deal with the views and its helpers.

6.4.7. The View and its friends

We’ve made it to the last part of this chapter where we’ll look at creating view templates and using some helper methods to make this task easier for you. We’ll continue this exploration with the views used by the UsersController from listing 6.12. While we’ll only use views defined in ruby there is no reason why you can’t use views that have been defined with the webforms view engine, sparkle, haml or... You can just mix and match those as you see fit.

Let’s start by explaining how you can use a layout aka. master page in asp.net mvc. By default the framework looks for shared views in the shared subdirectory of the views directory. So that may be an appropriate place to put the layout. I created a file called layout.html.erb in that shared folder and added a bunch of html to the file. There is only one line of relevance in there:

<!-- start content -->

<div id="content">

**<% yield %>**

</div>

<!-- end content -->

The yield keyword indicates where the template engine should insert the html of the actual view in the layout. The content of the view we’re about to discuss will be added between the div#content tags. The first method defined in the UsersController is the index method and in this method there is the line @users = user\_service.get\_all. This method will return a list of users and listing 6.13 shows the code of the view to render out that list of users.

Listing 6.13: The index view for the UsersController

<div class="post">

<div class="title">

<h2>Users: list</h2>

</div>

<div class="entry">

<p>Registered users for the MockChat application</p>

<div class="main">

**<%** if **view\_data.users.empty?** %> 1

<div>No users registered yet.</div>

<% end %>

<div>

**<%= html.action\_link('Add user', 'new', 'users')** %> 2

<br/>

</div>

<% unless view\_data.users.empty? %>

<table>

<thead>

<tr>

<th>Name</th>

<th>Username</th>

<th>&nbsp;</th>

</tr>

</thead>

<tbody>

<% **view\_data.users.each do |user|** %> 3

<tr>

<td><%= **html.action\_link(user.name,   
 { :action => :edit, :id => user.id})** %></td> 4

<td><%= user.username %></td>

<td><%= html.action\_link("destroy",   
 ={ :action => :destroy, :id => user.id}) %></td>

</tr>

<% end %>

</tbody>

</table>

<% end %>

</div>

</div>

<p class="links">&nbsp; </p>

</div>

1. Check for an empty result
2. Use a helper
3. Iterate over the user list
4. Access a property on a user

Some ruby code got mixed in the html above there and the first point of note is that we’re checking if the users collection on the view\_data is empty (#1). If it is empty we’ll tell the user that there are no users registered yet. The next item we’re going to tackle is: make sure that there is a link to create a new user. This link invokes the new action on the users controller (#2). We then get to the part where we’re actually rendering the list of users. This uses a html table and when it’s time to create the rows we wrap a single row in between an .each iteration (#3). We also provide a link to the edit action, which we pass the id of the user in that row (#4). For completeness there is also a destroy link that does the same as the edit link but to the destroy action.

There is still 1 thing we glanced over in the previous paragraph; there seem to be 2 different notations for executing ruby code. The first one is **<% %>** and the other one is **<%= %>**, what’s that about? The former indicates you want to execute some code but it won’t return a result or you’re not interested in the result. The latter is roughly the equivalent of Response.Write and can only be used with methods that return a string value.

Suppose a user clicks the edit link then the edit action on the controller would be invoked and that action sets the user instance variable, which we’ll use in the code presented in listing 6.14. The edit view contains a few methods that we haven’t seen yet.

Listing 6.14: The edit view

<div class="post">

<div class="title">

<h2>Edit <%= view\_data.user.name %></h2>

</div>

<div class="entry">

<%= **html.validation\_summary**("Account update was unsuccessful. Please correct the errors and try again.") %> 1

<% **html.begin\_form "update", "users", :id => view\_data.user.id** %> 2

<div class="user">

<fieldset>

<legend>Edit <%= view\_data.user.name %></legend>

**<% html.render\_partial 'user\_form\_fields' %>** 3

<p>

<button type="submit">Save</button> 4

&nbsp;</p>

</fieldset>

</div>

<% html.end\_form %>

</div>

<p class="links">&nbsp; </p>

</div>

1. Display a validation summary
2. Start a form
3. Render a partial view
4. Submit the form to the server

Earlier in this chapter we mentioned the ModelState a few times as the way to send validation messages back to the view. In the view we can display a summary of all the validation errors with the validation\_summary html helper method (#1). Next we begin a form (#2) that has its action set to update on the users controller. The fields for a user are being used in a number of places, there is a registration page and a create user page so instead of repeating this code it got separated in to a partial. The edit form renders this partial view too (#3). To post the form to the server we use a submit button (#4). The html contained in the user\_form\_fields partial is shown in listing 6.15.

Listing 6.15: The user\_form\_fields partial

<label for="user.name">Name:</label><br />

<%= html.text\_box("user.name", view\_data.user.name, :tabindex => 1) %>

<%= html.validation\_message("user.name") %><br />

<label for="user.username">Username:</label><br />

<%= html.text\_box("user.username", view\_data.user.username, :tabindex => 2) %>

<%= html.validation\_message("user.username") %><br />

<label for="user.email">Email:</label><br />

<%= html.text\_box("user.email", view\_data.user.email, :tabindex => 3) %>

<%= html.validation\_message("user.email") %><br />

This partial makes heavy use of html helpers more specifically the form helpers that render out textboxes and bind them to data that’s available to the view. Then similarly to the validation\_summary method there is also a validation\_message, which looks up error messages that are stored in the ModelState dictionary with the provided key.

At this point you’re able to use helper methods but helpers become a lot more useful when you can create your own to share across projects or specific to the current project.

Creating helpers

I didn’t get the chance to show you any of the custom helpers I built being used in the code. But I did write a few. Helpers provide yet another way for you to manipulate HTML and reuse logic. When I’m writing MVC applications my view needs to be as dumb as possible, to achieve this I made up a rule for me. It goes a little like this: Can you express it with a very simple if..else..end statement or is it a simple loop? Then it belongs in the template, but if the logic becomes a little bit more complex it’s probably a better idea to put it in a helper method.

Helpers can do anything from render a simple link to render an entire grid with preset CSS classes and the works. Helper methods are the closest thing you get to widgets/user controls in ASP.NET MVC. The way helper methods need to be built can use some improvement but at this time it involves a little bit of ceremony to make it work. The reason for this is that C# got extension methods added to the language which appear to be instance methods. These methods are implemented as static methods on a static class which means they’re about as final as they can be. There is no way to override a static method so that the CLR will accept this override. But we can reach the same goal if we just open up a HtmlHelper class and start adding our helpers there, as shown in listing 6.16.

Listing 6.16: The html helpers for the chat application

module System::Web::Mvc::IronRuby::Helpers

class RubyHtmlHelper

def menu\_link(text, url, key, route\_value\_key=:controller)

if key.to\_s.underscore == view\_context.route\_data.values[route\_value\_key].underscore 1

"<li class='current\_page\_item'><a href='#{url}'>#{text}</a></li>"

else

"<li><a href='#{url}'>#{text}</a></li>"

end

end

def login\_menu\_link(login\_text="Log in", logout\_text="Log out", login\_action=:log\_on, logout\_action=:log\_off, controller=:account)

text, act = view\_context.controller.is\_authenticated? ? [logout\_text, logout\_action] : [login\_text, login\_action] 2

menu\_link text, "/#{controller}/#{act}", controller

end

def format\_chat\_body(chat\_body)

encode(chat\_body).gsub("\n", "<br />")

end

end

end

1. Check for current page
2. Use multi-assignment

The helpers we’ll discuss are used in the layout file and create menu links tracking the current page the user is on. To do that we check if the url and key that has been provided is the one currently displaying (#1). If it’s the current page we add a css class to the list item otherwise we just render a regular link. There is one more specialized link, the login/logout link. This needs to display a different text and execute a different action when clicked. In the login\_menu\_link method we work out if the current request is authenticated by checking the context property on the html helper class. If this is an authenticated request, we return different values for the text and url (#2). You can use these helper methods just like you use the ones that come with the asp.net MVC framework. To use the format\_chat\_body helper you’d write: <%= html.format\_chat\_body(message.body ) %>.

This concludes our discussion on ASP.NET MVC; we’ll summarize what we’ve talked about in this chapter before moving on to the next top.

6.5 Conclusion

This chapter took us through how IronRuby looks when building web application using the ASP.NET MVC framework. We took some time to look at which components make up the MVC part and what their jobs are. We also learnt how an ASP.NET request gets processed when it arrives at the web server. We discussed how it first passes through a series of http modules and then gets executed by a handler. The result of that execution passes through a bunch of modules again before being sent back to the client. Because we know how this executes we can also see how the ASP.NET MVC framework leverages these things to do its magic.

We then saw how this handler uses routing to make sure the request arrive at the right controllers and actions. Once a request arrives it uses the models to do the actual work but the controller is the one who supervises the process. The controller supervises taking input, calling the appropriate models and the type of output. The last bit in our discussion dealt with views and their helpers, more specifically on how to create helper methods and how to leverage view data and validation in a view template. Using Asp.NET MVC with IronRuby gives you the advantage that your existing C# codebases aren’t lost but you can integrate them seamlessly with controllers defined in Ruby.

With Rails, Sinatra, Merb,… and ASP.NET MVC the ruby developer can’t say he doesn’t have choices for creating web applications. Our next chapter deals with Rails and how you can use that from IronRuby to deploy to an IIS server.