Microsoft | Services

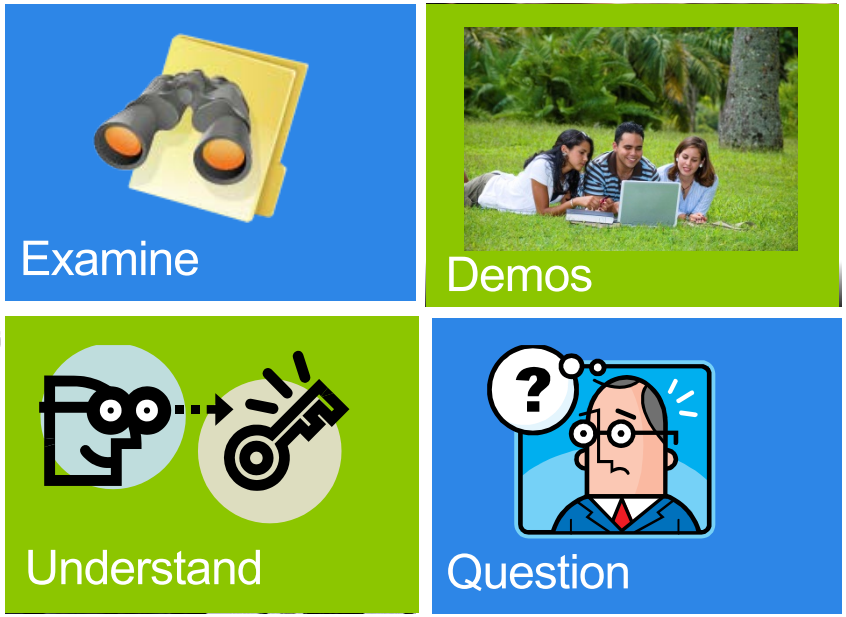
Tight-Coupling and Dependency Injection

STRATEGY ■ CONSULTING ■ **SUPPORT**

Rob Vettor, Application Developer Consultant
Microsoft Premier Support Services

Why we are here?

- Tight Coupling
- Inversion of Control
- Service Locator
- Dependency Injection
- Dependency Resolver



Key Takeaways

Understand *Dependency Injection* and its benefits

What's Wrong with this Component?

```
public class StoreManagerController : Controller
{
        private AlbumService _albumService;
        private ArtistService _artistService;
        private GenreService _genreService;

        public StoreManagerController()
        {
                _albumService = new AlbumService();
                _genreService = new GenreService();
                _artistService = new ArtistService();
        }
```

Fact: Takes 3 dependencies

Concern: Explicitly creates dependencies

Concern: Communicates directly with concrete classes

Relationship said to be "Tightly Coupled"

Tight Coupling

- Component contains *hard-coded* references to dependent classes…
  - Resistant to change
    - Changing dependent class can affect component
    - Requires recompiling and testing component
  - Hard to unit test
    - Difficult to quickly *swap-out* mock classes for testing
- Considered *liability* in your design…
  - Accuring technical debt

Reducing Responsibility

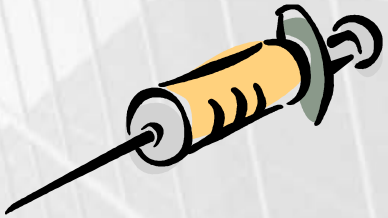
- Shift responsibility for instantiating dependent classes from component to consuming code

```
public class StoreManagerController : Controller
{
        private readonly AlbumService _albumService;
        private readonly ArtistService _artistService;
        private readonly GenreService _genreService;

        public StoreManagerController(AlbumService albumService,
                                      GenreService genreService,
                                      ArtistService albumService)
        {
                _albumService = albumService;
                _genreService = genreService;
                _artistService = artistService;
        }
```

Receives dependencies via constructor (constructor injection)

Constructor Injection



- *Inverts control*
  - Instantiation moved outside
  - Relieves component from responsibility of creating dependency classes
- *Two problems* still persist:
  - Component still tightly-coupled to concrete classes
  - Calling class now becomes more complex (must instantiate and inject dependent classes)

Reducing Coupling

- Add *abstraction layer* between component and dependencies by implementing *interfaces*

```
public class StoreManagerController : Controller
{
        private readonly IAlbumService _albumService;
        private readlonly IArtistService _artistService;
        private readonly IGenreService _genreService;

        public StoreManagerController(IAlbumService albumService,
                                      IGenreService genreService,
                                      IArtistService albumService)
        {
                _albumService = albumService;
                _genreService = genreService;
                _artistService = artistService;
        }

        }
```

Interface references

Communicates with concrete class via interface

Refactoring to Interfaces

- Key Benefits
  - Component dependent on interfaces…
    - Not aware of underlying concrete class
    - Communicates directly with interface
    - Can talk to any class that implements interface
  - Can inject different implementations of dependent class without modifying component
    - Flexibility/Interchangeability
    - Pass fake implementations for unit testing

We Have Improved…

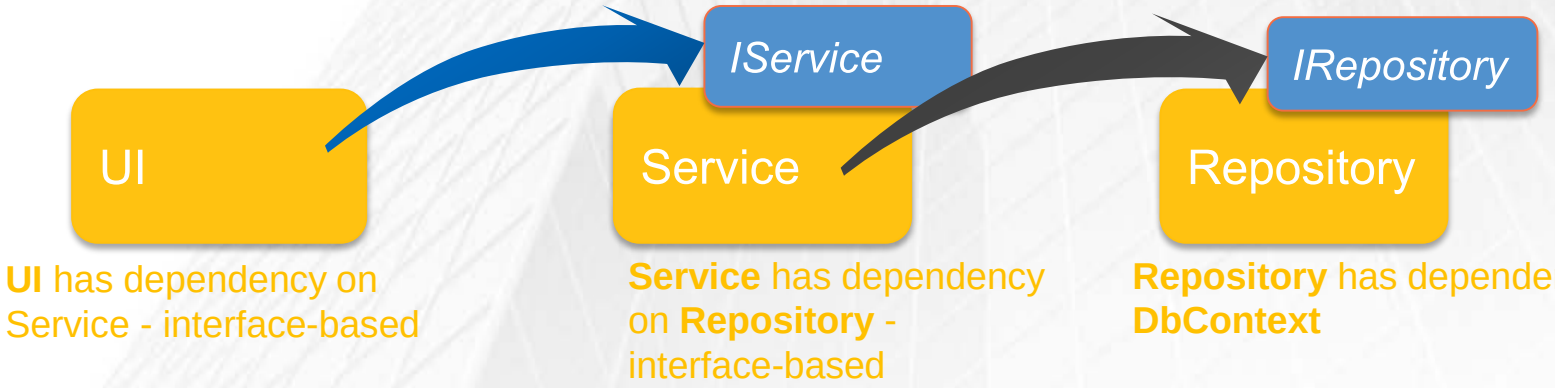
Tight-Coupling

to…

Loose-Coupling

Loose Coupling with Interfaces

- *Enforce loose coupling* by implementing Interfaces to communicate across each layer --
- A layer should not expose internal details on which another layer could depend
- UI, Services, Repositories all loosely coupled with interface referencs



- One Layer can Mock or Fake another layer to isolate functionality, enabling testability across each layer

Demo

Adding interfaces for communicating across layers

Design Still Has Problems…

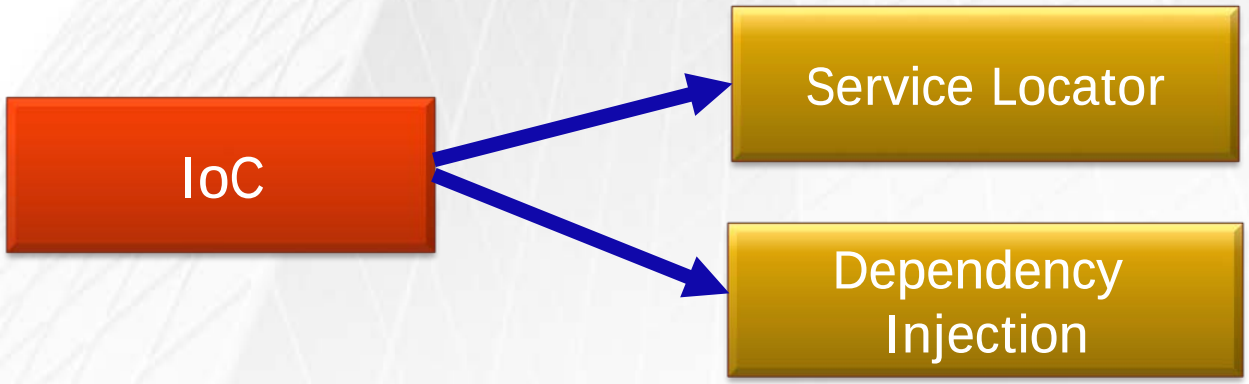
- *Calling code* too *complex*…
  - Now responsible for selecting, instantiating and injecting dependencies into component

Inversion of Control (*IoC*)

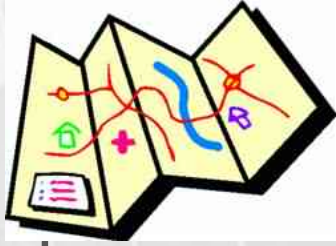
- Widely-accepted *design principle*
- Promotes *loose coupling* between objects by *inverting* the control flow of an application
- Dependent object *receives* its dependencies from a *container*
- Improves Unit testing, maintainability, extensibility and interchangeability

IoC Implementations

- Two common approaches:
  - Service Locator
  - Dependency Injection
- Both expose *centralized container* to manage dependent objects
- Removes *tight-coupling* between dependent objects…



Service Locator



- Adopts *pull model*, using registry class to create objects on your behalf
- *On Demand*
  - Component responsible for querying locator via key or interface type
- Similar to Factory Pattern, but Locater…
  - Responsible for managing lifetime of dependent object (maintains instance)
  - Capable of returning reference to any object, while Factories typically return a specific family of types
- Component takes dependency on Locator

Demo

Service Locator Pattern

Dependency Injection

- Adopts *push model*-
  - Container responsible for *Injecting* concrete objects at runtime
- Component *passive*
  - Not concerned with creating dependencies
  - *Receives* from outside
- Component has no dependency on container injecting dependent objects
- Benefits…
  - Increases maintainability/extensibility
  - Swap-out components without major refactoring
  - Develop components independently of each other

Demo

Poor-Man's Dependency Injection

DI Containers

- Primary responsibility to manage object instances
  - Maps concrete types to interface references
  - Instantiate and inject dependent objects into component
  - Think – *Object Composition*
- Several DI containers available for .NET
  - Autofac
  - Castle Windsor
  - Ninject
  - Spring.NET
  - Structure Map
  - Unity
- Similar functionality - differ in API design

What do DI Containers Do?

- Manage the Dependency Injection Lifecycle

Register

- Perform at start-up
- Register mappings that specify how…
  - Concrete objects will be injected
  - Object graphs will be constructed
- Typically, three ways to perform registration…
  - Programmatically (easy, but requires recompiles)
  - Design-Time (from configuration file)
  - Auto-Registration (use runner to scan assemblies and register objects based on predefined rules)

Types of Registration

- RegisterType
  - Resolve concrete class to interface or base class
- Register Instance
  - Instantiate type at runtime is treated as singleton
- Named type registration
  - Register multiple concrete classes with single interface
  - Used named label to select desired class
- InjectionConstructor Class (Line 90)
  - Configure Object Graph for component
- Open Generics
- Parameter overrides

Demo

Demonstrate…

- RegisterInstance
- RegisterType
- Named Type Registration
- InjectionContstructor Class

Object Lifetime

- Lifetime Managers manage the lifetime of objects instantiated by the container…
  - Transient Lifetime (default)
    - Creates new instance of object each for each resolve
  - Container Controlled Lifetime
    - Creates singleton instance upon demand
  - Per Request Lifetime
    - Creates instance for duration of a request
  - Per Thread Lifetime
    - Implements singleton behavior on a per-thread bases

Demo

Registration - Mapping the DI Container

Resolution

- Constructor Injection
- Explicit Resolve
- Property Injection
  - Attribute property to engage setter injection

```
[Dependency]
public SomeOtherObject DependentObject
{
    get { return _dependentObject; }
    set { _dependentObject = value; }
}
```

- Method Injection

```
[InjectionMethod]
public void Initialize(SomeOtherObject dep)
{
    _dependentObject = dep;
}
```

Dependency Resolver

- Abstracts/decouples application from specific DI container
- Framework communicates with DI container through the *IDependencyResolver* interface

```
public interface IDependencyResolver
{
        object GetService(Type serviceType);
        IEnumberable<object> GetServices(Type serviceType)
}
```

- Implement resolver and register DI Container in the Global.asax
- Can register any class that app needs to consume with DI container
- MVC first consults the resolver when it needs a class instance

Demo

Building the Dependency Resolver

Questions? Comments?