14

Model Binders and Value Providers

This chapter covers:

* Examining model binding
* Creating a custom model binder
* Extending value providers

The message protocal of the web, HTTP, is decidedly string-centric. Query string and form values in Web Forms and even ASP classic days were represented as loosely-typed key-value string dictionaries. But with the simplicity of controllers and actions came the ability to treat requests as method calls, and post variables as parameters to a method. In order to keep the dictionary abstractions at bay, we need a mechanism to translate dictionary-based requests. In this chapter, we will examine the abstractions ASP.NET MVC uses to translate request variables into action parameters and the different extension points that allow you to add your own translation logic.

14.1 Creating a custom model binder

The model binder in ASP.NET MVC is useful out of the box. It does a great job of taking request and form input and hydrating fairly complex models from them. It supports complex types, lists, arrays, dictionaries, even validation. But a custom binder can also remove another common form of duplication—loading an object from the database based on an action parameter. Most of the time, this action parameter is the primary key of the object or another unique identifier. Instead of putting this repeated data access code in all of our actions, we can use a custom model binder that can load the stored object before the action is executed. Our action can then take the persisted object type as a parameter instead of the unique identifier.

By default, the MVC model binder extensibility allows for registering a model binders by specifying the model type for which the binder should be used. In an application with dozens of entities, it is easy to forget to register the custom model binder for every type. For example, CodeCampServer uses a common base type (Entity) for all entities in the system. Ideally, we could register the custom model binder just once, or just leave it up to each custom binder to decide whether or not it should bind.

To accomplish this, we need to replace the default model binder with our own implementation. Additionally, we can define an interface, IFilteredModelBinder, for our new binders, as shown in listing 14.1.

Listing 14.1 The IFilteredModelBinder interface

public interface IFilteredModelBinder : IModelBinder

{

bool IsMatch(Type modelType);

}

The IFilteredModelBinder inherits from the MVC IModelBinder interface, and adds a method through which implementations can perform custom matching logic. In our case, we can look at the base type of the model type passed in to determine if it is a Entity type. To use custom filtered model binders, we need to create a DefaultModelBinder implementation, as shown in listing 14.2.

Cueballs in code and text

Listing 14.2 A smarter model binder

public class SmartBinder : DefaultModelBinder

{

private readonly IFilteredModelBinder [] \_filteredModelBinders;

public SmartBinder (

params IFilteredModelBinder [] filteredModelBinders) #1

{

\_filteredModelBinders = filteredModelBinders;

}

public override object BindModel ( #2

ControllerContext controllerContext,

ModelBindingContext bindingContext)

{

foreach (var modelBinder in \_filteredModelBinders) #3

{

if (modelBinder.IsMatch(bindingContext.ModelType))

{

return modelBinder.BindModel (controllerContext,

bindingContext); #4

}

}

return base.BindModel (controllerContext, bindingContext);

}

}

Our new SmartBinder class takes an array of IFilteredModelBinders (1), which we’ll fill in soon. Next, it overrides the BindModel method (2). BindModel loops through all of the supplied IFilteredModelBinders, and checks to see if any match the ModelType from the ModelBindingContext (3). If it is a match, we execute and return the result from BindModel for that IFilteredModelBinder (4). The complete class diagram is shown in figure 14.1.



Figure 14.1 The class diagram of our SmartBinder showing the relationship to IFilteredModelBinder

Now that we have a new binder that can match on more than one type, we can turn our attention to our new model binder for loading persistent objects. This new model binder will be an implementation of the IFilteredModelBinder interface. It will need to do a number of things in order to return the correct entity from our persistence layer:

1. Retrieve the request value from the binding context
2. Deal with missing request values
3. Create the correct repository
4. Use the repository to load the entity, and return it.

We won’t cover the third item in much depth, as this example assumes that an IoC container is in place. The entire model binder needs to implement our IFilteredModelBinder, and is shown in listing 14.3.

Cueballs in code and text

Listing 14.3 The EntityModelBinder

public class EntityModelBinder : IFilteredModelBinder

{

public bool IsMatch(Type modelType) #1

{

return typeof(Entity).IsAssignableFrom(modelType);

}

public object BindModel (

ControllerContext controllerContext,

ModelBindingContext bindingContext)

{

ValueProviderResult value =

bindingContext.ValueProvider [bindingContext.ModelName]; #2

if (value == null) |#3

return null; |#3

if (string.IsNullOrEmpty(value.AttemptedValue)) |#3

return null; |#3

var entityId = new Guid(value.AttemptedValue); #4

Type repositoryType = typeof(IRepository<>) |#5

.MakeGenericType(bindingContext.ModelType); |#5

var repository = (IRepository)IoC.Resolve(repositoryType); |#5

Entity entity = repository.GetById(entityId);

return entity;

}

}

In listing 14.3 we implement our newly created interface, IFilteredModelBinder. The additional method, IsMatch (1), returns true when the model type being bound by ASP.NET MVC is a Entity, our base type for all model objects persisted in a database. Next, we have to implement the BindModel method by following the steps laid out just before listing 14.3. First, we retrieve the request value from the ModelBindingContext (2) passed in to the BindModel method. The ModelBindingContext contains a dictionary of strings to ValueProviderResults in the ValueProvider property. If the ValueProviderResult does not exist, or the attempted value does not exist, we won’t try to retrieve the entity from the repository (3). Although the entity’s identifier is a Guid, the attempted value is a string, so we construct a new Guid from the attempted value on the ValueProviderResult (4).

Now that we have the parsed Guid from the request, we can create the appropriate repository from our IoC container (5). But because we have specific repositories for each kind of entity, we don’t know the specific repository type at compile time. However, all of our repositories implement a common interface, as shown in listing 14.4.

Listing 14.4 The common repository interface

public interface IRepository<TEntity>

where TEntity : Entity

{

TEntity GetById(Guid id);

}

We want the IoC container to create the correct repository given the type of entity we are attempting to bind. This means we need to figure out and construct the correct Type object for the IRepository we create. We do this by using the Type.MakeGenericType method to create a closed generic type from the open generic type IRepository<>.

Open and closed generic types

An open generic type is simply a generic type that has no type parameters supplied. IList<> and IDictionary<,> are both open generic types. To create instances of a type, we must create a closed generic type from the open generic type. A closed generic type is a generic type with type parameters supplied, such as IList<int> and IDictionary<string, User>.

When we have the closed generic type for IRepository using the ModelBindingContext.ModelType property, we can use our IoC container to create an instance of the repository to call and use. Finally, we call the repository’s GetById method and return the retrieved entity from BindModel. Because we cannot call a generic method at runtime without using reflection, we use another nongeneric IRepository interface that returns only objects as Entity, as shown in listing 14.5.

Listing 14.5 The nongeneric repository interface

public interface IRepository

{

Entity GetById(Guid id);

}

All repositories in our system inherit from a common repository base class, which implements both the generic and nongeneric implementations of IRepository. Because some places cannot hold references to the generic interface (as we encountered with model binding) the additional nongeneric IRepository interface supports these scenarios.

We have our SmartBinder and EntityModelBinder, which binds to entities from request values, but we still need to configure ASP.NET MVC to use these binders instead of the default model binder. To do this, we set the ModelBinders.Binders.DefaultBinder property in our application startup code, as shown in listing 14.6.

Listing 14.6 Replacing the default model binder

protected void Application\_Start()

{

ModelBinders.Binders.DefaultBinder =

new SmartBinder (new EntityModelBinder ());

At this point, we have only a single filtered model binder. In practice, we might have specialized model binders for certain entities, classes of objects (such as enumeration classes), and so on. By creating a model binder for entities, we can create controller actions that take entities as parameters, as opposed to just a Guid, as shown in listing 14.7.

Listing 14.7 Controller action with an entity as a parameter

public ViewResult Edit(Profile id)

{

return View(new ProfileEditModel(id));

}

With the EntityModelBinder in place, we avoid repeating code in our controller actions. Our edit screen, shown in figure 14.2, now becomes simpler to create without the boring repository lookups.

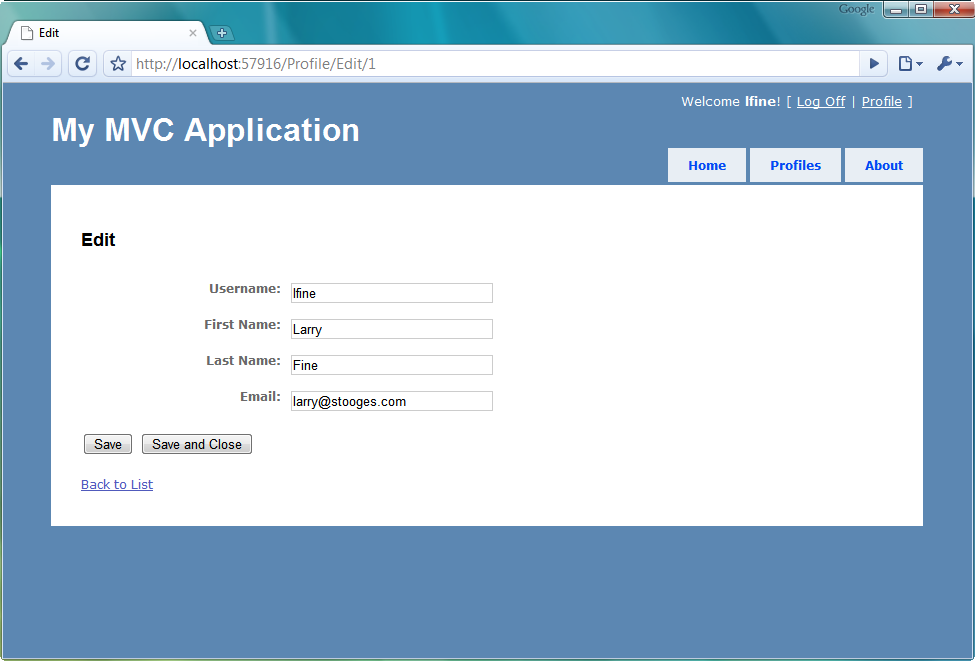


Figure 14.2 The edit screen now skips the need to load the Profile manually

This repetition would obscure the intent of the controller action with data access code that is not relevant to what the controller action is trying to accomplish.

14.2 Using custom value providers

In ASP.NET MVC 1.0, the responsibility of inspecting the various dictionary sources for values to bind was left to each individual model binder. This meant that if you wanted to supply new sources of values besides just the form variables, you needed to override large portions of the default model binder. If we had a model with mixed sources, whether it was from Session, a configuration, files and so on, it was a tricky maneuver to modify the default model binder to bind from multiple sources. With ASP.NET MVC 2, the concept of providing values to the model binder is abstracted into the IValueProvider interface, shown in listing 14.8 below.

Listing 14.8 The IValueProvider interface

public interface IValueProvider {

bool ContainsPrefix(string prefix);

ValueProviderResult GetValue(string key);

}

Internally, the DefaultModelBinder uses an IValueProvider to build the ValueProviderResult. It then uses the ValueProviderResult to bind up our complex models. To create a new custom value provider, we need to implement two key interfaces. The first is IValueProvider, and to allow the MVC framework to build our custom value provider, an implementation of ValueProviderFactory. The MVC framework ships with several value providers out of the box, bundled together in the ValueProviderFactories class, shown in listing 14.9.

Listing 14.9 The ValueProviderFactories class

public static class ValueProviderFactories {

private static readonly ValueProviderFactoryCollection \_factories =

new ValueProviderFactoryCollection() {

new FormValueProviderFactory(),

new RouteDataValueProviderFactory(),

new QueryStringValueProviderFactory(),

new HttpFileCollectionValueProviderFactory()

};

public static ValueProviderFactoryCollection Factories {

get {

return \_factories;

}

}

}

We can see from listing 14.9 that the initial value providers include implementations to support binding from form values, route values, query string and the files collection. However, we would like to add a new value provider, to bind values from Session. To add a new value provider, we simply need to add our custom value provider factory to the ValueProviderFactories.Factories collection, usually in application startup where we would also configure areas, routes and so on, as shown in listing 14.10.

Listing 14.10 Registering our custom value provider factory

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

ValueProviderFactories.Factories.Add(new SessionValueProviderFactory());

RegisterRoutes(RouteTable.Routes);

}

Instead of adding a value provider directly, ASP.NET MVC requires us to build a factory object to supply our custom value provider. For each request, the default model binder builds the entire collection of value providers from the registered value provider factories. Our SessionValueProviderFactory becomes quite simple, shown in listing 14.11.

Listing 14.11 The SessionValueProviderFactory class

public class SessionValueProviderFactory : ValueProviderFactory

{

public override IValueProvider GetValueProvider(

ControllerContext controllerContext)

{

return new SessionValueProvider(

controllerContext.HttpContext.Session);

}

}

We create our custom value provider factory by inheriting from ValueProviderFactory and overriding the GetValueProvider method. For each request, our custom SessionValueProvider will be instantiated, passing in the current request's Session object. The constructor is shown in listing 14.12 below.

Listing 14.12 The SessionValueProvider class and constructor

public class SessionValueProvider : IValueProvider

{

public SessionValueProvider(HttpSessionStateBase session)

{

AddValues(session);

}

When our SessionValueProvider is instantiated with the current Session, we want to examine the Session object and cache the possible results. In listing 14.13, we keep a list of prefixes and values for later matching.

Listing 14.13 The local values cache and AddValues method

private readonly HashSet<string> \_prefixes

= new HashSet<string>(StringComparer.OrdinalIgnoreCase);

private readonly Dictionary<string, ValueProviderResult> \_values

= new Dictionary<string, ValueProviderResult>(StringComparer.OrdinalIgnoreCase);

private void AddValues(HttpSessionStateBase session)

{

if (session.Keys.Count > 0) #1

{

\_prefixes.Add(""); #2

}

foreach (string key in session.Keys) #3

{

if (key != null)

{

\_prefixes.Add(key); #4

object rawValue = session[key];

string attemptedValue = session[key].ToString();

\_values[key] = new ValueProviderResult( |#5

rawValue, |#5

attemptedValue, |#5

CultureInfo.CurrentCulture); |#5

}

}

}

In the listing above, we first check to see if our Session object contains any keys (1). If so, we register a blank prefix to match (2). Next, we loop through every key in our Session (3), adding each key as an available prefix to match to our \_prefixes collection (4). After that, we pull every value out of Session, creating a new ValueProviderResult object (5) for each key/value pair found in Session. Each ValueProviderResult is then added to our local \_values dictionary. Because we figure out every possible prefix and value provider result when our SessionValueProvider is instantiated, implementing the other two required IValueProvider methods becomes very straightforward, as shown in listing 14.14.

Listing 14.14 The ContainsPrefix and GetValue methods

public bool ContainsPrefix(string prefix)

{

return \_prefixes.Contains(prefix); #1

}

public ValueProviderResult GetValue(string key) |#2

{ |#2

ValueProviderResult result; |#2

|#2

\_values.TryGetValue(key, out result); |#2

|#2

return result; |#2

}

In the ContainsPrefix method (1), we return a boolean signifying that our IValueProvider can match against the specified prefix. This is simply a lookup in our previously built HashSet of keys found in the current request's Session. If ContainsPrefix returns true, then our value provider will be chosen by the DefaultModelBinder to provide a result, in the GetValue method (2). Again, because we previously built up all possible ValueProviderResults, we can simply return the cached result.

So how do we take advantage of our new custom SessionValueProvider? We already registered the SessionValueProviderFactory. Next, we need some code to actually use Session. In our AccountController's LogOn action, we include some code to push the logged on user's Profile into Session, shown in listing 14.15.

Listing 14.15 Adding the current user's Profile to Session

var profile = \_profileRepository.Find(model.UserName);

if (profile == null)

{

profile = new Profile(model.UserName);

\_profileRepository.Add(profile);

}

Session[CurrentUserKey] = profile;

FormsService.SignIn(model.UserName, rememberMe);

The CurrentUserKey is a local constant in our AccountController class, shown in listing 14.16.

Jimmy, there is 1 figure in this whole chapter. After figure 14.1, please add screenshots with the steps the reader needs to go through in the application to demonstrate the code th at is being listed.

Listing 14.16 The key value used for Session

[HandleError]

public class AccountController : Controller

{

public const string CurrentUserKey = "CurrentUser";

If we recall our SessionValueProvider, it provides values for members that match any of the Session's key values. In our case for the current user's Profile, we only need to name a member as "CurrentUser", with a type of Profile, and the DefaultModelBinder will bind our value appropriately. For example, we might have a child action that shows the current user, if logged in, shown in listing 14.17.

Listing 14.17 A LogOnWidget child action for display current user information

[ChildActionOnly]

public ViewResult LogOnWidget(LogOnWidgetModel model)

{

bool isAuthenticated = Request.IsAuthenticated;

model.IsAuthenticated = isAuthenticated;

return View(model);

}

Previously, we would need to retrieve the Profile object by pulling directly from Session or loading from some other persistent store. But now we can modify our LogOnWidgetModel to include a CurrentUser member, shown in listing 14.18.

Listing 14.18 The LogOnWidgetModel with a CurrentUser member

public class LogOnWidgetModel

{

public bool IsAuthenticated { get; set; }

public Profile CurrentUser { get; set; }

}

Because the CurrentUser member name matches up with our Session key, the SessionValueProvider will pull the Profile out of Session, hand it to the DefaultModelBinder, which will finally provide this value for the CurrentUser property. The log on widget will now skip the database altogether, shown in figure 14.3.

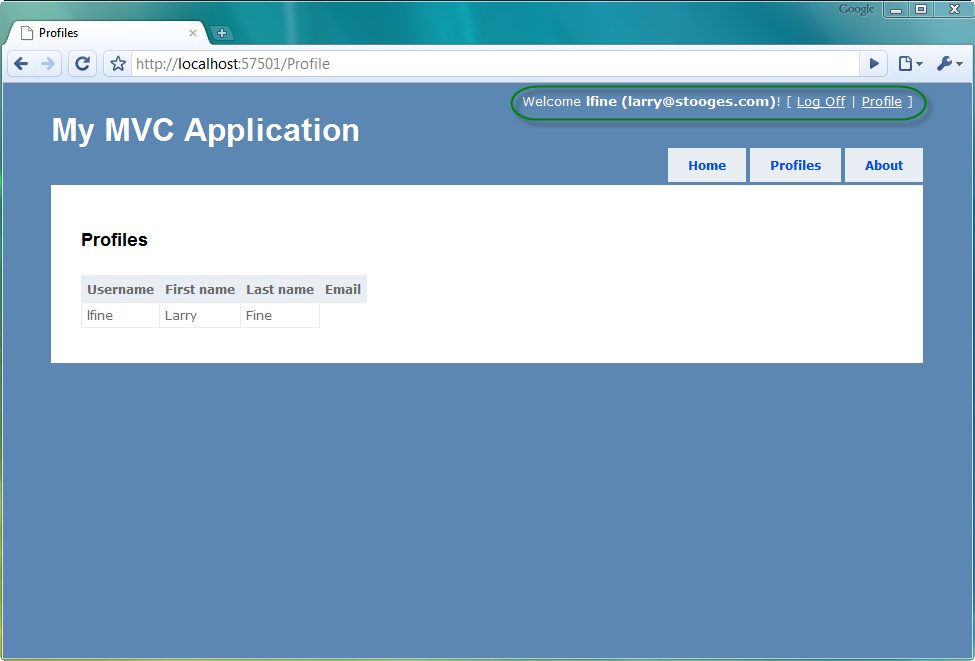


Figure 14.3 The log on widget pulls profile information straight from session

As long as the name matches up to our Session key, the value will be populated appropriately. We are not strictly limited to posted form values or route values for values provided to model binding. We can now bind from whatever locations we need.

One final note to keep in mind - value providers are evaluated in the order that they are added to the ValueProviderFactories.Factories collection. In our example, the SessionValueProviderFactory was added after all of the default, built-in value provider factories. This means that if we have a posted form value of "CurrentUser", its value would be used instead of the Session value.

14.3 Summary

The components that allow rich form posting and model binding are critical pieces of the ASP.NET MVC framework. They prevent the need to resort to examining the underlying Request object. The combination of custom model binders and custom value providers allows us to keep the existing rich binding behavior and extend it for custom and more exotic scenarios. The value provider abstraction added in ASP.NET MVC 2 expands the possibilities for providing model binding values beyond the traditional form and query string variables without resorting to heavily modifying the underlying model binding behavior.