Notice that the controller class depends on ProductRepository, and we are letting the controller create the ProductRepository instance. However, it's a bad idea to hard code the dependency in this way, for several reasons.

* If you want to replace ProductRepository with a different implementation, you also need to modify the controller class.
* If the ProductRepository has dependencies, you must configure these inside the controller. For a large project with multiple controllers, your configuration code becomes scattered across your project.
* It is hard to unit test, because the controller is hard-coded to query the database. For a unit test, you should use a mock or stub repository, which is not possible with the currect design.

We can address these problems by *injecting* the repository into the controller. First, refactor the ProductRepository class into an interface:

public interface IProductRepository

{

IEnumerable<Product> GetAll();

Product GetById(int id);

void Add(Product product);

}

public class ProductRepository : IProductRepository

{

// Implementation not shown.

}

Then provide the IProductRepository as a constructor parameter:

C#Copy

public class ProductsController : ApiController

{

private IProductRepository \_repository;

public ProductsController(IProductRepository repository)

{

\_repository = repository;

}

// Other controller methods not shown.

}

This example uses [constructor injection](http://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection). You can also use *setter injection*, where you set the dependency through a setter method or property.

But now there is a problem, because your application doesn't create the controller directly. Web API creates the controller when it routes the request, and Web API doesn't know anything about IProductRepository. This is where the Web API dependency resolver comes in.

**The Web API Dependency Resolver**

Web API defines the **IDependencyResolver** interface for resolving dependencies. Here is the definition of the interface:

C#Copy

public interface IDependencyResolver : IDependencyScope, IDisposable

{

IDependencyScope BeginScope();

}

public interface IDependencyScope : IDisposable

{

object GetService(Type serviceType);

IEnumerable<object> GetServices(Type serviceType);

}

The **IDependencyScope** interface has two methods:

* **GetService** creates one instance of a type.
* **GetServices** creates a collection of objects of a specified type.

The **IDependencyResolver** method inherits **IDependencyScope** and adds the **BeginScope** method. I'll talk about scopes later in this tutorial.

When Web API creates a controller instance, it first calls **IDependencyResolver.GetService**, passing in the controller type. You can use this extensibility hook to create the controller, resolving any dependencies. If **GetService** returns null, Web API looks for a parameterless constructor on the controller class.

**Dependency Resolution with the Unity Container**

Although you could write a complete **IDependencyResolver** implementation from scratch, the interface is really designed to act as bridge between Web API and existing IoC containers.

An IoC container is a software component that is responsible for managing dependencies. You register types with the container, and then use the container to create objects. The container automatically figures out the dependency relations. Many IoC containers also allow you to control things like object lifetime and scope.

**Note**

"IoC" stands for "inversion of control", which is a general pattern where a framework calls into application code. An IoC container constructs your objects for you, which "inverts" the usual flow of control.

For this tutorial, we'll use [Unity](https://msdn.microsoft.com/library/ff647202.aspx) from Microsoft Patterns & Practices. (Other popular libraries include [Castle Windsor](http://www.castleproject.org/), [Spring.Net](http://www.springframework.net/), [Autofac](https://code.google.com/p/autofac/), [Ninject](http://www.ninject.org/), and [StructureMap](http://structuremap.github.io/documentation/).) You can use NuGet Package Manager to install Unity. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, type the following command:

consoleCopy

Install-Package Unity

Here is an implementation of **IDependencyResolver** that wraps a Unity container.

C#Copy

using Microsoft.Practices.Unity;

using System;

using System.Collections.Generic;

using System.Web.Http.Dependencies;

public class UnityResolver : IDependencyResolver

{

protected IUnityContainer container;

public UnityResolver(IUnityContainer container)

{

if (container == null)

{

throw new ArgumentNullException("container");

}

this.container = container;

}

public object GetService(Type serviceType)

{

try

{

return container.Resolve(serviceType);

}

catch (ResolutionFailedException)

{

return null;

}

}

public IEnumerable<object> GetServices(Type serviceType)

{

try

{

return container.ResolveAll(serviceType);

}

catch (ResolutionFailedException)

{

return new List<object>();

}

}

public IDependencyScope BeginScope()

{

var child = container.CreateChildContainer();

return new UnityResolver(child);

}

public void Dispose()

{

Dispose(true);

}

protected virtual void Dispose(bool disposing)

{

container.Dispose();

}

}

**Note**

If the **GetService** method cannot resolve a type, it should return **null**. If the **GetServices** method cannot resolve a type, it should return an empty collection object. Don't throw exceptions for unknown types.

## Configuring the Dependency Resolver

Set the dependency resolver on the **DependencyResolver** property of the global **HttpConfiguration**object.

The following code registers the IProductRepository interface with Unity and then creates a UnityResolver.

C#Copy

public static void Register(HttpConfiguration config)

{

var container = new UnityContainer();

container.RegisterType<IProductRepository, ProductRepository>(new HierarchicalLifetimeManager());

config.DependencyResolver = new UnityResolver(container);

// Other Web API configuration not shown.

}

**After This add Nuget package for webApI unity and MVC.**

Install-Package Unity.AspNet.WebApi

Install-Package Unity.Mvc

Resolve the dependency in unityconfig as well as above

public static void RegisterTypes(IUnityContainer container)

{

// NOTE: To load from web.config uncomment the line below.

// Make sure to add a Unity.Configuration to the using statements.

// container.LoadConfiguration();

// TODO: Register your type's mappings here.

// container.RegisterType<IProductRepository, ProductRepository>();

container = new UnityContainer();

container.RegisterType<IShipRepository, ShipRepository>();

}

}