Singly-Registered Services

MVC has services that it consumes for which the user can register one (and exactly one) instance of that service. It calls these services *singly-registered services*, and the method used to retrieve singly-registered services from the resolver is `GetService`.

For all the singly-registered services, MVC consults the dependency resolver for the service the first time it is needed, and caches the result for the lifetime of the application. You can either use the dependency resolver API or the traditional registration API (when available), but you cannot use both because MVC is expecting to use exactly one instance of any singly-registered service.

Implementers of `GetService` should return an instance of the service that is registered in the resolver, or return `null` if the service is not present in the resolver.

Service: Controller Factory

Requested type: `IControllerFactory`

Traditional registration API: `ControllerBuilder.Current.SetControllerFactory`

Default implementation: `DefaultControllerFactory`

Translates controller names into controller types, and controller types into controller instances. In MVC 1.0 and MVC 2, this was the primary "hook point" for introducing dependency injection into the system, because the classes that developers primarily wanted to get dependency injection performed upon were controllers. For MVC 3 applications, it's not very common to need an implementation of `IControllerFactory`. Unless you need to change the mapping of names to types, you're much better off either allowing the dependency resolver to directly create controller instances (see the section titled "Creating Arbitrary Objects") or registering a controller activator.

Service: Controller Activator

Requested type: `IControllerActivator`

Traditional registration API: None

Default implementation: `DefaultControllerActivator`

A new service introduced in MVC 3; this turn controller types into controller objects. Added to MVC 3 to support dependency resolvers that aren't capable of building arbitrary types without preconfiguration; as such, it doesn't have a traditional registration API (because it's only intended to be used in coordination with a dependency resolver).

It is actually the `DefaultControllerFactory` class, and not the MVC framework itself, that understands and consumes the controller activator. Because the conversion of controller type into controller object has historically been the responsibility of the controller factory, it remains that the controller factory uses the controller activator to perform this operation. As such, if you register a controller factory that does not have this same behavior, it's possible that your MVC application will never use a controller activator.

Model Metadata Provider

Requested type: `ModelMetadataProvider`

Traditional registration API: `ModelMetadataProviders.Current`

Default implementation: `DataAnnotationsModelMetadataProvider`

The model metadata provider is responsible for returning information about model classes inside of MVC applications. The metadata returned by this provider includes several pieces of information, including display names, formatting instructions, data types, template names, and more.

For more information on model metadata providers, please see Chapter 6.

Service: View Page Activator

Requested type: `IViewPageActivator`

Traditional registration API: None

Default implementation: `DefaultViewPageActivator`

Like the controller activator, this is a new service introduced in MVC 3. And like the controller activator, it exists to support dependency resolvers that may not be able to create arbitrary objects. For this reason, it does not have a traditional registration API.

The view page activator is consumed by a view engine base class in MVC, `BuildManagerViewEngine`. This base class is responsible for turning view files (like the `.aspx` files of `WebForms` views, or the `.cshtml` files of Razor views) into implementation code using the `BuildManager` class in ASP .NET. Once the views have been converted into classes, the view engine uses the view page activator to create instances of those classes.

View engines that use `BuildManagerViewEngine` as their base class should get this behavior for free. For consistency, view engines that do not use this base class should use the dependency resolver to find the view page activator service, and use that service to create the view page objects.

Multiply-Registered Services

In contrast with singly-registered services, MVC also consumes some services where the user can register many instances of the service, which then compete or collaborate to provide information to MVC. It calls these kinds of services *multiply-registered services*, and the method that is used to retrieve multiply-registered services from the resolver is `GetServices`.

For all the multiply-registered services, MVC consults the dependency resolver for the services the first time they are needed, and caches the results for the lifetime of the application. You can use both the dependency resolver API and the traditional registration API, and MVC combines the results in a single merged services list. Services registered in the dependency resolver come before services registered with the traditional registration APIs. This is important for those multiply-registered services that compete to provide information; that is, MVC asks each service instance one-by-one to provide information, and the first one that provides the requested information is the service instance that MVC will use.

Implementers of `GetServices` should always return a collection of implementations of the service type that are registered in the resolver, or return an empty collection if there are none present in the resolver.

When listing the multiply-registered services that MVC supports, there is a designation titled "multi-service model," with one of two values:

- **Competitive services:** Those where the MVC framework will go from service to service (in order), and ask the service whether it can perform its primary function. The first service that responds that it can fulfill the request is the one that MVC uses. These questions are typically asked on a request-by-request basis, so the actual service that's used for each request may be different. An example of competitive services is the view engine service: Only a single view engine will render a view in a particular request.
- **Cooperative services:** Those where the MVC framework asks every service to perform its primary function, and all services that indicate that they can fulfill the request will contribute to the operation. An example of cooperative services is filter providers: every provider may find filters to run for a request, and all filters found from all providers will be run.

Service: Filter Provider

Requested type: `IFilterProvider`

Traditional registration API: `FilterProviders.Providers`

Default implementations:
`FilterAttributeFilterProvider`
`GlobalFilterCollection`
`ControllerInstanceFilterProvider`

Multi - service model: Cooperative

This is expected to return lists of filters that are associated with a given request (controller and action). Because filter providers are collaborative, all filters from all providers will execute during the request at the appropriate times.

Three filter providers are registered by default:

- The global filter list is contained inside of an instance of `GlobalFilterCollection`, which itself is a filter provider.
- Each controller object is itself also a filter, because it implements the four filter interfaces, so `ControllerInstanceFilterProvider` returns the controller itself as one of the filters for the request.
- Controller classes and action methods can be decorated with filters in the form of attributes. The `FilterAttributeFilterProvider` class uses reflection to find those filter attributes.

Model Binder Provider

Requested type: `IModelBinderProvider`

Traditional registration API: `ModelBinderProviders.BinderProviders`

Default implementations: None

Multi - service model: Competitive

These were introduced in MVC 3 to support dependency injection for model binders. From a service consumption perspective, MVC uses model binder providers to help find model binders; you inject the providers themselves rather than the binders, because of this mapping from binder to supported type. In prior versions of MVC, you could register model binders statically through `ModelBinders.Binders`, but this API wasn't suitable for dependency injection. This old API was a dictionary that mapped incoming model types to appropriate model binder instances. Because developers were forced to provide instances ahead of time, this API wasn't appropriate for dependency injection.

There are no default model binder providers because all the default model binders are registered with the traditional registration API. Because model binder providers are competitive, you can consider the old dictionary-based API to be a "model binder provider of last resort"; that is, the dictionary is consulted only in the event that there are no model binder providers that could provide a model binder for the given type.

Service: Validation Provider

Requested type: `ModelValidatorProvider`

Traditional registration API: `ModelValidatorProviders.Providers`

Default implementations:
`DataAnnotationsModelValidatorProvider`
`DataErrorInfoModelValidatorProvider`
`ClientDataTypeModelValidatorProvider`

Multi - service model: Cooperative

These participate in providing verification of business validation rules for models during model binding, as well as providing client-side validation hints to the runtime when client validation is enabled.

The following validation providers are registered by default in an MVC application:

- Classes and properties decorated with validation attributes from the DataAnnotations library are found via the `DataAnnotationsModelValidatorProvider`.
- Classes can also implement IDataErrorInfo for model level validation, which is supported by `DataErrorInfoModelValidatorProvider`.

Client-side validation information based on built-in simple types (that is, numbers) are discovered by `ClientDataTypeModelValidatorProvider`.

Value Provider Factory

Requested type: `ValueProviderFactory`

Traditional registration API: `ValueProviderFactories.Factories`

Default implementations (*in order*):
`ChildActionValueProviderFactory`
`FormValueProviderFactory`
`JsonValueProviderFactory`

```
RouteDataValueProviderFactory
QueryStringValueProviderFactory
HttpFileCollectionValueProviderFactory
```

Multi - service model: Competitive

Value providers are used during model binding in MVC to populate the values of models and model properties. A value provider generally pulls data from a single source, and the ordering of the providers dictates their precedence in providing values.

In MVC 1.0, value providers were in a simple list. In MVC 2, the concept of value provider factories was introduced to assist in dependency injection as well as providing the opportunity for contextual and stateful implementations of value providers.

The value provider factories themselves are not directly competitive, but their ordering dictates the ordering of value providers (which makes the factories indirectly competitive).

Service: View Engine

Requested type: `IViewEngine`

Traditional registration API: `ViewEngines .Engines`

Default implementations (*in order*): `WebFormViewEngineRazorViewEngine`

Multi - service model: Competitive

View engines are responsible for locating and rendering views and partial views. They may also be consumers of the view page activator described in the previous section, especially if the view engine derives from `BuildManagerViewEngine`.

View engines are competitive, but they rarely end up directly competing with one another, because developers generally only write a single type of view (or, if they are mixing views, they don't give views from two different view engines the same name). An exception to this rule is when a developer is porting views from one view engine to another. In those situations, it may be advantageous to reorder the view engines such that your newest view engine comes first, so that you can leave older views in place while upgrading to the new view engine.

Creating Arbitrary Objects

In MVC 3, there are two special cases where the MVC framework will request a dependency resolver to manufacture *arbitrary objects*; that is, objects that are not (strictly speaking) services. Those objects are controllers and view pages.

As you saw in the previous two sections, two services called activators control the instantiation of controllers and view pages. The default implementations of these activators ask the dependency resolver to create the controllers and view pages, and failing that, they will fall back to calling `Activator.CreateInstance`.

Creating Controllers

If you've ever tried to write a controller with a constructor with parameters before, at run time you'll get an exception that says "No parameterless constructor defined for this object." In an

MVC 3 application, if you look closely at the stack trace of the exception, you'll see that it includes `DefaultControllerFactory` as well as `DefaultControllerActivator`.

The controller factory is ultimately responsible for turning controller names into controller objects, so it is the controller factory that consumes `IControllerActivator` rather than MVC itself. The default controller factory in MVC 3 splits this behavior into two separate steps: the mapping of controller names to types, and the instantiation of those types into objects. The latter half of the behavior is what the controller activator is responsible for.

CUSTOM CONTROLLER FACTORIES AND ACTIVATORS

It's important to note that because the controller factory is ultimately responsible for turning controller names into controller objects, any replacement of the controller factory may disable the functionality of the controller activator. In MVC versions prior to MVC 3, the controller activator did not exist, so any custom controller factory designed for an older version of MVC will not know about the dependency resolver or controller activators. If you write a new controller factory, you should consider using controller activators whenever possible.

Because the default controller activator simply asks the dependency resolver to make controllers for you, many dependency injection containers automatically provide dependency injection for controller instances, because they have been asked to make them. If your container can make arbitrary objects without preconfiguration, you should not need to create a controller activator; simply registering your dependency injection container should be sufficient.

However, if your dependency injection container does not like making arbitrary objects, it will also need to provide an implementation of the activator. This allows the container to know that it's being asked to make an arbitrary type that may not be known of ahead of time, and allow it to take any necessary actions to ensure that the request to create the type will succeed.

The controller activator interface contains only a single method:

```
public interface IControllerActivator
{
       IController Create(RequestContext requestContext, Type controllerType);
}
```

In addition to the controller type, the controller activator is also provided with the `RequestContext`, which includes access to the `HttpContext` (including things like `Session` and `Request`), as well as the route data from the route that mapped to the request. You may also choose to implement a controller activator to help make contextual decisions about how to create your controller objects because it has access to the context information. One example of this might be an activator that chooses to make different controller classes based on whether the logged in user is an administrator or not.

Creating Views

Much like the controller activator is responsible for creating instances of controllers, the view page activator is responsible for creating instances of view pages. Again, because these types are arbitrary

types that a dependency injection container will probably not be preconfigured for, the activator gives the container an opportunity to know that a view is being requested.

The view activator interface is similar to its controller counterpart:

```
public interface IViewPageActivator
{
    object Create(ControllerContext controllerContext, Type type);
}
```

In this case, the view page activator is given access to the `ControllerContext`, which contains not only the `RequestContext` (and thus `HttpContext`), but also a reference to the controller, the model, the view data, the temp data, and other pieces of the current controller state.

Similar again to its controller counterpart, it is the case that the view page activator is a type that is indirectly consumed by the MVC framework, rather than directly. In this instance, it is the `BuildManagerViewEngine` (the abstract base class for `WebFormViewEngine` and `RazorViewEngine`) that understands and consumes the view page activator.

A view engine's primary responsibility is to convert view names into view instances. In MVC 3, the MVC framework splits the actual instantiation of the view page objects out into the view activator, while leaving the identification of the correct view files and compilation of those files to the build manager view engine base class.

ASP.NET'S BUILD MANAGER

The compilation of views into classes is the responsibility of a component of the core ASP.NET run time called `BuildManager`. This class has many duties, including converting `.aspx` and `.ascx` files into classes for consumption by WebForms applications.

The build manager system is extensible, like much of the ASP.NET core run time, so you can take advantage of this compilation model to convert input files into classes at run time in your applications. In fact, the ASP.NET core run time doesn't know anything about Razor; the ability to compile `.cshtml` and `.vbhtml` files into classes exists because the ASP.NET Web Pages team wrote a build manager extension called a build provider.

Examples of third-party libraries that did this were the earlier releases of the Subsonic project, an object-relational mapper (ORM) written by Rob Conery. In this case, SubSonic would consume a file that described a database to be mapped, and at run-time, it would generate the ORM classes automatically to match the database tables.

The build manager operates during design time in Visual Studio, so any compilation that it's doing is available while writing your application. This includes IntelliSense support inside of Visual Studio.

SUMMARY

The dependency resolver in ASP.NET MVC 3 enables several new and exciting opportunities for dependency injection in your web applications. This can help you design applications that reduce tight coupling and encourage better pluggability, which tends to lead to more flexible and powerful application development.