

14

Model binders and value providers

This chapter covers

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

The messaging protocol of the web, HTTP, is decidedly string-centric. Query-string and form values in Web Forms and even classic ASP applications 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 to post variables as parameters to a method. To keep the dictionary abstractions at bay, we need a mechanism to translate string-based input into strongly typed objects.

In this chapter, we'll examine the abstractions ASP.NET MVC uses to translate request variables into action parameters and the extension points that allow you to add your own translation logic.

14.1 Creating a custom model binder

The default 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, so instead of putting this repeated data access code in all 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 binder by specifying the model type for which the binder should be used, but in an application with dozens of entities, it's easy to forget to register the custom model binder for every type. Ideally, we could register the custom model binder just once for a common base type, or leave it up to each custom binder to decide whether 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` implements the `IModelBinder` interface and adds a method through which implementations can perform custom matching logic. In our case, we can look at the model type passed to the binder to determine if it inherits from our common base type, `Entity`.

To use custom filtered model binders, we need to create an implementation that inherits from `DefaultModelBinder`, as shown in listing 14.2.

Listing 14.2 A smarter model binder

```
public class SmartBinder : DefaultModelBinder
{
    private readonly IFilteredModelBinder [] _filteredModelBinders;

    public SmartBinder (
        params IFilteredModelBinder[]
        filteredModelBinders)
    {
        _filteredModelBinders = filteredModelBinders;
    }

    public override object BindModel (
```

① Accepts array of `IFilteredModelBinder`

② Overrides `BindModel`

```

ControllerContext controllerContext,
ModelBindingContext bindingContext)
{
    foreach (var modelBinder in
        _filteredModelBinders)
    {
        if (modelBinder.IsMatch(bindingContext.ModelType))
        {
            return modelBinder.BindModel (controllerContext,
                bindingContext);
        }
    }

    return base.BindModel (controllerContext, bindingContext);
}

```

③ Checks if binder should execute

④ Returns result of binding

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

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'll have to do a number of things to return the correct entity from our persistence layer:

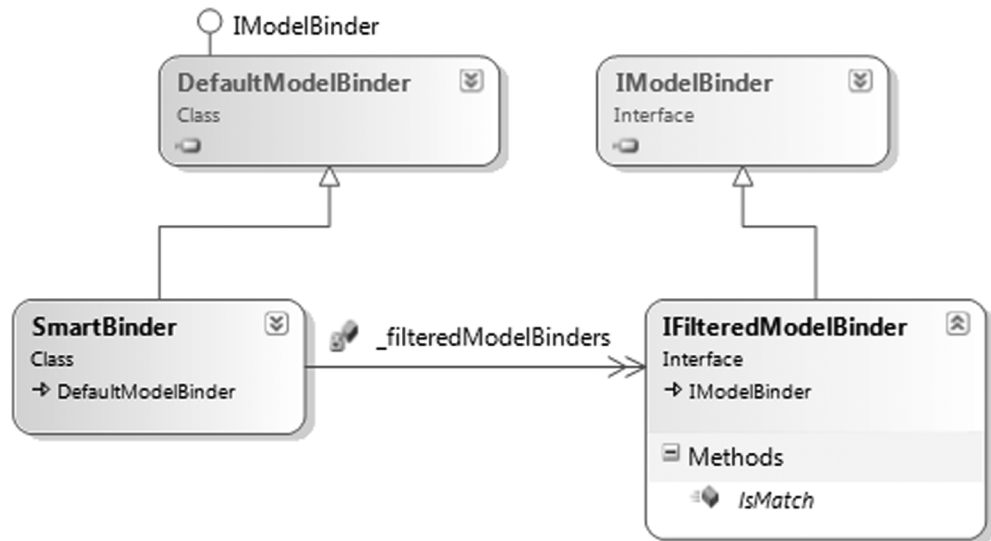


Figure 14.1 The class diagram of our SmartBinder showing the relationship to IFilteredModelBinder

- 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` interface and is shown in listing 14.3.

Listing 14.3 The `EntityModelBinder`

```
public class EntityModelBinder : IFilteredModelBinder
{
    public bool IsMatch(Type modelType)
    {
        return typeof(Entity).IsAssignableFrom(modelType);
    }

    public object BindModel (
        ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        ValueProviderResult value =
            bindingContext.ValueProvider
                .GetValue(bindingContext.ModelName);

        if (value == null)
            return null;

        if (string.IsNullOrEmpty(value.AttemptedValue))
            return null;

        int entityId;

        if (!int.TryParse(value.AttemptedValue,
            out entityId))
        {
            return null;
        }

        Type repositoryType = typeof(IRepository<>)
            .MakeGenericType(bindingContext.ModelType);
        var repository = (IRepository)IoC
            .Resolve(repositoryType);
        Entity entity = repository.GetById(entityId);
        return entity;
    }
}
```

1 Implements IsMatch

2 Retrieves request value

3 Returns when no value specified

4 Converts value to int

5 Resolves repository from container

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 an instance of `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` ❷ passed in to the `BindModel` method. The `ValueProvider` property can be used to retrieve `ValueProviderResult` instances that represent the data from form posts, route data, and the query string. If there's no `ValueProviderResult` that has the same name as our action parameter, we won't try to retrieve the entity from the repository ❸. Although the entity's identifier is an integer, the attempted value is a string, so we construct a new `int` from the attempted value on the `ValueProviderResult` ❹.

Once we've the parsed integer from the request, we can create the appropriate repository from our IoC container ❺. But because we have specific repositories for each kind of entity, we don't know the specific repository type at compile time. However, all 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 Get(int id);
}
```

We want the IoC container to create the correct repository given the type of entity we're 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 a generic type that has no type parameters supplied. `IList<>` and `IDictionary<,>` are both open generic types. A *closed generic type* is a generic type with type parameters supplied, such as `IList<int>` and `IDictionary<string, User>`.

To create instances of a type, we must create a closed generic type from the open generic type.

When the `ModelBindingContext.ModelType` property refers to a closed generic type for `IRepository`, we can use our IoC container to create an instance of the repository to call and use.

Finally, we call the repository's `Get` method and return the retrieved entity from `BindModel`. Because we can't 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 Get(int 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 can't 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 our `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 an integer, as shown in listing 14.7.

Listing 14.7 Controller action with an entity as a parameter

```
public ActionResult 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.

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

Controllers should control the storyboard of the application, and data lookups can easily be factored out of them and into model binders. The built-in model binder looks for action parameters in the forms collection, the route values, and the query string. By registering a custom value provider, we can easily extend the list of locations automatically checked by the model binder.

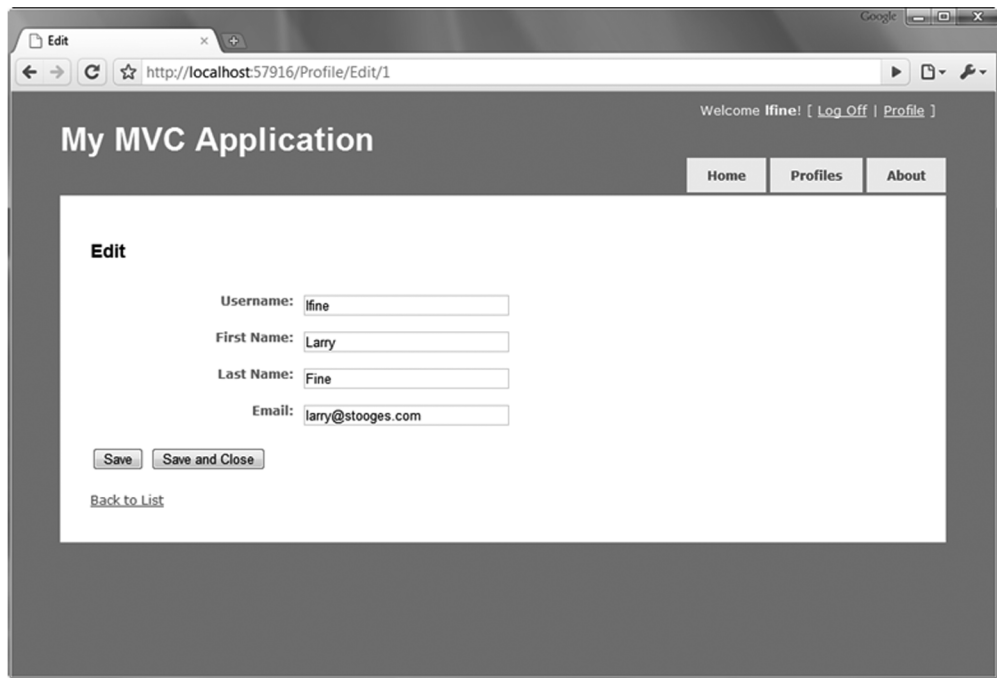


Figure 14.2 The Edit screen now skips the need to load the profile manually.

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 we wanted to supply new sources of values besides just the form variables, we 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, modifying the default model binder to bind from multiple sources was tricky.

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.

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 obtain the values used to bind our complex models. To create a new custom value provider, we need to implement two key interfaces. The first is *IValueProvider*; the second, to allow the MVC framework to build our custom value provider, is 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 that support binding from form values, route values, the query string, and the files collection. But we'd 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 at application startup, where we'd 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, as 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.

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 cache the prefixes and values obtained from `Session` 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)
    {
        _prefixes.Add("");
    }

    foreach (string key in session.Keys)
    {
        if (key != null)
        {
            _prefixes.Add(key);

            object rawValue = session[key];
            string attemptedValue = session[key].ToString();
            _values[key] = new ValueProviderResult(
                rawValue,
                attemptedValue,
                CultureInfo.CurrentCulture);
        }
    }
}

```

In listing 14.13, we first check to see if our `Session` object contains any keys ❶. If so, we register a blank prefix to match ❷. Next, we loop through every key in our `Session` ❸,

adding each key as an available prefix to match to our `_prefixes` collection ❹. After that, we pull every value out of `Session`, creating a new `ValueProviderResult` object ❺ 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 straightforward, as shown in listing 14.14.

Listing 14.14 The `ContainsPrefix` and `GetValue` methods

```
public bool ContainsPrefix(string prefix)
{
    return _prefixes.Contains(prefix);
}

public ValueProviderResult GetValue(string key)
{
    ValueProviderResult result;
    _values.TryGetValue(key, out result);
    return result;
}
```

In the `ContainsPrefix` method, 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, our value provider will be chosen by the `DefaultModelBinder` to provide a result in the `GetValue` method. 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 use `Session`. From the default project template, you're familiar with the `AccountController`. In the `AccountController`'s `LogOn` action, we include some code to push the logged-on user's `Profile` into `Session`, as shown in listing 14.15. We're working toward the result shown in figure 14.3.

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);
```

We're finding the Profile and saving it to Session so that the value provider can find it. The `CurrentUserKey` is a local constant in our `AccountController` class, shown in listing 14.16.

Listing 14.16 The key value used for Session

```
[HandleError]
public class AccountController : Controller
{
    public const string CurrentUserKey = "CurrentUser";
    ...
}
```

If you 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 by extracting the Profile instance from the Session. For example, we might have a child action that shows the current user, if logged in, as shown in listing 14.17.

Listing 14.17 A LogOnWidget child action for displaying current user information

```
[ChildActionOnly]
public ActionResult LogOnWidget(LogOnWidgetModel model)
{
    bool isAuthenticated = Request.IsAuthenticated;
    model.IsAuthenticated = isAuthenticated;
    return View(model);
}
```

Previously, we'd have needed 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, as 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 logon widget will now skip the database altogether, as shown in figure 14.3.

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

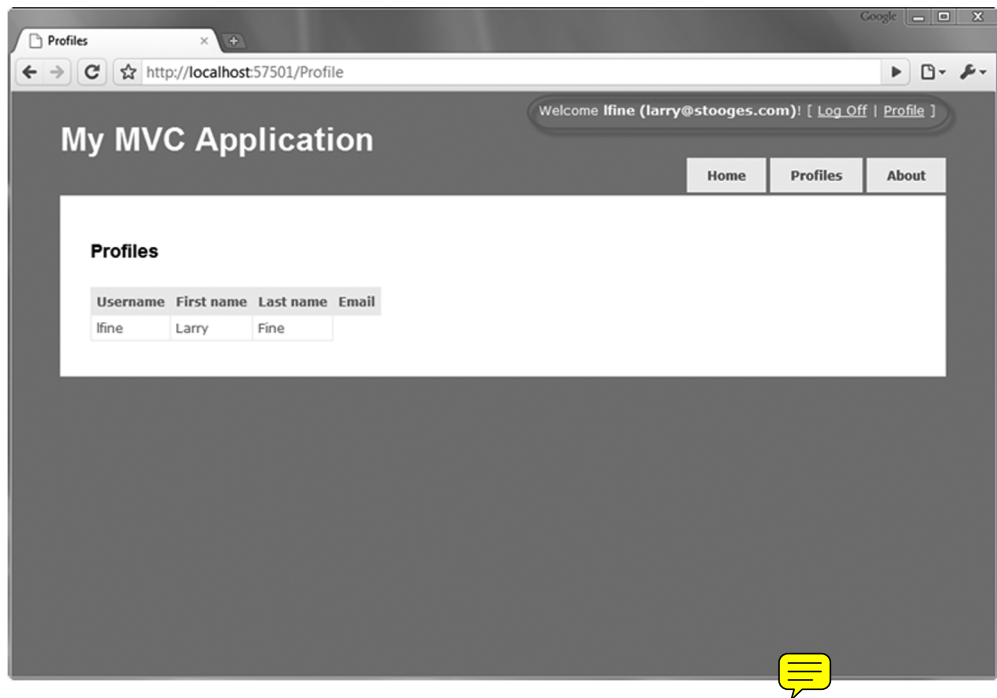


Figure 14.3 The logon widget pulls profile information straight from **Session**

One final note to keep in mind—value providers are evaluated in the order that they're added to the `ValueProviderFactories.Factories` collection. In our example, the `SessionValueProviderFactory` was added after all the default, built-in value provider factories. This means that if we've a posted form value of "CurrentUser", its value will 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 eliminate 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 heavily modifying the underlying model binding behavior.

In the next chapter, we'll look at how ASP.NET MVC 2 can be used to validate user input on both the server and the client.