**Accessing the database via KLAS code**

In our MVC Controllers we are leveraging “Dependency Injection” to handle our accessing of the database. Specifically we are using “[Ninject](http://www.ninject.org/) Dependency Injection framework” an IOC container. To learn more about Dependency Injection see <http://bobcravens.com/2010/03/dependency-injection-and-inversion-of-control-containers/> and <http://blog.agilistic.nl/a-step-by-step-guide-to-using-ninject-for-dependancy-injection-in-c-sharp/> . Also see <http://www.ninject.org/>

IOC containers add abstraction around the creation / life-cycle management process. This abstraction requires a certain level of understanding for a developer working on the code. Developers are much more used to using ‘new’ to create objects than wiring up the IOC container. Once this barrier to entry has been overcome, then clarity is not really an issue.

In our MVC Controllers you will notice the “[Inject]” attribute, followed by a property:

[Inject]

public EmployeeController EmployeeController { get; set; } – we don’t seem to ever get or set this property within the MVC Controller, so perhaps declaring it as a property is required by Ninject????

“EmployeeController” class belongs to our Domain.Controllers namespace, and it contains a property of a repository Interface type.

IEmployeeRepository \_employeeRepository (Declaring the prop as an Interface type seems to be related to Dependency Injection). In comparison the C:\Code\KLAS\Klas.Domain\Controllers\EcosystemController.cs declares \_ecosystemRepository as readOnly without the get; set; (so it’s a private variable and not a property). This variable (or property, as the case may be) is set in the constructor.

The “IEmployeeRepository” interface is implemented by the “EmployeeRepository” class which belongs to the Domain.Repositories namespace.

The Domain.Controller (i.e EmployeeController class) has methods that access the database, through the “IEmployeeRepository” variable (property)

public void AddEmployee(string firstName, string lastName, string userName)

{

\_employeeRepository.AddEmployee(firstName, lastName, userName);

}

\_employeeRepository.AddEmployee first defined (without any functionality or logic) in public interface IEmployeeRepository. The public class EmployeeRepository implements the interface and defines the functionality (logic) of AddEmployee.

class EmployeeRepository declares private KlasEntities \_dbContext; KlasEntities is an auto-generated class (file) from the EDMX. \_dbContext is set in the Constructor of EmployeeRepository

Within the Klas.Domain project there is a Ninject folder with a file (class) named KlasDependencies which creates a concrete implementation of the ‘NinjectModule’ abstract class. In there are many Bind statements for example: Bind<IEmployeeRepository>().To<EmployeeRepository>().InRequestScope();

The generic Ninject ‘Bind’ method takes the type of the service (‘IEmployeeRepository for instance) to bind. The Ninject ‘To’ method takes the implementation type (‘EmployeeRepository’). This wires up the Ninject container to provide an instance of ‘EmployeeRepository’ for every ‘IEmployeeRepository request. The reason for using InRequestScope () is to make sure that a single instance of an object is shared by all objects created via the Ninject kernel for that HTTP request (e.g. to share an object that is expensive to create).

**And as if by magic The declaration in the MVC controller gives access to the database!!!**

**Jan 12, 2016- I created a test project to explore Ninject and Dependency Injection. See Zip DependencyInjection.zip (C:\websites\Sites\Docs).**

**Here is what I learned:**

Dependency Injection

In software engineering, dependency injection is a software design pattern that implements inversion of control for resolving dependencies. A dependency is any object that another object requires. An injection is the passing of a dependency to a dependent object that needs/uses (consumes) it. The purpose of is to reduce hard-coded dependancies between classes and in so doing introduce flexiblity. Dependencies make unit testing (nearly) impossible and makes the changing of a concrete implementation sloppy and difficult. The bottom-line is that your consuming object (class) doesn't need to know what concrete implementation of the consumed object is used. What concrete implementation is used, should be determined outside of your classes. That way, you can swap out the consumed object with another implementation if the need arises (in testing the need often arises).   
  
In this example we explore both Setter Injection and Constructor Injection.   
  
**Constructor Injection**

* Constructor injection is done by defining a Constructor in our MVC Controller Class that expects parameters (normally MVC Controller Classes don't have constructors).
* For instance in this example we define public DefaultController(IRepo repo)
* The problem now becomes how are we going to call this new constructor since our application doesn't instaniate MVC controllers directly? After all this controller gets created automatically when it routes a request for it, and the asp.net Web API (which creates the instance of the controller class) doesn't know anything about this overloaded construtor or our new IRepo Interface. In fact, prior to setting up Ninject, the project will build with this new construtor but at run time an error will be generated complaining about not having a parameterless constructor.
* Understanding how the asp.net Web API works then is important. When the Web API instantiates an MVC controller class, it first calls System.Web.Mvc.IDependencyResolver.GetService, passing in the controller type. If GetService returns null, Web API looks for a parameterless constructor (or no constructor) on the controller class.
* Understanding the above point, we create a new class that implements System.Web.Mvc.IDependencyResolver and then we tell Asp.net MVC (aka the Web Api) to use our implemenation of IDependencyResolver. In this case we have created a Ninject Dependency Injection (DI) Container class called NinjectDependencyResolver that implements IDependencyResolver and overrides GetService and GetServices method. Also our Ninject DI Container includes a Constructor that expects a Ninject.Syntax.IResolutionRoot parameter (which will be assigned the Ninject Standard Kernel).
* Armed with our Ninject DI Container (class NinjectDependencyResolver) the rest of the set-up is done in the Global.asax. In the Global.asax.cs Application\_Start method we tell Asp.net MVC (aka the Web Api) to use our Ninject DI Container by calling the SetResolver method of System.Web.Mvc.DependencyResolver and passing it an instance of our DI Container to whos constructor we send the ninject Standard Kernel. We used a new method called SetUpNinject() for this purpose. Note that for constructor injection we load the mappings of interfaces to actual implemenations to the Standard Kernel object, see kernel.Load(new Dependency()). For growth (as more injection is done in our app) we have created a seperate class to hold all the dependency mappings called Dependency. Note, the Dependency class and the call to kernel.Load(new Dependency()) aren't needed for Setter Injection.

**Setter Injection**

* Setter Injecton is done via the [Inject] attribute on our public \_setterInjection property. Setter injection has nothing to do with the dependency mappings in the dependency class (Dependency() is mentioned above). To enable setter injection, we need to define our ninject DI container( in this case class ninjectdependencyresolver) and tell asp.net mvc to use it. That's it; no mapping of interfaces to implementations and no loading of those mappings.
* As in Constructor Injection in global.asax.cs application\_start method instaniate the ninject Standard Kernal, and pass that to our di container via dependencyresolver.setresolver(new ninjectdependencyresolver(kernel)), but the "kernel.Load(new Dependency());" is meaningless and would be omitted if this is the only type of injection the code uses.
* ***This seems like a less flexible way to do injection as \_setterinjection is declared as an implemenation type(ie.Repo1) and not as an interface type (ie not IRepo).***

**A variation on Constructor Injection**

* Using the same set up in the global.asax.cs application\_start method as with Setter Injection we can define a Constructor that has a parameter of a concrete (implemenation) type. For example "public DefaultController(Repo3 x)"
* As you see this constructor has a parameter of an implemenation type (ie Repo3) and not an interface type (ie IRepo).
* And as in the Constructor injection example above var x can be assigned to our private \_constructorInjection
* Again, this doesn't seem like a best practice as it's not intuitive as to why or how this constructor is called, and we aren't really doing Inversion of Control as var x is defined as a concrete type and not an Interface. We see this type of psuedo IOC used in C:\Code\KLAS\Klas.Toolbox\Areas\AssignmentPortal\Controllers\AssignmentPortalController.cs with the public AssignmentPortalController(AssignmentPortalFilterRepository assignmentPortalFilterRepository) constructor.