RefactorMe Web API

Zaldy De Leon

November 6, 2017

Table of Contents

Introduction

As developers, most of our time is spent working on and in existing software code. We regularly change source code written by others (or ourselves). Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code slowly sinks from engineering to hacking.

Refactoring is the process of clarifying and simplifying the design of existing code, without changing its behavior. Agile teams are maintaining and extending their code a lot from iteration to iteration, and without continuous refactoring, this is hard to do. This is because un-refactored code tends to rot. Rot takes several forms: unhealthy dependencies between classes or packages, bad allocation of class responsibilities, way too many responsibilities per method or class, duplicate code, and many other varieties of confusion and clutter. (Code Refactoring, 2017)

With refactoring one can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay. (Fowler, 2002)

The existing application was written terribly on purpose. It is a test designed to appraise a C# applicant's ability to look into an existing application and make changes to improve its design and operation.

This paper evaluates the existing application with the objective of upgrading its design and developing a robust and scalable system leveraging latest technology and best practices. Refactoring is one of these best practices.

The Existing Application

The existing application is a Web API written in C#. An initial look into the project reveals that while it is not the best in terms of design and lacking in best practices, it may work with a few tweaks.

Concerns

Upon closer look, a number of concerns were discovered.

- Adding a hyphen to a project name, as in refactor-me, generates unexpected errors and exceptions. This same issue is reported in a number of posts on the net. A few is listed below.
  - Dash "-" in project name is not converted to "_" in the namespace
  - Incorrect namespace generated for new files when the project contains special characters
  - Project names with hyphens
  - Default namespace with dashes causes newly added classes to have the same

  While hyphen is valid character in a project name, it is best not to use it a project name to avoid unexpected issues later in the development stage.

- Missing classes. An ASP.NET web application needs to run an initialization code at start up. It includes:
  - BundleConfig class – This class is used to create and register bundles for CSS and JS files. By default various bundles are added in this file including jQuery, jQueryUI, jQuery validation, Modernizr and Site Css.
  - FilterConfig class – This class is used to create and register global MVC filters - error filter, action filter etc. By default it contains HandleErrorAttribute filter.
  - RouteConfig class – This class is used to register various route patterns for your ASP.NET application. By default, one route is registered here named as Default Route.
- The ProductsController class performs CRUD operations on the database itself without any layer of abstraction.
- In the Models folder is the Helpers class which contains the connection information to the database.
- Also, in the Models folder is the Products class which contains the properties of the Product entity, and methods that perform database I/O.

Design Flaws of the Existing Application

An attempt was made to fix the existing application. However, every time a new class is added, the IDE generates an error to with dependencies and missing references. While these errors can be resolved, it does not guarantee that the API would work. Experience dictates that creating the project from scratch using a template is more practical when time of the essence (like in this case).

- Missing classes/methods. While these can be created manually, a Web API template automatically creates these classes and downloads/installs the dependencies and/or references required.
- Products controller class. While some say, "Why Entity Framework renders the Repository pattern obsolete", Bispin Joshi, in his book, Beginning SOLID Principles and Design Patterns for ASP.NET, wrote:

  "In simple applications you may not [need to implement the repository pattern], but in real-world systems adding one more layer between the client code and the Entity Framework is beneficial…"

  The repository is intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing this pattern can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development. (Dykstra, 2013)

- Database I/O inside the Products class. The Single Responsibility Principle (SRP) states that, "A class should have only a single responsibility." It can be as simple as holding an application state or as complex as resource-intensive processing. However, if a class is designed to carry multiple responsibilities, it can create problems at a later stage. (Joshi, 2016)

The Repository class is the best place to put the database I/O methods.

- Helpers class in the Models folder. Another principle of object-oriented design is Separation of Concerns (SoC). The principle suggests separating a computer program into distinct sections, such that each section address a separate concern. (Wiki, 2017)

  The Helpers class should be stored elsewhere and not in the Models folder.

The Refactored Application

The refactored application is created from scratch using the Visual Studio Express for Web 2015 Web API template. The rationale for creating a new project is to avoid issues that may arise resulting from the design flaws described in the previous section.

The project is named RefactorMe, this time, without the hyphen. The same database is used and the same functionality is provided. Where relevant, appropriate tools, technology and best practices are applied.

The project is organized as follows:

- RefactorMe – this is the main project. It contains the application data (Database.mdf), start up classes, content, controllers, and scripts.
- RefactorMe.Models – this is the data access layer. It uses the Entity Framework to talk to the database which is set up as an ADO.Net Entity Data Model. This project also contains the repository and unit of work classes as well as the database model and model properties.
- RefactorMe.Entities – this class library contains the transfer objects that will be used to move data to/from the project and the business logic layer.
- RefactorMe.Services – this is the business logic layer. It is concerned with the retrieval, processing, transformation, and management of application data; application of business rules and policies; and ensuring data consistency and validity. (MSDN, Business Layer Guidelines, 2017)

The class libraries implement the Separation of Concerns (SoC) principle, a key principle of software development and architecture. At a low level, this principle is closely related to the Single Responsibility Principle of object oriented programming. The general idea is that one should avoid co-locating different concerns within the design or code. (Wiki, 2017)

API Controllers

The controllers in the refactored application do not have direct access to the physical database. Access is done thru the Entities and Services class libraries, which in turn, invokes the methods in the Generic Repository and Unit of Work classes contained in the Models class library.

Products Controller

The ProductsController handles all HTTP request and response on the Product entity. It exposes six end points as follows:

GET products

- Returns all products
- <URL>/products

GET products?name={name}

- Returns products matching the specified name.
- <URL>/products?name=ProductName

GET products/{id}

- Returns the product that matches the id specified, where id is the product GUID.
- <URL>/products/10

POST products

- Create a product
- <URL>/products

UPDATE products/{id}

- Update the product that matches the id specified, where id is the product GUID.
- <URL>/products/10

DELETE products/{id}

- Delete the product that matches the id specified, where id is the product GUID.
- <URL>/products/10

Product Options Controller

The ProductOptionsController handles all HTTP request and response on the Product Option entity. It exposes five end points as follows:

GET products/{id}/options

- Returns all product options for the id specified, where id is the product GUID
- <URL>/products/22/options

GET products/{id}/options/{optiondid}

- Return the product option that matches the id and optionid specified, where id is the product GUID and optioinid is the product option GUID.
- <URL>/products/22/options/33

POST products/{id}/options

- Add a product option to the id specified, where id is the product GUID.
- <URL>/products/22/options

UPDATE products/{id}/options/{optionid}

- Update the product option that matches the id and optionid specified, where id is the product GUID and optionid is the product option GUID.
- <URL>/products/22/options/33

DELETE products/{id}/options/{optionid}

- Delete the product option that matches the id and optionid specified, where id is the product GUID and optionid is the product option GUID.
- <URL>/products/22/options/33

Classes and Interfaces

ProductEntity

The ProductEntity class defines the properties of the Product model. This class is used to provide a layer of abstraction between the client code and the database entities.

ProductOptionEntity

The ProductOptionEntity class defines the properties of the Product Option model. This class is used to provide a layer of abstraction between the client code and the database entities.

GenericRepository

The Repository pattern is used to manage CRUD operations through an interface that exposes domain entities and hides the implementation details of database access code. It is intended to create an Abstraction layer between the Data Access layer and Business Logic layer. It is a data access pattern that prompts a more loosely coupled approach to data access. (Watmore, 2015)

As far as ASP.NET applications are concerned, they may use ADO.NET objects or Entity Framework for the database operations. If the application uses ADO.NET objects, the repository pattern is definitely beneficial. (Joshi, 2016)

In the refactored application, a GenericRepository class is created. Its signature, `public class GenericRepository<T> where T : class` allows code reuse for the different entities used in the project.

The public member methods in the GenericRepository class perform CRUD operations on the domain entities. The relevant methods and method signatures are listed below.

```
// Generic method to get an entity using id
public virtual T GetById(object id)

// Generic method to insert an entity
public virtual void Insert(T entity)

// Generic method to delete an entity using id
public virtual void Delete(object id)

// Generic method to delete an entity
public virtual void Delete(T entityToDelete)

// Generic method to update an entity
public virtual void Update(T entityToUpdate)

// Generic method to get entities using criteria specified
public virtual IEnumerable<T> GetAll(Func<T, bool> where)

// Generic method to get entities using criteria specified
public T Get(Func<T, bool> where)
```

```
// Generic method to delete entities using criteria specified
public void Delete(Func<T, bool> where)

// Generic method to get all entities
public virtual IEnumerable<T> GetAll()
```

UnitOfWork

The unit of work pattern keeps track of a business transaction that is supposed to alter the database in some way. Once the business transaction is over, the tracked steps are played onto the database in a transaction so that the database reflects the desired changes. Thus the unit of work pattern tracks a business transaction and translates it into a database transaction, wherein steps are collectively run as a single unit. The SaveChanges() method executes these operations in a transaction as a unit of work. (Joshi, 2016)

In the refactored application, a unit of work class is created. Its signature,

```
public class UnitOfWork : IDisposable
```

implements IDisposable to free up resources, i.e., connections and objects. The class defines the get/set method properties of the Product and Product Option repositories.

ProductServices

The ProductServices class implements the IProductServices interface.

Interfaces are a powerful programming tool because they let you separate the definition of objects from their implementation. (MSDN, When to Use Interfaces, 2017)

The IProductServices interface defines six methods, namely:

```
GetAllProducts();
GetProductByName(string name);
GetProductById(Guid id);
CreateProduct(ProductEntity entity);
UpdateProduct(Guid id, ProductEntity entity);
DeleteProduct(Guid id);
```

These methods perform the respective tasks. For example, the GetAllProducts() method executes a LINQ to Entities query against the Product DbSet and fetches all the Product entities from the database.

Similarly, the CreateProduct(ProductEntity entity) method adds a new Product to the Product DbSet.

The methods also queries the data source for the data, maps the data from the data source to a business entity and persists changes in the business entity to the data source.

ProductOptionServices

The ProductOptionServices class implements the IProductOptionServices interface. The IProductOptionServices interface defines five methods, namely:

```
GetOptionsForProduct(Guid id);
GetByProductIdAndOptionId(Guid id, Guid optionid);
CreateProductOption(Guid id, ProductOptionEntity entity);
UpdateProductOption(Guid id, Guid optionid,
ProductOptionEntity entity);
DeleteProductOption(Guid id, Guid optionid);
```

These methods perform the respective tasks. For example, the GetOptionsForProducts() method executes a LINQ to Entities query against the ProductOption DbSet and fetches all the ProductOption entities for a Product from the database.

Similarly, the CreateProductOption(ProductOptionEntity entity) method adds a new ProductOption for a Product to the ProductOption DbSet.

The methods also queries the data source for the data, maps the data from the data source to a business entity and persists changes in the business entity to the data source.

Testing

The refactored Web API was tested using Postman. Postman is an HTTP client for testing web services. There is a simple test client for ASP.NET Web API, WebApiTestClient 1.1.1, that can be downloaded from Nuget and installed as part of the application, but an independent and impartial test is preferred.

What next

There are still flaws in the design of the refactored application. The following changes are recommended to make it better:

- Use Inversion of Control with Unity Container and MEF (Managed Extensibility Framework) to resolve the dependency of RefactorMe.Models.UnitOfWork from RefactorMe.Services.
- Implement some kind of Authentication/Authorization to secure the application and recommend HTTPS.
- Implement request logging and exception handling. Since we are exposing our end points through the API, we must know where the requests are coming from and what requests are coming to our server. Logging is beneficial in a lot of ways - debugging, debugging, tracing, monitoring and analytics.
- Database columns should use the smallest data type necessary to implement the required functionality. For example, the author prefers an integer primary key as these are stored in a much more compact format than character formats. Primary keys are also used as foreign keys in another table and may be used in indexes. The smaller the key, the smaller the index, the less pages used in the cache.

Tools, Technology and Best Practices

The refactored application is a RESTful API developed using the latest technology and best practices. These are briefly described below.

Visual Studio 2015 Express for Web

Everything you need to create great apps for devices or desktop apps, for the web and in the cloud. Write code for iOS, Android, and Windows in one IDE. Get great IntelliSense, easy code navigation, fast builds, and quick deployment.

ASP.NET Web API

ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. It is an ideal platform for building RESTful applications on the .NET Framework.

C#

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework.

Repository Pattern

The Repository pattern is used to manage CRUD (Create, Read, Update, and Delete) operations through an abstract interface that exposes domain entities and hides the implementation details of database access code.

Unit of Work Pattern

The Unit of Work pattern is used to group one or more database operations into a single transaction or "unit of work", so that all operations either pass or fail as one.

Inversion of Control (IoC)

In procedural programming, control of what happens and when, is entirely in the hands of the programmer. With IoC, control over happens and when, is inverted or placed in the hands of the user.

Dependency Injection (DI)

Dependency injection is a pattern used to implement IoC. It is the act of connecting object with others, or "injecting" objects into other objects, usually done by an assembler rather than by the objects themselves.

Unity Container

The Unity Container (Unity) is a lightweight, extensible dependency injection container. It facilitates building loosely coupled applications and provides simplified object creation, abstraction of requirements, increased flexibility by deferring component configuration to the container, service location capability, instance and type interception, and registration by convention.

Bootstrap

An open source toolkit for developing with HTML, CSS, and JS.

Postman

A powerful HTTP client for testing web services. Postman makes it easy to test, develop and document APIs by allowing users to quickly put together both simple and complex HTTP requests.

References

Code Refactoring. (2017). Retrieved from VersionOne: https://www.versionone.com/agile-101/agile-software-programming-best-practices/refactoring/

Dykstra, T. (2013, July 30). *Microsoft Docs*. Retrieved from Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Fowler, M. (2002). *Refactoring: Improving the Design of Existing Code*. 1999: Addison Wesley Longman, Inc.

Joshi, B. (2016). *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*.

MSDN. (2017). *Business Layer Guidelines*. Retrieved from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/ee658103.aspx

MSDN. (2017). *When to Use Interfaces*. Retrieved from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/3b5b8ezk(v=vs.90).aspx

Watmore, J. (2015, January 28). *Unit of Work + Repository Pattern in MVC5 and Web API 2*. Retrieved from Jason Watmore's Blog: http://jasonwatmore.com/post/2015/01/28/unit-of-work-repository-pattern-in-mvc5-and-web-api-2-with-fluent-nhibernate-and-ninject

Wiki. (2017, March 21). *Separation of concerns*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Separation_of_concerns