When developing MVC applications with Kentico, you may want to use some of the Kentico services. For example, services for usi ng shopping cart, tracking contacts, handling newsletter subscriptions, and more. The Kentico services are provided as NuGet int egration packages, which you can install in the MVC application. To initialize a service, you need to instantiate the service instance in the class where the service is used. We recommend instantiating Kentico services via dependency injection.

Tip: If you are not familiar with the dependency injection design pattern in general, we recommend that you learn about it first. For example, you can start with the following two articles:

- Dependency Injection and Inversion of Control with ASP.NET MVC by Mike Brind
- Dependency Injection by Steve Smith

Dependency injection design pattern

Dependency injection is a design pattern that is common to many programming languages and environments. Instead of instantiating a service instance inside a class constructor, you can pass the service instance into the constructor as a parameter. This way classes (e.g. controllers) do not need to obtain and configure the instance of a class they depend on.

See the following code snippet for a general example of the dependency injection design approach. Notice that the service is expected to be instantiated by the dependency injection container, no need to instantiate explicitly:

```
public class OrdersController : Controller
{
    private readonly IShoppingService mShoppingService;

    public OrdersController (IShoppingService shoppingService)
    {
        mShoppingService = shoppingService;
    }

        ...

}
```

Developing the MVC application with the dependency injection pattern in mind provides the following benefits:

- Less maintenance – changing object implementation is easier without strong coupling.
- Easier testing – when writing tests for MVC controllers, the controllers can be tested in isolation from the dependent objects.
- Loose coupling – classes are not directly dependent on one another.

Dependency injection containers

You can use a dependency injection container to instantiate interfaces of Kentico services. The purpose of dependency injection containers is to create a mapping between interfaces and concrete types. By using these containers, you reduce the complexity of instantiating various services and resolving the dependencies between them.

Some of the common dependency injection containers include:

- Autofac
- CastleWindsor
- Ninject
- Spring.NET
- StructureMap

For example, our sample Dancing Goat MVC project on GitHub uses the Autofac container to resolve dependencies.

Initializing dependency injection containers

If you want to use the dependency injection design pattern for development, we recommend the following approach:

When registering types from the *Kentico.** libraries in a dependency injection container, register all types implementing the **Kentico.Core.DependencyInjection.IService** or **Kentico.Core.DependencyInjection.IRepository** interface as necessary. No parameters need to be registered. The interfaces are included as part of the **Kentico.Core** NuGet package (a dependency of all Kentico packages containing service or repository implementations).

Similarly, when registering custom types in a dependency injection container, we recommend that all implement either the *Kentico.Core.DependencyInjection.IService* (if the component is a service) or *Kentico.Core.DependencyInjection.IRepository* (if the component is a repository) interface to simplify the registration process.

The following example demonstrates the registration of all services from the **Kentico.Search** assembly implementing the *Kentico.Core.DependencyInjection.IService* interface using Autofac:

```
private void ConfigureDependencyResolver()
{
        // Initializes the Autofac builder instance
        var builder = new ContainerBuilder();

        // Registers all types implementing the IService interface from an assembly
        builder.RegisterAssemblyTypes(
            typeof(SearchService).Assembly) // Identifies the 'Kentico.Search' assembly
            .Where(x => x.IsClass && !x.IsAbstract && typeof(IService).IsAssignableFrom
(x))
            .AsImplementedInterfaces()
            .InstancePerRequest();

        // Resolves the dependencies
    DependencyResolver.SetResolver(new AutofacDependencyResolver(builder.Build()));
}
```