**Understanding the MVC Pattern**

In high-level terms, the MVC pattern means that an MVC application will be split into at least three pieces:

* *Models*, which contain or represent the data that users work with. These can be simple *view models*, which just represent data being transferred between views and controllers; or they can be *domain models*, which contain the data in a business domain as well as the operations, transformations, and rules for manipulating that data.
* *Views,* which are used to render some part of the model as a user interface.
* *Controllers*, which process incoming requests, perform operations on the model, and select views to render to the user.

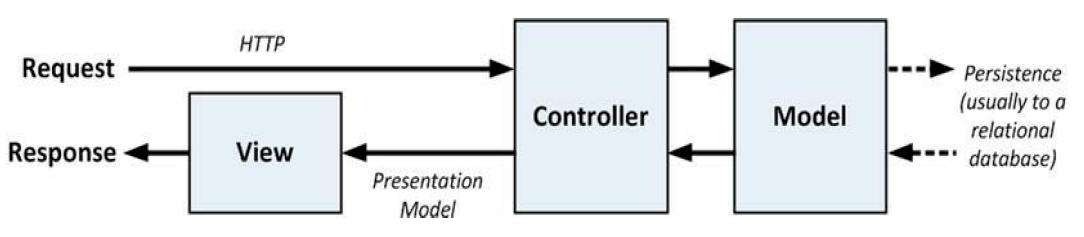
***Models*** are the definition of the universe your application works in. In our human resource (HR) content management system (CMS) application, for example, the model represents everything that the application supports, such as branches, the job applicants, and job descriptions—as well as the job fairs for individual branches and maintaining the current jobs available for each branch. The model is also responsible for preserving the overall state and consistency of the data—for example, making sure that all applicants are processed and sent to appropriate hiring manager, and that when a job position is filled the job posting is unpublished or a position is available a job posting is created. Models are also defined by what they are *not* responsible for: models don’t deal with rendering UIs or processing requests—those are the responsibilities of *views* and *controllers.*

***Views***contain the logic required to display elements of the model to the user—and nothing more. They have no direct awareness of the model and do not directly communicate with the model in any way.

***Controllers***are the bridge between views and the model—requests come in from the client and are serviced by the controller, which selects an appropriate view to show the user and, if required, an appropriate operation to perform on the model.

Each piece of the MVC architecture is well-defined and self-contained—this is referred to as the *separation of concerns*. The logic that manipulates the data in the model is contained *only* in the model; the logic that displays data is *only* in the view, and the code that handles user requests and input is contained *only* in the controller. With a clear division between each of the pieces, your application will be easier to maintain and extend over its lifetime, no matter how large it becomes.

**The ASP.NET Implementation of MVC**

In MVC, controllers are C# classes, usually derived from the System.Web.Mvc.Controller class. Each public method in a class derived from Controller is an *action method*, which is associated with a configurable URL through the ASP.NET routing system. When a request is sent to the URL associated with an action method, the statements in the controller class are executed to perform some operation on the domain model and then select a view to display to the client. Figure 3-1 shows the interactions between the controller, model, and view.

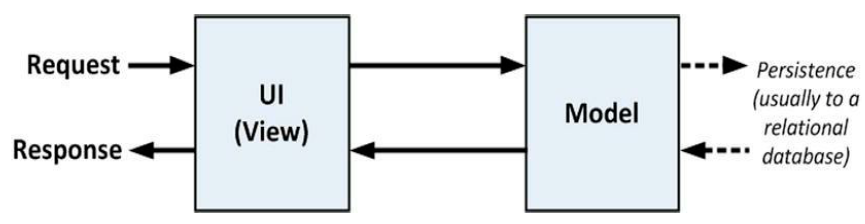
***Figure 3-1.*** *The interactions in an MVC application*

The ASP.NET MVC Framework uses a *view engine*, which is the component responsible for processing a view to generate a response for the browser. Earlier versions of MVC used the standard ASP.NET view engine, which processed ASPX pages using a streamlined version of the Web Forms markup syntax. MVC 3 introduced the Razor view engine, which was refined in MVC 4 (and unchanged in MVC5) and that uses a different syntax entirely.

* **Tip** Visual Studio provides IntelliSense support for Razor, making it a simple matter to inject and respond to view data supplied by the controller. ASP.NET MVC doesn’t apply any constraints on the implementation of your domain model. You can create a model using regular C# objects and implement persistence using any of the databases, object-relational mapping frameworks, or other data tools supported by .NET.

**Understanding the Model-View Architecture**

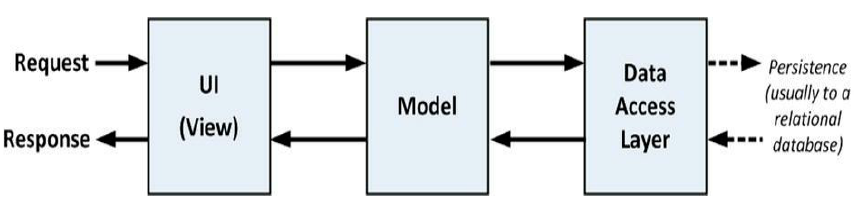
The area in which maintenance problems tend to arise in a Smart UI application is in the business logic, which ends up so diffused across the application that making changes or adding features becomes a troubled process. An improvement in this area is offered by the *model-view* architecture, which pulls out the business logic into a separate domain model. In doing this, the data, processes, and rules are all concentrated in one part of the application, as shown in Figure 3-3.



***Figure 3-3.*** *The model-view pattern*

**Understanding Classic Three-Tier Architectures**

To address the problems of the model-view architecture, the *three-tier* or *three-layer* pattern separates the persistence code from the domain model and places it in a new component called the DAL (Data Access Layer or “Middle Tier”). This is shown in Figure 3-4.



***Figure 3-4.*** *The three-tier pattern*

The three-tier architecture is the most widely used pattern for business applications. It has no constraints on how the UI is implemented and provides good separation of concerns without being too complicated. And, with some care, the DAL can be created so that unit testing is relatively easy. You can see the obvious similarities between a classic three-tier application and the MVC pattern. The difference is that when the UI layer is directly coupled to a click-and-event GUI framework, it becomes almost impossible to perform automated unit tests. And because the UI part of a three-tier application can be complex, there’s a lot of code that can’t be rigorously tested.

In the worst scenario, the three-tier pattern’s lack of enforced discipline in the UI tier means that many such applications end up as thinly disguised Smart UI applications, with no real separation of concerns. This gives the worst possible outcome: an untestable, unmaintainable application that is excessively complex.