How to use ASP.NET MVC Core Dependency Injection.

Visual Studio 2017, ASP.NET Core 2.0

By John kocer- SmartIT

Introduction

In this article we will learn

-Dependency Injection
-Singleton
-Scoped
-Transient
-Loose Coupling
- Interface
-Repository Database
-Converting Simple To-do MVC Core application to use Dependency Injection.

Before we learn about dependency injection and Unity, we need to understand why we should use them. And in order to understand why we should use them, we should understand what types of problems dependency injection and Unity are designed to help us address.

Below is the some of the problems as a .NET developer we may encounter with.

Motivations

When we design, and develop software systems, there are many requirements to consider. Some will be specific to the system in question and some will be more general in purpose. We can categorize some requirements as functional requirements, and some as non-functional requirements (or quality attributes). The full set of requirements will vary for every different system. The set of requirements outlined below are common requirements, especially for line-of-business (LOB) software systems with relatively long anticipated lifetimes. They are not all necessarily going to be important for every system we develop, but we can be sure that some of them will be on the list of requirements for many of the projects you work on.

Maintainability

As systems become larger, and as the expected lifetimes of systems get longer, maintaining those systems becomes more and more of a challenge. Very often, the original team members who developed the system are no longer available, or no longer remember the details of the system. Documentation may be out of date or even lost. At the same time, the business may be demanding swift action to meet some pressing new business need. *Maintainability* is the quality of a software system that determines how easily and how efficiently you can update it. You may need to update a system if a defect is discovered that must be fixed (in other words, performing *corrective maintenance*), if some change in the operating environment requires you to make a change in the system, or if you need to add new features to the system to meet a business requirement (*perfective maintenance*). Maintainable systems enhance the agility of the organization and reduce costs.

Testability

A *testable* system is one that enables you to effectively test individual parts of the system. Designing and writing effective tests can be just as challenging as designing and writing testable application code, especially as systems become larger and more complex. Methodologies such as test-driven development (TDD) require you to write a unit test before writing any code to implement a new feature and the goal of such a design technique is to improve the quality of your application. Such design techniques also help to extend the coverage of your unit tests, reduce the likelihood of regressions, and make refactoring easier. However, as part of your testing processes you should also incorporate other types of tests such as acceptance tests, integration tests, performance tests, and stress tests.

Flexibility and Extensibility

Flexibility and *extensibility* are also often on the list of desirable attributes of enterprise applications. Given that business requirements often change, both during the development of an application and after it is running in production, you should try to design the application to make it flexible so that it can be adapted to work in different ways and extensible so that you can add new features. For example, you may need to convert your application from running on-premises to running in the cloud.

Parallel Development

When you are developing large scale (or even small and medium scale) systems, it is not practical to have the entire development team working simultaneously on the same feature or component. In reality, you will assign different features and components to smaller groups to work on in parallel. Although this approach enables you to reduce the overall duration of the project, it does introduce additional complexities: you need to manage multiple groups and to ensure that you can integrate the parts of the application developed by different groups to work correctly together.

Loose Coupling

You can address many of the requirements listed in the previous sections by ensuring that your design results in an application that loosely couples the many parts that make up the application. *Loose coupling*, as opposed to *tight coupling*, means reducing the number of dependencies between the components that make up your system. This makes it easier and safer to make changes in one area of the system because each part of the system is largely independent of the other.

How can we address some of the most common requirements in enterprise applications by adopting a loosely coupled design to minimize the dependencies between the different parts of your application. However, if a class does not directly instantiate the other objects that it needs, some other class or component must take on this responsibility. In this chapter, you'll see some alternative patterns that you can use to manage how objects are instantiated in your application before focusing specifically on dependency injection as the mechanism to use in enterprise applications.

What is Dependency Injection?

Dependency injection (DI) is a technique for achieving loose coupling between objects and their collaborators, or dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs in order to perform its actions are provided to the class in some fashion. Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. This approach is known as "constructor injection".2

When classes are designed with DI in mind, they are more loosely coupled because they do not have direct, hard-coded dependencies on their collaborators. This follows the Dependency Inversion Principle, which states that *"high level modules should not depend on low level modules; both should depend on abstractions."* Instead of referencing specific implementations, classes request abstractions (typically interfaces) which are provided to them when the class is constructed. Extracting dependencies into interfaces and providing implementations of these interfaces as parameters is also an example of the Strategy design pattern.

We have a simple ASP.NET CORE 2.0 To-do application written without Dependency Injection. Our task is the refactor this To-to application to make it loose Coupling.

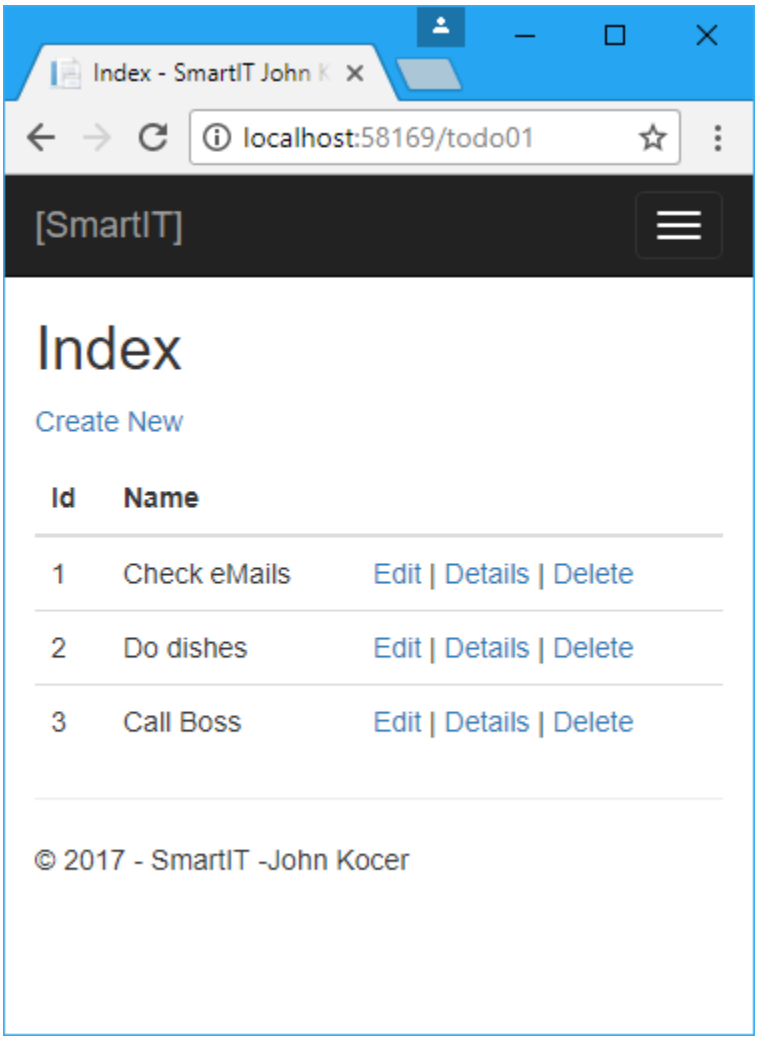
As a developer, you may have many situations that require you to work with some other developer codes.

-Download TodoMvcSolutin from below github repository.

https://github.com/SmartITAz/TodoMvcSolution

-Change solution name from TodoSolution.sln to TodoDependencyInjectionSolution.sln

-Compile and run the solution.



-Here is the question, what we need to change to make it loose couple?

Service lifetime

Before we can talk about how injection is done in practice, it is *critical* to understand what is **service lifetime**. When a component requests another component through dependency injection, whether the instance it receives is unique to that instance of the component or not depends on the lifetime. Setting the lifetime thus decides how many times a component is instantiated, and if a component is shared.

There are 3 options for this with the built-in DI container in ASP.NET Core:

1-Singleton
2-Scoped
3-Transient

Singleton means only a single instance will ever be created. That instance is shared between all components that require it. The same instance is thus used always.

Scoped means an instance is created once per *scope*. A scope is created on every request to the application, thus any components registered as Scoped will be created once per request.

Transient components are created every time they are requested and are never shared.

It is important to understand that if you register component A as a singleton, it cannot depend on components registered with Scoped or Transient lifetime. More generally speaking:

A component cannot depend on components with a lifetime smaller than their own.

The consequences of going against this rule should be obvious, the component being depended on might be disposed before the dependent.

Typically, you want to register components such as application-wide configuration containers as Singleton. Database access classes like Entity Framework contexts are recommended to be Scoped, so the connection can be re-used. Though if you want to run anything in parallel, keep in mind Entity Framework contexts cannot be shared by two threads. If you need that, it is better to register the context as Transient. Then each component gets their own context instance and can run in parallel.

-Here is the question, what we need to change to make it loose couple?

TodoController →TodoRepository
TodoRepository inherits from ITodoRepository

Step 1: Extract an Interface from your custom class which need to have all public methods, properties, indexes you will use in the interface. If you have some public methods or properties in the custom class but declaration of this public method or properties not in the inherited interface you cannot use it.

We have TodoRepository class inherits from ITodoRepository interface.

Step 2: Register your services in your Startup file **ConfigureServices** method. We added

```
services.AddSingleton<ITodoRepository, TodoRepository>();
```

```
our servise as Sigleton.
```

```
Why we chose as Sigleton?Because our service is an In-memory repository data services. We want only a
single instance will be created.
```

```
public class Startup
  {
    public Startup(IConfiguration configuration)
    {
      Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddMvc();
      services.AddSingleton<ITodoRepository, TodoRepository>(); // here where you add your services
    }
```

Step 3: Injection

Where should we inject TodoRepositor into TodoController?We can inject into TodoController constructor.

Here is TodoCOntroller before dependency injection. A default contructor is provided because the TodoController class contains no instance constructor declaration.

```
public  TodoController(){}
```

TodoCOntroller before dependency injection:

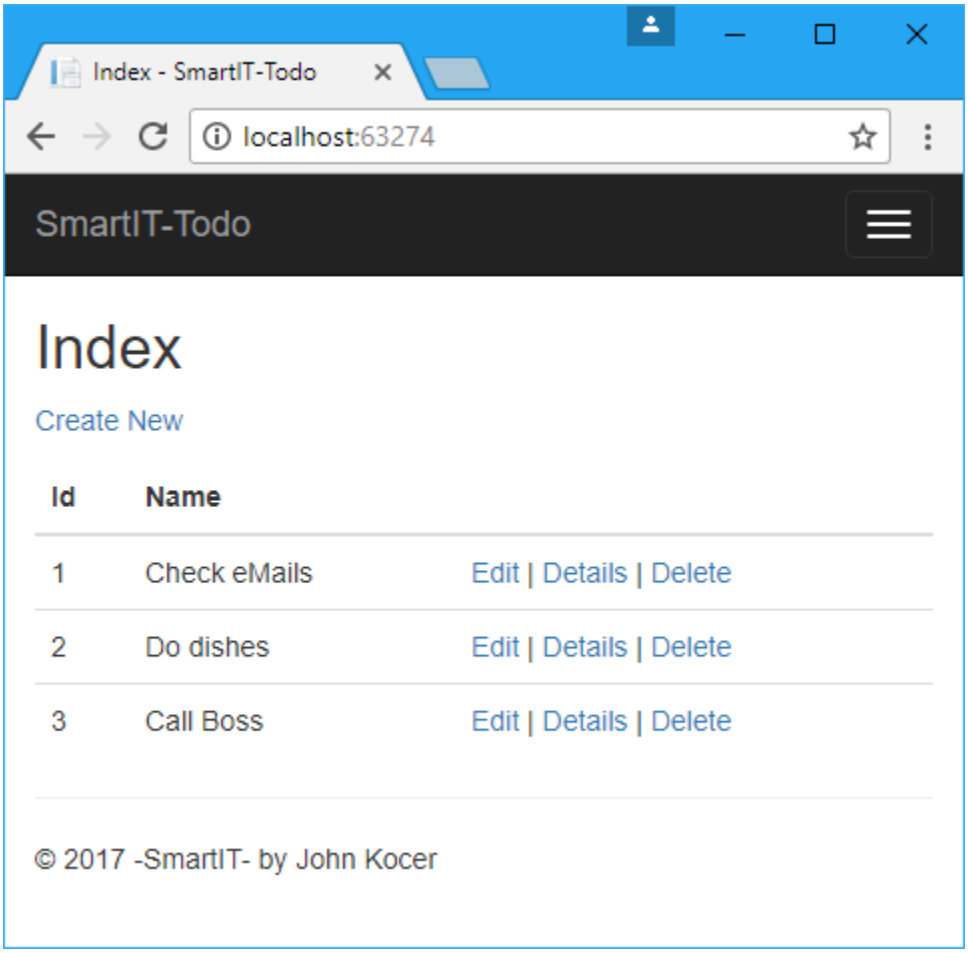
```
public class TodoController : Controller
  {
    TodoRepository _todoRepository = new TodoRepository();
    // GET: Todo
    public ActionResult Index()
    {
      var todos = (List<SmartIT.Employee.MockDB.Todo>)_todoRepository.GetAll();
      return View(todos);
    }
```

-We added constructor **`public TodoController(ITodoRepository todoRepository) that takes ItodoTrpository interface as argument.`**

```
public class TodoController : Controller
  {
    ITodoRepository _todoRepository;
    public TodoController(ITodoRepository todoRepository)
    {
      _todoRepository = todoRepository;
    }

// GET: Todo
    public ActionResult Index()
    {
      var todos = (List<SmartIT.Employee.MockDB.Todo>)_todoRepository.GetAll();
      return View(todos);
    }
```

-Run the project and it work with dependency injection now.



Summary

In this article, we will learn

-Dependency Injection
-Singleton
-Scoped
-Transient
-Loose Coupling
- Interface

-Repository Database
-Converting Simple To-do MVC Core application to use Dependency Injection.

Lab Exercise

A lab exercise for you to demonstrate what have you learned from this training material to create your own EmployeeController and add IEmployeeRepository Dependency Injection using EmployeeRepository included in this training material.

-You can follow above steps to create your own Employee CRUD ASP.NET MVC application.

```
     //Use below Employee repository to created your CRUD operation
      EmployeeRepository _employeeRepository= new EmployeeRepository();
      _employeeRepository.Add(new Employee() { Name = "Mat Stone", Gender = "Male", DepartmentId = 2,
Salary = 8000 });
      _employeeRepository.CDump("Employees");
    //  DebugHelper.cs(29): Employees
    //{
"Items":[{"Id":1,"Name":"Mike","Gender":"Male","Salary":8000,"DepartmentId":1,"Department":null},
    //{"Id":2,"Name":"Adam","Gender":"Male","Salary":5000,"DepartmentId":1,"Department":null},
    //{"Id":3,"Name":"Jacky","Gender":"Female","Salary":9000,"DepartmentId":1,"Department":null},
    //{"Id":4,"Name":"Mat
Stone","Gender":"Male","Salary":8000,"DepartmentId":2,"Department":null}],"Count":4}
```

For ASP.NET MVC Core Build in Dependency injection

Download color pdf version of this article with pictures.

https://github.com/SmartITAz/TodoDependencyInjectionSolution/blob/master/TodoAspMvcDependencyInjection.pdf

NOTE: If you need to copy and paste the code download the pdf file locally.

Download source code from GitHub: https://github.com/SmartITAz/TodoDependencyInjectionSolution