* Course Overview
* Introduction
  + Autofac is one of the more modern DI containers for .NET
  + MVC and Web API
* Course Agenda
  + DI Primer/App Demo
  + Autofac Integration
    - NuGet packages
  + Registering Components
* Course Agenda, Continued
  + Injection & Resolve
  + Advanced patterns
    - Decorators
    - Asp.net filters
  + Additional scenarios
    - Owin
    - Web forms
    - Client-side libraries
* Begin Autofac Primer
* Overview of Primer App in Ra..
* Introduce Abstractions
* Library Class Modification
  + Have the class implement the interfaces
  + Use constructor to inject services
* Hard-to-Maintain Instantiations
* Install Autofac and Registration
  + Install autofac nuget package
  + Autofac is a two step container building process
  + Create container builder
    - Ex) ContainerBuilder builder = new ContainerBuilder();
  + Then register services that need to resolve DI
    - Ex) builder.RegisterType<AvengerRepository>().As<IAvengerRepository>();
    - Everytime container request IAvengerRepository it will return an instance of Avenger Repository class
  + Non abstracted type must also be registered
  + Then build container
    - Ex) IContainer container = builder.Build();
* Resolving the Service
  + Ask container for service
    - Ex) var superheroService = container.Resolve<SuperheroService>();
* EasyBlog Intro
* EasyBlog Architecture
* Untestable Controllers
* Run Demo
* Overview
  + NuGet packages
    - Autofac
    - Autofac.Mvc5
    - Autofac.WebApi2
* Installing Nuget Packages
  + Install Autofac
  + Install Autofac.Mvc5
* Global.ASAX and ContainerBuilder
  + Create a container builder class
* Controller Registration and Bui…
  + Autofac requires any class to be resolved by container to be registered in container
  + Autofac.Mvc5 has extension method that does controller registration automatically
  + Ex)
    - builder.RegisterControllers(typeof(MvcApplication).Assembly);
  + Scans the assembly and look for all mvc controllers
* Setting the MVC Depedency…
  + MVC resolvers controller through Dependency resolver
  + Default dependency resolver merely instantiates the controller it needs
  + Install autofac dependency resolver in its place
  + Ex)
    - DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
* Web API Integration and Contr..
  + Install Autofac.WebApi2
  + Register all api controllers
  + Ex)
    - builder.RegisterApiControllers(typeof(MvcApplication).Assembly);
* Web API Dependency Resolver
  + Set web api dependency resolver
  + Ex)
    - GlobalConfiguration.Configuration.DependencyResolver = new AutofacWebApiDependencyResolver(container);
* OWIN Preview
  + OWIN requires a startup class
* Overview
  + Component lifetime
  + Assembly scanning
  + Autofac modules
* Component Lifetime
  + Lifetime are determined at component registration
  + Transient
    - Component life time is equated to that of its parent
    - New instance is provided to every controller and every service
  + Singleton
    - Component lifetime is equated to lifetime of container
    - Instantiated once and same instance reused on subsequent resolves
    - Same for every object and every request
  + Scoped
    - Component life time is not equated to anything and handled through manual means
    - Same within a request but different across different requests
* Registration Techniques
  + Procedural
    - One component registered at a time
    - Ex) RegisterType<T>().As<U>()
  + Assembly
    - Assemblies are scanned for multiple registrations
    - Ex) RegisterControllers(), RegisterApiControllers()
  + Modules
    - Component registrations offloaded to separate class
    - Can be logically grouped together
  + Configuration
    - Component registrations listed declaratively in config
    - Can be XML of JSON
    - Can be individual or module-based
* Lifetime in Controller Compon…
  + Will use transient lifetime by default
  + IDisposable interface
    - Autofac mvc persist container for us
    - Autofac do not release instance of IDisposable types until container is disposed
  + When inject and resolving Disposable classes with transient means new instance of class is being resolved every time controller is hit
    - Disposable classes is not released and could cause a memory leak
  + Scoped lifetime
    - Uses once instance across the same request
    - Ex) builder.RegisterControllers(typeof(MvcApplication).Assembly).InstancePerRequest();
* Singleton Test
* Register Extensibility Manager
* Resolve and Initialize the Mana…
* Registration Parameters
  + Can add parameter to registration what will be used in constructor of class being resolved
  + Can address construction argument by its name or type
  + Ex)
    - builder.RegisterType<BlogPostRepository>().As<IBlogPostRepository>().WithParameter(new TypedParameter(typeof(string), “easyBlog”));
  + ex)
    - builder.RegisterType<BlogPostRepository>().As<IBlogPostRepository>().WithParameter(new NamedParameter(“connectionStringName”, “easyBlog”));
* Assembly Scan and Register
  + Repository class name end in ‘Repository’
  + Repository interface starts with ‘I’
  + These conventions enable registering them easily in autofac
* Assembly Scan and Register, C…
  + Ex) builder.RegisterAssemblyTypes(typeof(BlogPostRepository).Assembly).Where(t => t.Name.EndsWith(“Repository”)).As(t => t.GetInterfaces()?.FirstOrDefault(i => i.Name == “I” + t.Name)).InstancePerRequest().WithParameter(new TypedParameter(typeof(string), “easyBlog”));
* Registration Modules
  + Can add registration into modules
  + Ex)
    - public class RepositoryRegistrationModule : Autofac.Module
    - {
    - protected override void Load(ContainerBuilder builder)
    - {

builder.RegisterAssemblyTypes(typeof(BlogPostRepository).Assembly).Where(t => t.Name.EndsWith(“Repository”)).As(t => t.GetInterfaces()?.FirstOrDefault(i => i.Name == “I” + t.Name)).InstancePerRequest().WithParameter(new TypedParameter(typeof(string), “easyBlog”));

* + - }
    - }
  + Then register module in Global.asax
  + Ex)
    - builder.RegisterModule<RepositoryRegistrationModule>();
* Create Configuration
  + Install nuget package Autofac.Configuration
  + Install Microsoft.Extension.Configuration.Json
  + Create json file that contains type definition for the autofac module we created earlier
    - Create json file ‘autofac.json’
    - Ex)
    - {
    - “defaultAssembly”: “EasyBlog.Web”,
    - “modules”: [
    - { “type”: “EasyBlog.Web.Core.RepositoryRegistrationModule” }
    - ],
    - “components”: [
    - “services”: [
    - {
    - “type”: “EasyBlog.Web.Core.IExtensibilityManager”,
    - “instanceScope”: “singleinstance”
    - }
    - ]
    - }
    - ]
    - }
* Load and Register Configuration
  + Register configuration in Global.asax
  + Ex)
    - IConfigurationBuilder config = new ConfigurationBuilder();
    - Config.AddJsonFile(“autofac.json”);
  + Ensure property of autofac.json is ‘copy is newer’ or ‘always copy’
  + Then create instance of Autofac configuration module class and register it
  + Ex)
    - ConfigurationModule module = new ConfigurationModule(config.Build());
    - Builder.RegisterModule(module);
  + More module and components can be added to json file without changing Global.asax
* Summary
  + Component Lifetime
    - Transient
    - Singleton
    - Scoped
  + Registration Parameters
  + Assembly Scanning w/Lambda
  + Registration Modules
  + Configuration
* Overview
  + Standard Injection
  + On-Demand Resolving
  + Injecting Into Views
* Standard Injection
  + constructor
    - Components injected as constructor arguments
    - Container will always be the one to resolver parent components
  + Property
    - Components injected as public properties
    - Controller wo8uld need to be registered with the trailing method PropertiesAutowired()
  + Container resolves dependency along with recursively resolving every dependency down its object graph
* On-demand Resolve
  + Resolve as needed
    - Not all methods require all dependencies
    - Resolve components when needed
  + “Creation” Component
    - Itself is registered and injected
    - Used to resolve other components
    - Based on abstract factory pattern
    - Variation of service locator
  + Service Locator Controversy
    - Some consider it an anti-pattern
    - Anti-pattern states a concrete locator embedded in your components
    - This version is abstracted and registered
    - It itself is injected as a dependency
    - Host component remains fully testable
* Injecting into Views
  + View obtains DependencyResolver
  + Resolves desired component
  + Used standard razor syntax
  + Bypasses controller-based injection
  + Technique works with any DI container
  + Not specific to Autofac
* Home Controller Modification
  + Remove hard instantiations and use dependency injection from controller constructor
  + Ex)
    - IBlogPostRepository \_BlogPostRepository;
    - public HomeController(IBlogPostRepository blogPostRepository)
    - {
    - \_BlogPostRepository = blogPostRepository;
    - }
* Property Injection Test
  + Ex)
    - public IBlogPostRepository BlogPostRepository { get; set;}
  + then in Global.asax
  + ex)
    - builder.RegisterControllers(typeof(MvcApplication).Assembly).InstancePerRequest().PropertiesAutowired();
  + constructor inject lends it self to be test creation
* Create Component Locator
  + Create class and interface
  + Ex)
    - public class ComponentLocator : IComponentLocator
    - {
    - ILifetimeScope \_container;
    - public ComponentLocator(ILifetimeScope container)
    - {
    - \_container = container;
    - }
    - public T ResolveComponent<T>()
    - {
    - return \_Container.Resolve<T>();
    - }
    - }
    - public interface IComponentLocator
    - {
    - T ResolveComponent<T>();
    - }
* Inject Container into Compone…
  + Autofac feature, container itself is abstracted to interface ILifeTimeScope and automatically registered
* Inject into API Controller and ..
  + Ex)
    - IComponentLocator \_ComponentLocator;
    - public BlogApiController(IComponentLocator componentLocator)
    - {
    - \_ComponentLocator = componentLocator;
    - …
    - }
  + Then ask \_ComponentLocator to resolve instance of blog post respoitory
  + Ex)
    - IBlogPostRepository blogPostRepository = \_ComponentLocator.ResolveComponent<IBlogPostRepository>();
* Finish Modifications and Regist..
  + Add Component locator to json file
  + Ex)
    - ….
    - {
    - “type”: “EasyBlog.Web.Core.ComponentLocator”,
    - “services”: [
    - {
    - “type”: “EasyBlog.Web.Core.IComponentLocator”
    - }
    - ]
    - }
    - …
* Injecting Extensibility Manager
* Create and Register Local Strin…
  + Create class to hold local strings
  + Abstract class to interface to inject into views
  + Register it in json
* Inject into Layout View
  + Ex)
    - <!DOCTYPE html>
    - @{
    - var localStrings = DependencyResolver.Current.GetService<EasyBlog.Web.Core.ILocalStrings>();
    - }
* Inject into Other Views
* Summary
  + Couple of ways to inject into controllers
  + Container-agnostic way to inject into views
* Overview
  + Decorators
  + ASP.NET Filters
* Decorators
  + You may have constructs whose code you don’t control
  + Existing components where changing would break applications
  + Not all constructs are DI-enabled
  + Examples
    - ConfigurationManager
    - Activator methods
    - Static classes
    - No interface abstraction
    - Locked into single implementation
  + Solution
    - Provide a wrapper for static construct or “factory” to provide information
    - Configuration classes
    - Object instance
    - Abstract factory to interface
* Extensibility Manager Before-s..
* Configuration Factor Constru..
  + Create IConfigurationFactory interface
  + Ex)
    - public interface IConfigurationFactory
    - {
    - EasyBlogModulesConfigurationElementCollection GetModules();
    - }
  + Create ConfigurationFactory class to implement interface
  + Ex)
    - public class ConfigurationFactory : IConfigurationFactory
    - {
    - EasyBlogConfigurationSection config = ConfigurationManager.GetSection(“easyBlog”) as EasyBlogConfigurationSection;
    - if (config != null)
    - return config.Modules;
    - else
    - return null;
* Inject and Use Configuration F..
  + Inject configuration factory into ExtensibilityManager
  + Then register ConfigurationFactory in json file
* Replace Activator.CreateInstan…
  + Inject container into ExtensibilityManager
    - Of instance ILifetimeScope
    - Ex)
    - public ExtensibilityManager(IConfigurationFactory configurationFactory, ILifetimeScope container) {}
  + ILifetimeScope is the autofac container itself, registers on its own when you builder a container
* Wildcard Concrete Class Regis..
  + Autofac lets us automatically register non interface associated classes in a wildcard fashion
  + Create autofac module to do so
  + Ex)
    - public class ExtensibilityRegistrations: Module
    - {
    - protected override void Load(ContainerBuilder builder)
    - {
    - builder.RegisterSource(new AnyConcreteTypeNotAlreadyRegisteredSource(t => {return t.GetInterfaces().FirstOrDefault(i => i.Name == “IEasyBlogModule”) != null;}));
    - }
    - }
  + Register module in json file
* Filters
  + Attributes you can place on controllers
  + MVC Filters
    - Inherit from System.Web.Mvc.ActionFilterAttribute
    - Override OnActionExecuting and/or OnActionExecuted
    - For autofac integration: builder.RegisterFilterProvider() Use property injection
  + Web API Filters
    - Inherit from System.Web.Http.Filterse.ActionFilterAttribute
    - Override OnActionExecuting and/or OnActionExecuted
    - For autofac integration: builder.RegisterWebApiFilterProvider() Use property injection
* Logger Class and interface
* LogMvcAction Filter
  + Add class that inherit from Attribute class with Attribute sufix
  + Ex) ActionFilterAttribute from System.Web.Mvc
    - public class LogMvcActionAttribute : ActionFilterAttribute
    - {
    - public ILogger \_Logger { private get; set; }
    - public override void OnActionExecuted(ActionExecutedContext filterContext)  
       {
    - base.OnActionExecuted(filterContext);
    - \_Logger.Log(filterContext.ActionDescriptor.ActionName);
    - }
    - }
* Register MVC Filter and Test
  + In Global.asax
  + Ex)
    - builder.RegisterFilterProvider();
  + RegisterFIlterProvider() will handle the registration of any MVC filter we have
    - Handles the resolve of filters that decorate actions and controllers
    - Ensure resolve will resolve public property dependencies as well
  + Then use filter in controllers]
  + Ex)
    - [LogMvcAction]
    - public ActionResult Index() {]
* LogWebApiAction Filter
  + create class
  + ex)ActionFilterAttribute from System.Web.Http.Filters
    - public class LogWebApiAction : ActionFilterAttribute
    - {
    - public override void OnExecuted(HttpActionExecutedContext actionExecuted)
    - {
    - base.OnActionExecuted(actionExecutedContext);
    - }
    - }
* Register API Filter and Test
  + in global.asax
    - builder.RegisterWebApiFilterProvider(GlobalConfiguration.Configuration);
* Global Filters
  + global filters works on all controllers without having to place attribute on individual controllers and actions
  + add to global mvc filters collection
  + in global.asax
    - ex) GlobalFilters.Filters.Add(container.Resolve<LogMvcActionAttributre>()));
  + RegisterFilterProvider(), ReigsterWebApiFilterProvider() only for filters used in an attribute capacity
  + need to register global filter the manual way
    - builder.RegisterType<LogMvcActionAttribute>().PropertiesAutowired();
  + to add global api filters
  + in global.asax
    - GlobalConfiguration.Configuration.Filterse.Add(container.Resolve<LogWebApiAction>());
  + register filter
    - builder.RegisterType<LogWebApiAction>().ProperiesAutowired();
* Summary
  + Beyond core Autofac usage
  + Decorators can have many flavors
  + Filters are not required but useful
* Overview
  + Autofac in owin hosted services instead of IIS
  + autofac in ASP.NET web forms
  + Client-side Libraries
* Owin
  + preferred by many as for hosting services
  + decoupled from ASP.NET
  + handles its own pipeline
  + introduces middleware constructs
  + served as the basis for ASP Core
  + owin offers injection points by way of middleware
  + not on the ASP.NET pipeline
  + not served by IIS
  + injection into pipeline performed by middleware components
  + uses Startup class
  + can host Web API Services
  + same controllers, filters, routing
  + startup class is similar to Global.asax
  + Autofac.WebApi2.Owin
  + Autofac.Owin
  + registration options are same as before
  + resolving is as easy as before
* Owin Project Overview
  + Startup.cs
    - creates its own HttpConfiguration class instead of relying on dot net to create one for us
    - add routes
    - install middleware components
    - controllers are the same
* Add Autofac Packages
  + autofac.webapi2.owin
    - adds dependencies for us
  + update Autofac and Autofac.WebApi2
* Register Components and Buil…
  + create container builder
  + ex)
    - ContainerBuilder builder = new ContainerBuilder();
  + register repository and other services
  + ex)
    - builder.RegisterType<AvengerRepository>().As<IAvengerRepository>().InstancePerRequest();
    - builder.RegisterType<Logger>().As<ILogger>().InstancePerRequest();
  + register middleware
  + ex)
    - builder.RegisterType<LoggerMiddleware>().InstancePerRequest();
  + register controller
  + ex)
    - builder.RegisterApiControllers(Assembly.GetExecutingAssembly()).InstancePerRequest();
  + build container and set it in dependency resolver
  + ex)
    - IContainer container = builder.Build();
  + set dependency resolver
  + ex)
    - config.DependencyResolver = new AutofacWebApiDependencyResolver(container);
* Set up “Use” Statements
  + need to turn on autofac in owin
  + ex)
    - appBuilder.UseAutofacWebApi(config).UseAutofacMiddleware(container).UseWebApi(config);
  + UseAutofacMiddleware()
    - automatically install middleware components it recognizes by looking in container
* Middleware Order Control
  + To control order of middle
  + Ex)
    - appBuilder
    - .UseAutofacWebApi(config)
    - .UseAutofacLifetimeScopeInjector(container)
    - .UseMiddlewareFromContainer<LoggerMiddleware>()
    - .UseWebApi(config);
  + UseAutofacLifetimeScopeInjector(container) has to be first the first middleware component installed
  + Then make call UseMiddlewareFromContainer() for each middleware component you want hookup
  + Then make modifications to controller class to use dependency injection
* Adjust Controller Class
* Adjust Middleware Component
* Test Run and Examine Results
* Web Forms
  + Part of ASP.NET
  + Directly supported by Autofac
  + Can inject unlimited levels of components into Web Form Code-Behind classes
  + Web forms need love too
  + Web Forms integration package
  + Installation of two web server modules in config file
  + Minor modification of global class
  + Building the container in your registration technique of choice
* Install Autofac Packages
  + Install nuget package ‘Autoface.web’
  + In global.asax you need to implement interface from the package, IContainerProviderAccessor
  + Ex)
    - public class Global : HttpApplication, IContainerProviderAccessor
    - {
    - static IContainerProvider \_ContainerProvider;
    - public IContainerProvider ContainerProvider
    - {
    - get { return \_ContainerProvider; }
    - }
    - void Application\_Start(object sender, EventArgs e)
    - {
    - RouteConfig.RegisterRoutes(RouteTable.Routes);
    - BundleConfig.RegisterBundles(BundleTable.Bundles);
    - }
    - }
* Setting up Container and Registrat..
  + Ex)
    - void Application\_Start(object sender, EventArgs e)
    - {
    - RouteConfig.RegisterRoutes(RouteTable.Routes);
    - BundleConfig.RegisterBundles(BundleTable.Bundles);
    - ContainerBuilder builder = new ContainerBuilder();
    - builder.ReigsterType<AvengerRepository>().As<IAvengerRepository>().InstancePerRequest();
    - builder.RegisterType<Logger>().As<ILogger>().InstancePerRequest();
    - builder.RegisterType<SuperheroService>().As<ISuperheroService>().InstancePerRequest();
    - IContainer container = builder.Build();
    - \_ContainerProvider = new ContainerProvider(container);
    - }
* Modify Web.Config
  + Web forms uses code behind class behind each class that must have a default constructor, we cannot use constructor injection
  + Can use property injection
  + In Web.Config under <Configuration></Configuration>
    - Ex)
    - <system.webServer>
    - <modules>
    - <add name=”PropertyInjection” type=”Autofac.Integration.Web.Forms.PropertyInjectionModule, Autofac.Integration.Web” precondition=”managedHandler” />
    - <add name=”ContainerDisposal” type=”Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web” precondition=”managedHandler” />
    - </modules>
    - <system.webServer>
* Modify Code-behind Classes
  + Use property injection
* Demo Run
* Client-side Libraries Mention a…
  + ASP.NET
    - Relied on a third party DI container
    - Autofac
    - Additional integration packages simplified usage
    - Registration handled by autofac
    - Injection in standard c# syntax
    - Resolve handled by autofac
  + JS-based client libs
    - Typically built into JS library or TS in the case of angular
    - Each library has its proprietary flavor of DI
    - Some general-purpose products(Kontainer) exist
    - No autofac for js
    - No ASP.NET involved here