." **The model binder uses components known as *value providers* to search for values in different areas of a request. The model binder can look at route data, the query string, and the form collection, and you can add custom value providers if you so desire.**

The model binder is a bit like a search-and-rescue dog. The runtime tells the model binder it wants a value for id, and the binder goes off and looks everywhere to find a parameter with the name id.

Model binding implicitly goes to work when you have an action parameter. You can also explicitly

invoke model binding using the UpdateModel and TryUpdateModel methods in your controller.

The "UpdateModel()" function inspects all the HttpRequest inputs such as Posted Form data, QueryString, Cookies, and Server variables and populate the employee object.

UpdateModel throws an exception if something goes wrong during model binding and the model is

invalid

You can check model state any time after model binding occurs to

see whether model binding succeeded:

TryUpdateModel also invokes model binding, but doesn't throw an exception. TryUpdateModel does return a bool—a value of true if model binding succeeded and the model is valid, and a value of false if something went wrong.

```
[HttpPost]
public ActionResult Edit()
{
var album = new Album();
if (TryUpdateModel(album))
{
db.Entry(album).State = EntityState.Modified;
db.SaveChanges();
return RedirectToAction("Index");
}
else
{
ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
"Name", album.GenreId);
ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
"Name", album.ArtistId);
return View(album);
}
```

}

As mentioned, in addition to using the Bind attribute to restrict implicit model binding, you can also restrict binding when you use UpdateModel and TryUpdateModel. Both methods have an override allowing you to specify an includeProperties parameter. This parameter contains an array of property names you're explicitly allowing to be bound, as shown in the following code:

UpdateModel(product, new[] { "Title", "Price", "AlbumArtUrl" });

If any errors occurred during model binding, model state will contain the names of the properties that caused failures, the attempted values, and the error messages. Although model state is useful for your own debugging purposes, it's primarily used to display error messages to users indicating why their data entry failed and to show their originally entered data (instead of showing default values).

[Bind(Include = "StudentId, StudentName")] Student std

```
[HttpPost]
public ActionResult Edit()
{
var album = new Album();
try
{
```

```
 UpdateModel(album);//explicitno
}

[HttpPost]
 public ActionResult Edit(Album album)//implicitno
 {
}
```

The model binder catches all the failed validation rules and

places them into model state.

Not only does model state contain all the values a user attempted

to put into model properties, but model state also contains all the errors associated with each

property (and any errors associated with the model object itself). If any errors exist in model state,

ModelState.IsValid returns false

You can also look in model state to see the error message associated with the failed validation:

```
var lastNameErrorMessage = ModelState["LastName"].Errors[0].ErrorMessage;
```

Of course, you rarely need to write code to look for specific error messages. Just as the run

time automatically feeds validation errors into model state, it can also automatically pull errors

out of model state.

the built-in HTML helpers use model state (and the presence of errors in model state) to change the display of the model in a view. For example, the

ValidationMessage helper displays error messages associated with a particular piece of view data

by looking at model state.

```
@Html.ValidationMessageFor(m => m.LastName)
```

When validation fails, an action generally re-renders the same

view that posted the model values. Re-rendering the same view allows the user to see all the validation errors and to correct any typos or missing fields. The AddressAndPayment action shown in the

following code demonstrates a typical action behavior:

```
[HttpPost]

public ActionResult AddressAndPayment(Order newOrder)

{

 if (ModelState.IsValid)

 {

 }

 // Invalid -- redisplay with errors

 return View(newOrder);

}
```

When a user submits a form using an HTTP GET request, the browser takes the input names and values inside the form and puts them in the query string.