# Composition services

## The composition infrastructure

While it is true that the composition (see also Inversion of Control or Dependency Injection) provides an abstraction level which, in the beginning, may be hard to understand with respect to the way it works, in the end it makes the application easier to keep under control, more extensible, more maintainable, and, very important, easier to unit test.

The infrastructure for composition includes:

* The interface IConventionsRegistrar, which is the contract for registering composition conventions.
* The interface IConventionsBuilder, which is the contract for defining conventions using a fluent API.
* The interface ICompositionContainer, which is the contract for components providing composition hosting.
* The class CompositionContainerBuilderBase, which provides a base implementation for builders of composition containers.

How the composition infrastructure works:

1. All the convention registrars are collected (simply all the classes implementing IConventionRegistrar) and then they are invoked to register the conventions.
2. The composition container builder registers the log manager, the configuration manager, and the platform manager with factory export providers.
3. The composition container builder registers all application services [link] according to their metadata provided by the [AppServiceContract] attribute.
4. The composition container is built using the provided conventions.
5. And last, the composition container registers itself as the service exporting ICompositionContainer.

Recommendations:

* There is no restriction about the number of convention registrars per assembly nor what those registrars should register. However, to keep the things under control, a registrar should not register conventions for components outside the scope of the assembly where it is defined and, also, it is recommended to have one registrar per assembly.
* For components participating in composition, if possible, import the required services in the constructor. By using this approach it is clearly defined what is required for the component to function properly and also specific checks may be performed at the constructor level regarding imported services. However, if there are a lot of dependencies, the constructor may not be very appropriate due to an ugly signature, therefore in this case it is acceptable to use either property import or a combination of them.
* Prefer conventions over attributes. The code becomes clearer and more concise, and the dependencies on specific IoC containers will diminish.

See exposing application services [link] for more details about how to use in practice the composition.

## Alternative container implementation

Kephas provides a default composition container based on the portable MEF implementation. If a custom implementation is required for other IoC containers, please take care of the following:

* The composition container should export itself as a shared service for the ICompositionContainer contact, so that services requiring the composition container get this service injected. Accessing ambient services (like AmbientServices.Instance.CompositionContainer [link]) makes unit testing very hard.
* Use a composition container builder derived from the one provided as base, to have access to all the features it provides, including the registration of application services.

# Application services

Application services are services discovered and registered with the composition container at application level.

Steps for defining an application service:

1. Define the application service contract and configure it using the [AppServiceContract] attribute (Shared: yes/no, Allow multiple: yes/no).
2. Implement one or more application services based on the contract defined in the step above. Note: for contracts not allowing multiple service implementations, it is a recommended practice to decorate the service implementation with the [OverridePriority] attribute.
3. Consume the service.

Note: By default, the application services are shared. To change this default behavior, set the lifetime of the exported service in the [AppServiceContract] attribute.

Note: Kephas registers automatically application services with the composition container, so no composition registrars are required in this case.

Note: Because the sharing scope is defined at the service contract level, there is no need to set it at the implementation level. Otherwise it’s even counterproductive and confusing, making hard to identify possible bugs.

Note: in most cases the definition and discovery of application services should be sufficient, and no other conventions registrar for other kind of components should be required.

## Override priorities

An override priority is used for services not allowing multiple implementations at the same time, to ensure a deterministic identification of the desired service.

Kephas exposes its default services either with a lowest override priority (for example for null services), or with a low priority (the rest of them), to allow an uncomplicated override, because when an override priority is not provided, the normal value is used in this case.

## Multiple services with the same contract

If the application service contract should allow multiple registered service implementations, set the AllowMultiple option to true in the contract declaration.

Example:

[AppServiceContract(AllowMultiple = true)]

Note: generic application service contracts allow multiple registrations by default, because it is expected that multiple services will be defined with different actual generic type parameters.

## Composition metadata

Application services may indicate metadata attributes that they use. The following conventions are applied:

* The attributes must implement IMetadataValue<TValue>. The Value property will provide the value of the metadata key.
* The attribute type without the “Attribute” suffix will be the metadata key.
* When declaring the contract, the supported metadata attributes must be declared.
* The attributes are applied to the service implementations.

Example:

/// <summary>

/// Application service for request processing interception.

/// </summary>

/// <typeparam name="TRequest">The type of the request.</typeparam>

[AppServiceContract(AppServiceLifetime.Instance, AllowMultiple = true, MetadataAttributes = new[] { typeof(ProcessingPriorityAttribute) })]

public interface IRequestFilter<TRequest> : IRequestFilter

{

}

## Generic application service contracts

When exposing generic application service contracts, Kephas will try to find a non-generic application service contract in the implemented interfaces with the same full name. If one is found, it is considered that this non-generic contract is the actual service contract, and the generic one is provided for convenience and for collecting metadata.

Example:

/// <summary>

/// Application service for handling requests.

/// </summary>

public interface IRequestHandler

{

}

/// <summary>

/// Application service for handling requests.

/// </summary>

/// <typeparam name="TRequest">The type of the request.</typeparam>

/// <typeparam name="TResponse">The type of the response.</typeparam>

[AppServiceContract(AppServiceLifetime.Instance)]

public interface IRequestHandler<TRequest, TResponse> : IRequestHandler

{

}

In this example, the request handlers are registered exporting the non-generic IRequestHandler interface, so that all of them can be collected by the composition using the non-generic contract, and later on decisions may be taken based on the generic type metadata.

Additional to the metadata collected by using the MetadataAttributes declaration, Kephas collects also from the service implementations the actual generic types and adds them to the existing composition metadata. The following rules are applies:

* The actual generic parameter is the metadata value.
* The adjusted name of the generic parameter is the metadata key. The adjusted name is obtained by stripping the leading “T”, if specified, and appending “Type”, if not already there.