2

Model

A model is a representation of something meaningful. Not necessarily something physical, but something real: a concept or a business or an API that's difficult to work with.

When we write object oriented software, we create classes that make up this representation. We can create our representation so that when we use it we are working in our natural human language, like English or Spanish or business jargon, instead of in mere programming language constructs like booleans, meaningless strings and integers.

When working with a UI framework like ASP.NET MVC the meaningful thing we have, the complex problem we manage, is the UI. It's the data in a window, a form submission from a user, the options in a select list. The model represents the screen.

2.1 The M in MVC

Consider a screen that shows a table to the user:

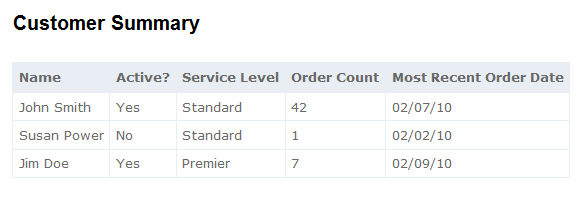


Figure 2.1 A table in our user interface

This table in Figure 2.1 is the product of our software development. It deserves to exist as a first-class object in our system. This will allow us to intentionally create it and to maintain it after its initial development. A first-class object representing this table, or rather, representing each row, will also allow our view an unencumbered mechanism to display the table itself.

Listing 2.1 CustomerSummary.cs

public class CustomerSummary

{

public int Number { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public bool Active { get; set; }

public string ServiceLevel { get; set; }

public string OrderCount { get; set;}

public string MostRecentOrderDate { get; set; }

}

It's simple on purpose. Our model consists mostly of strings. That's what we're representing, after all: text on a page. The logic that displays the data in this object will be straightforward; the view will only output it. The presentation model is designed to minimize decision making in the view.

The model for the entire table is of type IEnumerable<CustomerSummary>. With a simple model like that, the view only has to iterate through it, writing a row for each CustomerSummary.

2.2 Delivering the presentation model

Somewhere in our application we'll build this presentation model. It may be hydrated with the results of a simple database query, like a flat report. Or it may be calculated and projected from another set of interesting data. It's common to have a class whose sole responsibility is to formulate the presentation model. Doing the work of building a presentation model in application code is better than doing that work in the view. The view is convoluted enough as it is, and focused on HTML and style. A separate class that creates the presentation model can be easily tested, programmed and maintained.

It's also best to not perform this work in the controller. The controller is busy deciding which view the render and coordinating these efforts. From the controller's perspective, there's nothing to it. A simplistic look at how a controller might send the presentation model to the view is offered in Listing 2.2.

Listing 2.2: A controller action preparing the presenation model

public ViewResult Index()

{

IEnumerable<CustomerSummary> summaries =

\_customerSummaries.GetAll();

return View(summaries);

}

Once the model, those summaries in Listing 2.2, are ready, the controller passes them into the View() method, transfering them to the view.

2.3 ViewData.Model

The controller and view share an object of type ViewDataDictionary named ViewData. ViewData is a regular dictionary, with string keys and object values, but it also features a Model property. Conveniently, ViewData.Model is where we put our model. And the Model property is strongly typed, so our view knows exactly what to expect and developers can take advantage of IDE features like IntelliSense and support for renaming variables.

Listing 2.3 shows how a view can describe its model type in the Page directive.

Listing 2.3 Defining the model in the Page directive

<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"

Inherits="System.Web.Mvc.ViewPage<IEnumerable<CustomerSummary>>" %>

The Inherits tag in Listing 2.3 specifies that the view's model (the ViewData.Model property) is of type IEnumerable<CustomerSummary>. Because we designed our model to work with our screen, it's easy to mark up with HTML.

Listing 2.4 Using the model in the view

<table>

<tr>

<th>Name</th>

<th>Active?</th>

<th>Service Level</th>

<th>Order Count</th>

<th>Most Recent Order Date</th>

</tr>

<% foreach (var summary in Model) { %> #1

<tr>

<td><%= summary.FirstName %> <%= summary.LastName %></td> |#2

<td><%= summary.Active ? "Yes" : "No" %></td> |#2

<td><%= summary.ServiceLevel %></td> |#2

<td><%= summary.OrderCount %></td> |#2

<td><%= summary.MostRecentOrderDate %></td> |#2

</tr>

<% } %>

</table>

#1 Model is IEnumeable<CustomerSummary>

#2 Working with the model

This markup renders our table. Instead of relying on "magic string" keys and complex logic, we're free to work directly with a strong, clear model. By constructing the model elsewhere and designing it to represent the screen, the developer's job here is easy.

There are other types of models. Domain models typically represent a part of a business, and we cover the domain model in chapter 8. A model that represents the user interface is sometimes called a presentation model or a view model. Many simple applications will share a domain model and a presentation model - the UI and core of the application will use the same classes. But these are only the most trivial of applications, and even then it's advisable to segregate these duties.

Somewhere in our application we'll build the presentation model. It may come directly from a database, like a plain, tabular report. Each column would be a property and each row would be an item in a collection. It may be projected from a domain model. It's common to have a class whose sole responsibility is to translate from a domain model to a presentation model.

We'll discuss later how a controller takes this presentation model and sends it to a view. And we'll see how a view is left with the relatively easy task of painting the model on the user's monitor.

One key tip that we learned the hard way: don't share models between views. A presentation model exists for just one. Sharing them makes changing one screen or the other difficult, because they both depend on the same thing. We want to allow our models the flexibility to fit each screen and not be tied to another, common structure.

2.1 ViewData.Model

In ASP.NET MVC the model goes in the Model property of an object named ViewData. The controller and the view share this object, and the controller uses it to give the view its data.

It's of type ViewDataDictionary, which is a regular dictionary but it also has a special property called Model.

ViewModel

- ViewDataDictionary

- code snippet of magic string stuff from default template

- show controller

- why this is bad and error prone

- the Model property

- why this is better

- show page directive

- show <%= summary.FirstName %>

- show controller

Display model example

2.2 Presenting data

Input model example

- the input model represents user input

- Helpers (quick)

- Modelbinding (quick)

Complex scenarios