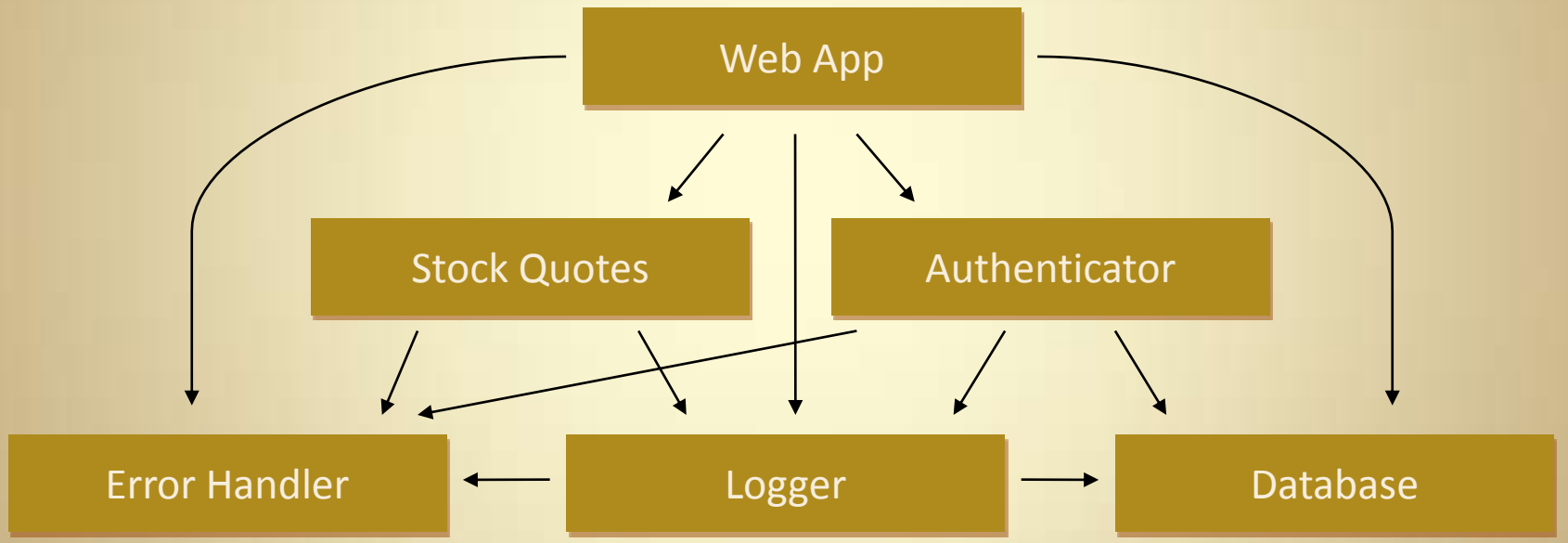
Dependency Injection and the Unity Container

Brad Wilson
Microsoft Corporation

The Path To Dependency Injection

Simple Example

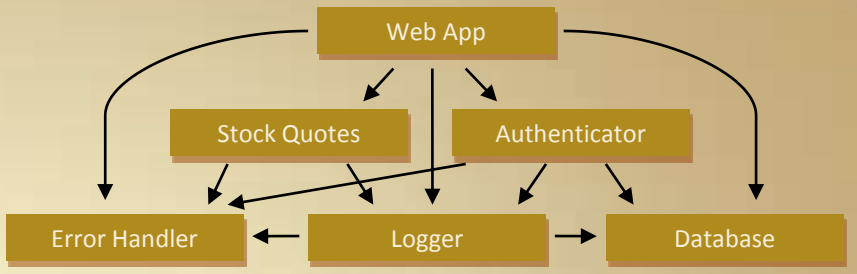


This example was originally created by Jim Weirich at http://shrinkster.com/dcj

Stage 0

"I Don't Need Help!"

Example




```
public class WebApp
{
  readonly Authenticator authenticator;
  readonly Database      database;
  readonly ErrorHandler  errorHandler;
  readonly Logger        logger;
  readonly StockQuotes   stockQuotes;

  public WebApp()
  {
    authenticator = new Authenticator();
    database      = new Database();
    errorHandler  = new ErrorHandler();
    logger        = new Logger();
    stockQuotes   = new StockQuotes();
  }
}
```

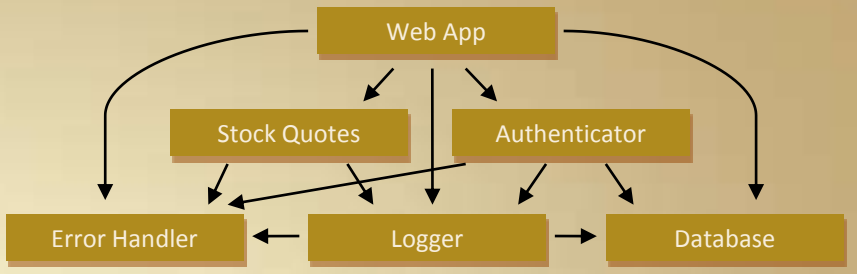
Observations

- Everybody creates what they need
- Everybody has to know *exactly* what to create
- No ordering issues (create things in any order)
- How do we share instances? (singletons)
- How do we test our components in isolation?

Stage 1

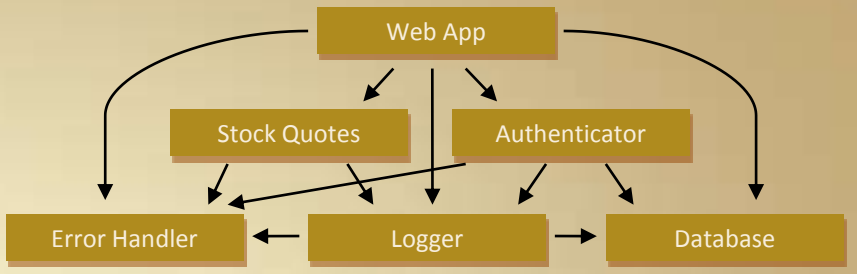
Hard-Wired Service Locator

Example



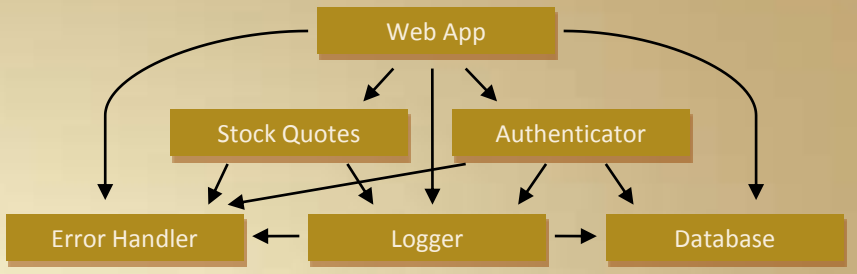

```
public interface IServiceLocator
{
   Authenticator Authenticator { get; }
   Database      Database      { get; }
   ErrorHandler  ErrorHandler  { get; }
   Logger        Logger        { get; }
   StockQuotes   StockQuotes   { get; }

}
```

Example




```
public class SingletonServiceLocator
             : IServiceLocator
{
  public SingletonServiceLocator()
  {
    ErrorHandler   = new ErrorHandler();
    Database       = new Database();
    Logger         = new Logger(this);
    Authenticator  = new Authenticator(this);
    StockQuotes    = new StockQuotes(this);
  }

  public Authenticator Authenticator { get; private set; }
  public Database      Database      { get; private set; }
  public ErrorHandler  ErrorHandler  { get; private set; }
  public Logger        Logger        { get; private set; }
  public StockQuotes   StockQuotes   { get; private set; }
}
```

Example

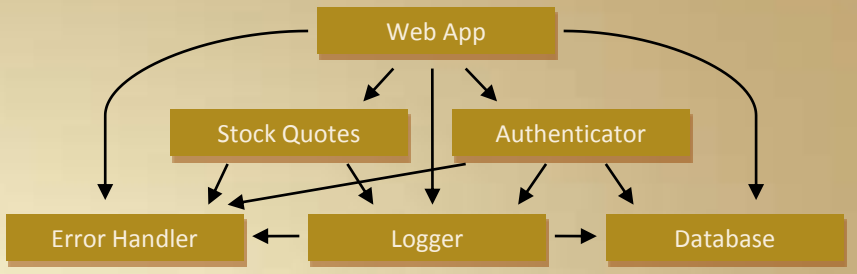



```
public class Authenticator
{
    readonly ErrorHandler errorHandler;
    readonly Logger       logger;
    readonly Database     database;

    public Authenticator(IServiceLocator locator)
    {
        errorHandler = locator.ErrorHandler;
        logger       = locator.Logger;
        database     = locator.Database;
    }
}
```

Observations

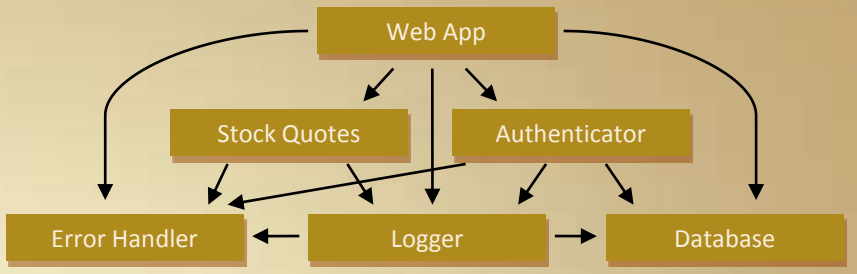
- Locator is very specific to our needs
- Locator makes the policy: singleton or not
- One policy for everybody
- Introduced a creation order issue
- Have an “opaque requirements” issue
- Objects are created whether they’re needed or not
- We can create our a special locator for testing
  - Requires us to use interfaces or abstract/virtual classes

Example




```
public class DelayCreationSingletonServiceLocator
             : IServiceLocator
{
  IAuthenticator authenticator;
  IDatabase      database;
  IErrorHandler  errorHandler;
  ILogger        logger;
  IStockQuotes   stockQuotes;

  public IAuthenticator Authenticator
  {
    get
    {
      if (authenticator == null)
        authenticator = new Authenticator(this);
      return authenticator;
    }
  }

  [...]
}
```

Example




```
public class Authenticator : IAuthenticator
{
    readonly IErrorHandler errorHandler;
    readonly ILogger       logger;
    readonly IDatabase     database;

    public Authenticator(IServiceLocator locator)
    {
        errorHandler = locator.ErrorHandler;
        logger       = locator.Logger;
        database     = locator.Database;
    }
}
```

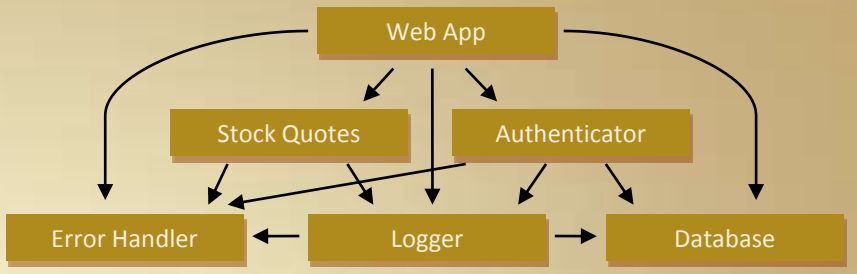
Observations

- We've solved the ordering problem
- We've solved the over-creation problem
- Still one policy for everybody
- Still have opaque requirements
- Still have a locator that's specific to our needs

Stage 2

Generic Service Locator

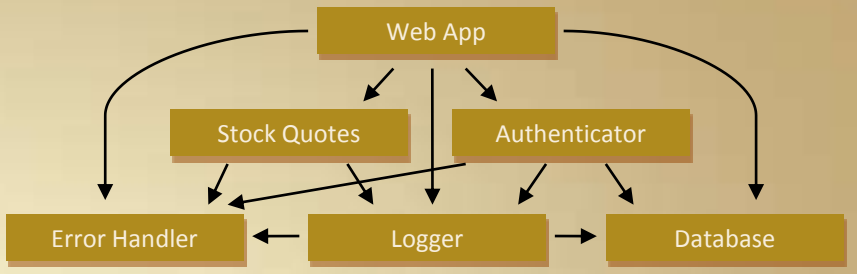
Example




```
public interface IServiceLocator
{
  TService Get<TService>();
}
```

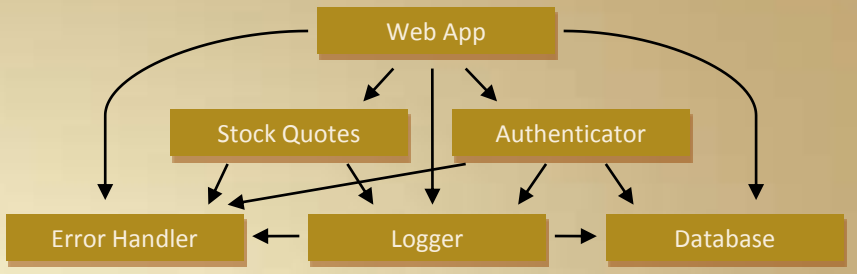
```
public class Authenticator : IAuthenticator
{
  readonly IDatabase      database;
  readonly IErrorHandler errorHandler;
  readonly ILogger        logger;

  public Authenticator(IServiceLocator locator)
  {
    errorHandler = locator.Get<IErrorHandler>();
    logger       = locator.Get<ILogger>();
    database     = locator.Get<IDatabase>();
  }
}
```

Example




```
public class ServiceLocator : IServiceLocator
{
  readonly Dictionary<Type, object> services;

  public TService Get<TService>()
  {
    object result;

    if (services.TryGetValue(typeof(TService), out result))
      return (TService)result;

    throw new ArgumentException("Unknown service type " +
                                typeof(TService).FullName);
  }

  public void Register<TService>(TService serviceInstance)
  {
    services[typeof(TService)] = serviceInstance;
  }
}
```

Example




```
ServiceLocator locator = new ServiceLocator();

locator.Register<IErrorHandler> (new ErrorHandler());
locator.Register<IDatabase>     (new Database());
locator.Register<ILogger>       (new Logger(locator));
locator.Register<IAuthenticator>(new Authenticator(locator));
locator.Register<IStockQuotes>  (new StockQuotes(locator));
```

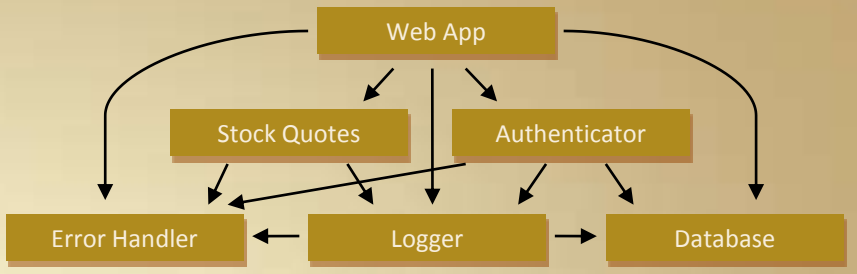
Observations

- Locator has become general purpose
- Don't need separate testing version(s) of locator
- One definitive answer per type
- It's all about types – no primitives
- Object creation cannot be done by the locator
- Back to having creation ordering issues again
- Forced into singletons (no more policy choice)
- Requirements have become even more opaque
- No way to generally know what's inside
- Objects are tightly coupled to a locator

Stage 3

Dependency Injection Container

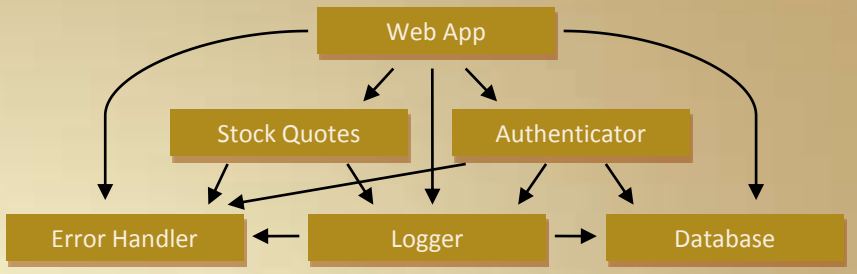
Example




```
public interface IDependencyContainer
{
    TService Get<TService>();
    void Map<TFrom, TTo>() where TTo : TFrom;
}
```

```
public class Authenticator : IAuthenticator
{
  readonly IDatabase      database;
  readonly IErrorHandler errorHandler;
  readonly ILogger        logger;

  public Authenticator(IDatabase database,
      IErrorHandler errorHandler, ILogger logger)
  {
    this.database      = database;
    this.errorHandler = errorHandler;
    this.logger        = logger;
  }
}
```

Example




```
DependencyContainer container = new DependencyContainer();

container.Map<IAuthenticator, Authenticator>();
container.Map<IDatabase, Database>();
container.Map<IErrorHandler, ErrorHandler>();
container.Map<ILogger, Logger>();
container.Map<IStockQuotes, StockQuotes>();
```

Observations

- Component requirements are clear
  - This form of dependency injection is “constructor injection”
- Objects don’t need or know about containers
- Creation order problem is gone
- Objects are created as necessary
- Container controls object policies

Considerations

- Type of resolution
  - Resolve by type?
  - Resolve by name?
  - Resolve a single instance vs. several?
- Types of injection
  - Constructor injection?
  - Property Setter injection
  - Method Call injection
- Lifetime policies
  - New instance?
  - Singleton per AppDomain?
  - Singleton per thread?

Considerations

- Method interception
  - Virtual method interception?
  - Interface interception?
  - MarshalByRefObject interception?
- Special behaviors
  - Behavior when resolution fails?
  - Event brokering?
  - Parent/child container relationships?
- Configuring the container
  - Hand-written code?
  - XML and/or App.config?
  - Attributes?

Unity Container Demo

Resources

Unity Container
http://www.codeplex.com/unity

Patterns & Practices
http://msdn.microsoft.com/practices

Me ☺
bradwils@microsoft.com
http://bradwilson.typepad.com