Dependency Injection

The goal of this lab is to teach you what dependency injection is, why it's important, and how to do it. We'll be starting off with a very simple project, migrating it to a project structure that allows for dependency injection, implementing dependency injection, and then adding some tests.

Getting it to work

Downloading the Project

The project is located here. Run `git clone https://github.com/raik383h/DependencyInjection.git` to download the project.

Run the project as is

Make sure there are no problems with running the project locally. Try adding a few contacts, editing and deleting.

Add validation to the Contacts Controller

Right now, there is nothing preventing people from entering invalid emails and phone numbers. Normally, this is something that you'd also want to handle on the front end, but we'll just do backend validation. Go to `ContactsController.cs`, and look at the various methods. Add validation for the phone number and email address in the `PostContact` and `PutContact` methods before the database is modified. Return a `BadRequest()` with a description if either are invalid.

Splitting up logic

Right now, all the logic is in controllers. It's really easy to find and work with, but as the project grows, it will become super difficult to maintain. Imagine having to do validation and update multiple different models in just that one method. We could just make more methods, or more classes to help organize and separate the logic, but we're actually going to go a step further, by creating files in entirely different projects.

Move models to core project

In `DependencyInjection.Web/Models` there is a `Contact.cs` file. In `DependencyInjection.Core/Models` there is also a `Contact.cs` file. Copy the rest of the properties and attribute tags to the file in the core project, and delete the one in the web project.

Implement the Contacts Accessor

Open the `DependencyInjection.DataAccess/ContactsAccessor.cs` class. Implement the `Find`, `Insert`, and `Delete` methods using the `DbSet<Contact> Contacts` defined at the top of the class.

Implement the Contacts Engine

Open the `DependencyInjection.BusinessLogic/ContactsEngine.cs` class. Implement the `InsertContact` and `UpdateContact` methods using the provided `ContactsAccessor`. Also, move the validation logic that you did in the the controller to the appropriate methods, blocking database changes if validation fails. Also, do not forget to call `_contactsAccessor.SaveChanges()` after you are finished doing database operations that will modify anything.

Update the Controller

Now, all of the logic that's in the controller should be in other places. Delete the `ApplicationDbContext` class, and fix the logic in the controllers to make use of the new `ContactsEngine` instead.

Run the project

There shouldn't be any errors at this point, so run the project and check that everything still works.

Implementing Dependency Injection

Add an interface for ContactsAccessor in core project

Look at the `ContactsAccessor` again, and create an interface for it in the core project. Make a new folder called `Accessors` and add the interface there, calling it `IContactsAccessor`. Copy all the method signatures (just the names, return types, and parameters, not the implementation or access modifier) from `ContactsAccessor` to `IContactsAccessor`. Go back to the `ContactsAccessor` and also extend this interface.

Add interface for ContactsEngine in core project

Look at the `ContactsEngine` again, and create an interface for it in the core project. Make a new folder called `Engines` and add the interface there, calling it `IContactsEngine`. Once again, copy all the method signatures from `ContactsEngine` to `IContactsEngine`. Go back to the `ContactsEngine` and extend this interface.

Register the interfaces with Unity

Now that everything is defined, let's go ahead an actually do the dependency injection. Right click on `DependencyInjection.Web` and select `Manage Nuget Packages`. Search for "unity" and install the `Unity` and `Unity.AspNet.WebApi` packages by `Microsoft.Practices.Unity`. Notice that two files have been added to the `App_Start` folder in the web project.

Open the `UnityConfig.cs` file and find the TODO section of the `RegisterTypes` method. Following that format, register the two interfaces you just made with the implementations. Note that you will need to add the appropriate using statements.

Remove the manual dependencies and use dependency injection instead

Now, go to the `ContactsEngine`. Notice how we're still manually creating a `ContactsAccessor`. Let's just inject that instead. Change the constructor to have a parameter of `IContactsAccessor`, and change the class variable to be an `IContactsAccessor` instead of directly being a `ContactsAccessor`. The top of the class should now look like this:

```
private readonly IContactsAccessor _contactsAccessor;

public ContactsEngine(IContactsAccessor contactsAccessor)
{
    _contactsAccessor = contactsAccessor;
}
```

You have just injected your first dependency.
You should notice now that the `ContactsController` has an error because it is missing the new parameter to manually create a `ContactEngine`. Change this to be injected as well. The top of the class should look like this:

```
private readonly IContactsEngine _contactsEngine;

public ContactsController(IContactsEngine contactsEngine)
{
    _contactsEngine = contactsEngine;
}
```

These classes have dependencies on other classes, but because we inject them, we don't have to worry about manually creating each dependency. Once configured, dependency injection allows for complex classes to easily be injected and used.

Using Dependency Injection

So this is all well and good, and it's slightly easier to use more complex classes. But why did we bother doing all this work when we haven't changed functionality at all? Modularity is huge. Doing dependency injection means your code is significantly more modular, and can very easily be tested. Before, you *wouldn't be able* to write a test for the ContactsEngine. Now we can. So we're going to. We'll make sure the validation logic on the emails and phone numbers works as expected.

Complete the MockedContactsAccessor

Implement the `Exists` method for a list in the `DependencyInjection.BusinessLogic.Tests/MockedAccessors` folder.

Add tests for the validation logic

Implement the `InsertContact` and `UpdateContact` methods in the `ContactsEngineTests.cs` file. Pay attention to how the `GetAllContacts` method works.

Run the tests

Open the Test Explorer window from the tabs at the top `Test > Windows > Test Explorer` and then build the solution, by clicking `Build > Build Solution`. The tests should appear. Run them, and take a screenshot of all the tests passing.