4

Controller

This chapter covers

* Understanding the Controller Anatomy
* Storyboarding the Application.
* Mapping the Presentation Model
* Using Input from the Browser
* Passing View Metadata

The focus of the Model-View-Controller pattern is the controller. With this pattern, every request is handled by a controller and rendered by a view. Without the controller, presentation and business logic would move to the view, as we have seen with Web Forms. With the ASP.NET MVC Framework, every request routes to a controller, which is simply a class that implements the IController interface (see chapter 9). Microsoft provides the base class System.Web.Mvc.Controller to make creating a controller easy. The controller base class you choose is not crucial because most request processing goes into executing the ActionResult, which is the type that each action returns.‑­

An action is a method that handles a particular request. This method can take no parameters or many, but by the time the action method finishes executing, there ought to be one or many objects ready to be sent to the view, and the name of the view should be selected if the view does not follow the convention of having the same name as the action. Beyond that, the developer is in complete control regarding how to implement a controller and its actions. This chapter will explore controllers that use many actions and inherit from the System.Web.Mvc.Controller base class. The meat of the controller is the action.

4.1 The Anatomy of a Controller

A controller is simply a class that inherits from System.Web.Mvc.Controller that contains one or more methods that act as Actions. An action method is used to serve a single web requests that normally returns an ActionResult and can take zero or many arguments. Parameters are resolved into the action method by the Model Binders, by using this automatic binding to complex types Action methods can focus on the concerns of controlling the application logic rather than spending time: translating input from the browser into Input Models or mapping Domain models into presentation models.

An well written action should have a clear purpose and a single responsibility. That responsibility is to accept input from the browser and coordinate the flow of the application. Along the way, the action should rely on application services to perform tasks such executing business logic, performing data access, or file I/O.

Listing 4.1 The SimpleController decides on ViewData and renders a view

using System.Web.Mvc;

namespace MvcInAction.Controllers

{

public class SimpleController : Controller

{

public ActionResult Hello()

{

ViewData.Add("greeting", "Hello Readers!");

return View();

}

}

}

4.2 Storyboarding the Application

Action methods exist to perform presentation coordination for a screen/page. This coordination is the glue that puts ties together the Storyboard of the Application. Imagine drawing the flow of application screens on a whiteboard. Each place that a user has the ability to input information through a form or click a button, there are at least two possible outcomes. These are a successfully handled request or an alternate path that requires additional input or a decision from the user. This alternate path could included dealing with data type validation errors, business rule validation errors, or exception handling.

Focusing an Action to concentrate on wiring together the storyboard of the application has some good side effects. Actions tend to become smaller and focused. By moving business logic out of the Action and into supporting services, the actions are less complex and easier to test. A lean action should result in two possible outcomes: Happy Path ( a successfully processed request) or a alternate path. If an action starts branching to handle multiple alternate paths this is sign that the Action method is handling too much and some design should be put into the storyboard of the application.

Figure 4.2 Storyboard of a User Interaction

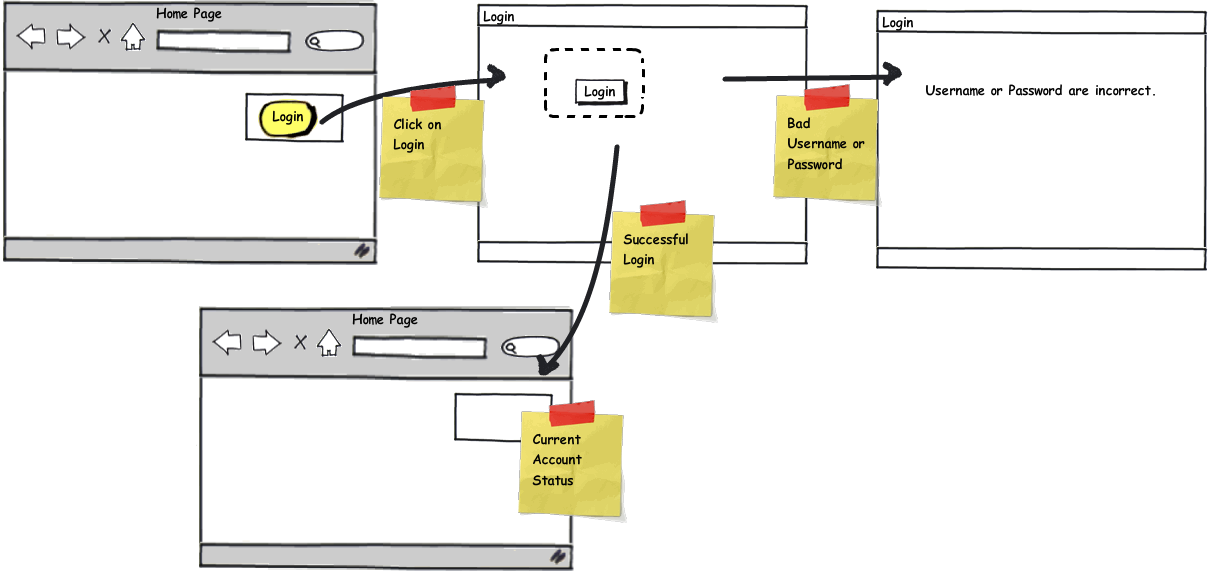


Figure 4.2 shows a sample Storyboard of how a user would login to a web application and see some customized content after a successful login. The Action that handles the Login Form post would decide to redirect the user to the homepage or re-render the Login form with a message that explains the user needs to enter a correct username and password combination. While this seems like a very obvious path that needs to be developed, it is very easy to overlook the alternate paths when you do not storyboard them. By using a storyboard the design and requirements of your actions just fall off of the storyboard. This technique helps developers and designers communicate how the screens would work before writing a single line of code.

4.3 Transforming a model to a view model

A common role of an action is to simple do the work necessary to mold a Domain model into a presentation model for a view, JSON, or other output type. This type of action handles a GET request to the web server and in its simplest form returns Html to the browser.

Listing 4.3 - An Action that prepares a Presentation Model for a view

public ActionResult Index()

{

IEnumerable<User> users = UserRepository.GetAll();

UserDisplay[] viewModel = users.Select(

user => new UserDisplay {

Username = user.Username

}).ToArray();

return View(viewModel);

}

In the code example listing 4.3, this action retrieves a collection of User domain model objects and transforms the objects into a Presentation Model of type UserDisplay[]. The action relies on a UserRepository class to handle all the communication to the database and turning the native database objects into the User collection. Next, the action uses some System.Linq syntax to minimize the noise in performing this type of transformation. The last line of the action sends the Presentation Model to a View helper method and returns a ViewResult to the MVC framework. Since a view name was not specified the framework uses a convention and looks for a view that matches the Action name. In this case it would look for a view called Index.

4.3 Accepting Input

Accepting input from a web browser into an action can happen through a Accepting input from a web browser into an action. The controller uses the Model Binder feature resolve values from form posts and query string and turn them into .Net object that match the named of parameters of an Action. The internals of how this works is covered in a later chapter, for now it is important to understand that a convention is used to match up form values by their name to the parameter name of an action.

Listing 4. A value object bound to an Action from a query string

[HttpGet]

public ActionResult Edit(int Id)

{

User users = UserRepository.GetById(Id);

....

}

The code in listing 4.4 shows a value object being bound from the query string of a url. The url to send an id with the value 4 would be http://localhost/User/Edit?id=4. The Action can use the value to perform it's work without having to deal with the concerns of pulling values out of the HttpContext. When the code in an action accesses the Request object to translate data into an object, that is a sign that the action is doing work that it should not be. Actions need to be focused on the storyboard instead of translating input data.

Listing 4.5 A complex object bound to an Action from a Form Post

[HttpPost]

public ActionResult Edit(UserInput input)

{

if (ModelState.IsValid)

{

... //Update the User object.

return RedirectToAction("index");

}

return View(input);

}

The code listing 4.5 demonstrates how a

4.3.1 Handling the Successful processing of Action Input

Explain ModelState.IsValid as one branch in flow  
Show code from Edit Action

Explain Post Redirect Get

Diagram to explain PRG Pattern

TempData as a way to pass view data/success message

4.3.2 Handling the Failure processing of the Action Input

Show code from the failure branch of the Edit Action

Explain ModelState as a way to add additional error messages.

4.4 Summary