Tips and tricks for adding unit tests to legacy applications

Legacy codes come with tight coupling or lacks abstraction of dependencies. In legacy code, classes and dependencies usually have more than a single purpose. Adding unit tests to existing application without a business knowledge sometimes can get into our way. Some steps we need to take are refactoring the classes to extract interfaces, especially if the class is a dependent class to the class in test. Injecting dependencies, refactoring the class to split functions into simpler unit of works, using mocking (i.e. light classes such as provided by frameworks i.e. MOQ), and implementing testing frameworks such as MS Test are some of the things we can do resolve legacy codes. In this presentation, I will share my experience adding unit test to existing .Net legacy application and illustrate some tips and tricks to make this applicable unit testable.

It must be acknowledged that a working code however bad or tightly coupled is a working code if it is serving a purpose. Thus, re-writing the entire code base is not a good software development practice. As part of incremental improvement, unit tests can be added as we make changes to the code base and we are one step further towards a good code. Again, we must remember that ‘good’ is a relative concept and it changes its definition from person, context, time and resource constraints. An ideal condition is to have a legacy code that has unit tests when all the codes. In this presentation, I am going to share my experience improving the legacy code base.

Note that the ResultController is coupled tightly with Manager class.

To start the decoupling, the first step is to move the initialization to constructor.

Later one for unit testing, we have to mimic the behavior of manager. So we will need an interface. So we can go ahead and create an interface for Manager.

Once that is done, we create a private variable of type IManager. and create a default constructor where \_manager is initialized with a new instance of Manager. If we fail to do this, the existing code will fail.

We create another constructor in which we inject the dependency via constructor.

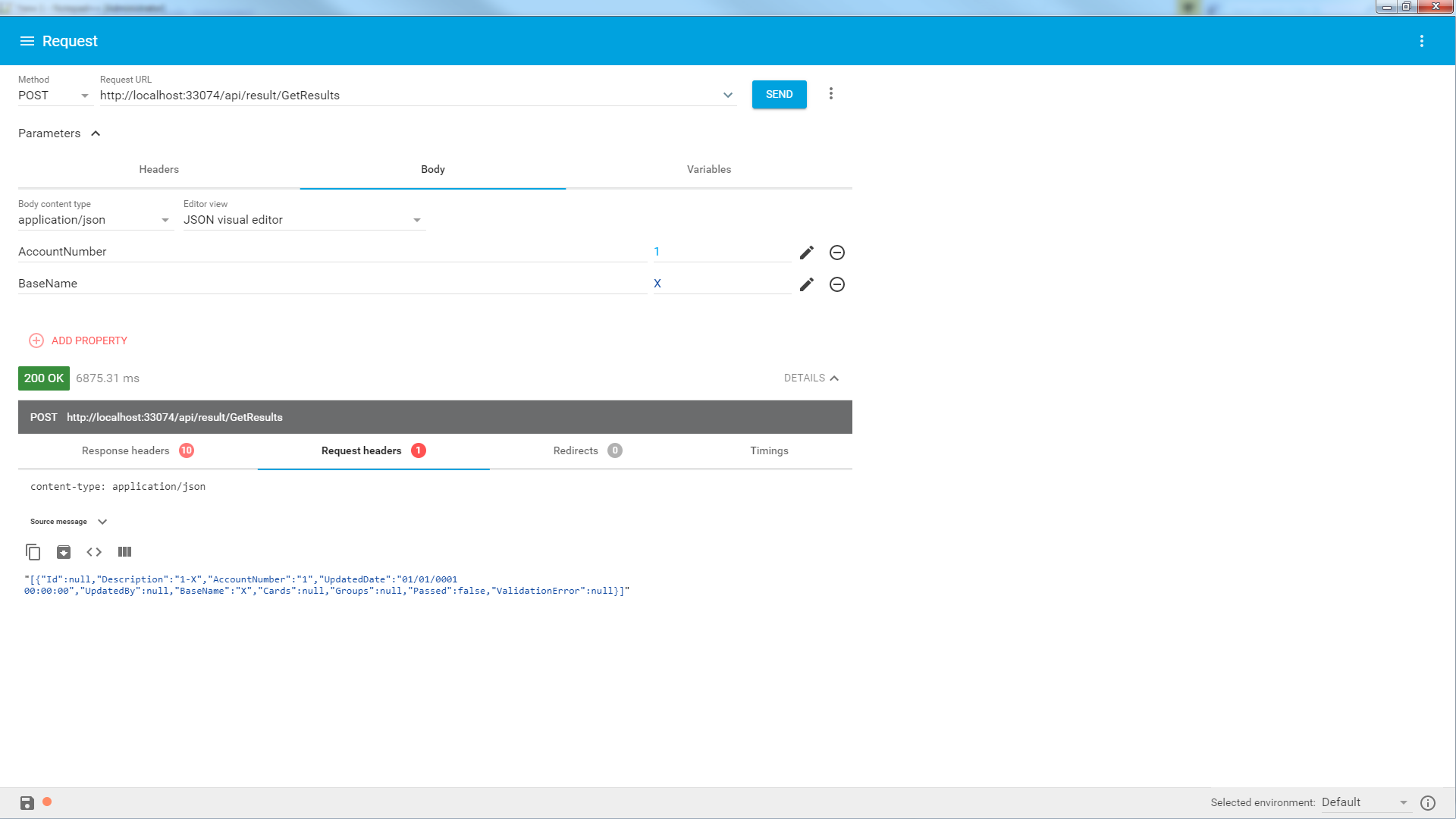
With these changes, the GetResults method is free from dependency to the behavior of the \_manager. This can be provided at initialization step.

The rest of the flow does not need to change.

Now going back to Manager, we have a tight coupling with ResultsRepository() in the GetResultList method. We can take the similar steps that we took in Controller class to manage the dependency.

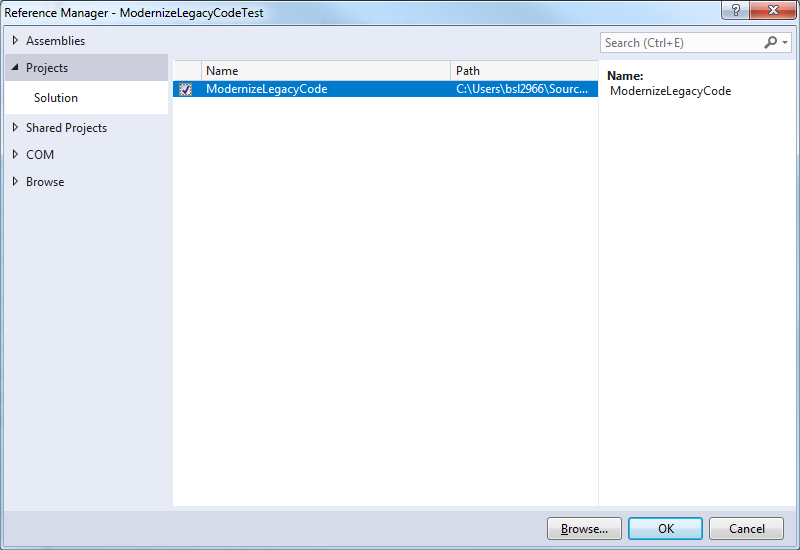
- We will create an interface for ResultsRepository and make ResultsRepository implement the interface.

Now we will test these changes. I used Advanced Rest Client to test this.



Next I will add a unit test project in visual studio. First of all, I am going to test the controller class. There is just one method ‘GetResults()’ in this class. I can have more than one test methods to test the class.

Now that I have to refer to the api project, I will add reference.



With the addition to reference to I have access to the class I am testing in the test project.