**You can refer to the Week5.WebApp1 for help with this lab.**

**Exercise 1**

[**X**] Create a new MVC project for this lab.  Copy code from last week's lab as a starting point.

Created full copy of lab 4 as a starting point.

1. Create a separate “repository” class with an associated interface for storing data in the database.  
   Unlike the example, for the lab you can use a single repository class for both of your entities.
2. Create an instance of the database context class in the constructor of your new “repository” class.
3. Replace the reference to the database context in each of your controllers with a reference the new “repository” class.  
   Create an instance of the “repository class” in the constructor of each controller.
4. Add methods to the “repository” interface and class that implement all of the database needs of your controllers.
5. Move the code in the controllers that implements the calls to the database context to the methods in the new “repository class”.
6. Add calls to the “repository” class methods in the controllers to satisfy their data access requirements.
7. Run the application to test to see if you can still successfully apply each of the operations (list, create, view details, edit and delete) to  
   each of the entities.

**Exercise 2**

1. Install the SimpleInjector NuGet package.
2. Add the code needed to configure dependency injection for the “repository” class.  
   See Global.asax and DependencyInjectionConfig.cs in the Week5.WebApp1 example.
3. Replace the creation of the “repository” class in each of the controllers with the injection of the “repository” class via a constructor in each controller.
4. Run the application to test to see if you can still successfully apply each of the operations (list, create, view details, edit and delete) to  
   each of the entities.

**Exercise 3**

1. Create a service class with an interface for your second entity (the entity you created in the last lab).
2. Add the code needed to configure dependency injection for the “service” class.
3. Make the new service class serve the purpose of retrieving and sending the entity's data required by the related controller.
   1. Add methods to the new service for all of the data needs of the controller.
   2. Inject the DbContext into the repository class instead of creating an instance in the repository constructor.  
      You'll need to add the code needed to configure the dependency injection of your DbContext derived class.
   3. Inject the related repository class into the service class.
   4. Inject the service class into the related controller class.
   5. Call the methods on the service class in the controller to retrieve and save data.
4. Run the application to test to see if you can still successfully apply each of the operations (list, create, view details, edit and delete) toeach of the entities.

**Exercise 4**

1. Add a method on the service to perform some "business rule logic" related to your entity that uses data in an instance of the entity.
2. Call the service class method created in step 4 when the action for displaying the details of the entity is invoked.
3. Display the results of the call to the service method in the entity’s detail view.  Pass the result to the "details view" via the entity's view model.
4. Run the application to test to see if the new "detail" based on the result of the service call is displayed correctly.
5. Briefly explain what dependency injection is and why it is beneficial.

**Exercise 5**

1. Create new project in you solution for unit tests.
2. Install the following NuGet packages in your new "test" project:
   1. **NUnit**
   2. **NUnit3TestAdapter**
   3. **FakeItEasy**
3. Create one or more unit tests to test to result of calling the "Business logic" service method.  
   You'll need to create a fake/mock repository in order to control what data the service class gets when you run the test.  
   For help, see the unit tests in the Week5.WebApp1 example.
4. Run the unit tests via the Visual Studio test explorer (Test => Windows => Test Explorer).  
   Did the tests succeed.  If not, fix the application code or the unit test and rerun the unit tests.
5. Break the service method being tested intentionally and return the unit test(s).  You should see at least one unit test fail.  
   Fix the tested service method and return the test.  You should see all green now.  Time to celebrate!
6. Briefly explain what unit tests are and why they are beneficial?

**Extra Credit**

Add logging to your application.  See the Week 5 Lab video for a short presentation on logging.