Completed .NET Code Test

Kevin Narushoff

Statement of Delivery

The following document describes the delivery of my implementation of the .NET Code Test. The delivery and related artifacts represent an initial delivery as found within a disciplined agile delivery process, known as a Minimally Viable Product (MVP). As such, the goal of the MVP is to provide sufficient features to satisfy early adopters.

Normally, the work surrounding the MVP, would require subsequent iterations to follow, mainly to

collaborate with the stakeholders (i.e. customers, product management, design, user experience, and the product documentation team) and receive guidance for development until product completion. The final product is a complete set of features as a consequence of implementing the feedback from the product's initial users.

The delivery of this MVP met the deadline that was stated in the original estimate:

**Tuesday April 25, 2017**.

Requirements Overview

The requirements offer a UI sample, baseline functionality, supported browsers, and the concept of a new Product entry. Each requirement was satisfied as follows:

1. Visible Fields

As stated in the requirements, the application was required to maintain visibility of the following 7 fields from [Production].[Products] table (as found in the AdventureWorks2012 database): Name, ProductNumber, Color, SafetyStockLevel, ReorderPoint, StandardCost, and ListPrice.

While the above was accomplished adequately, a later discovery revealed that other fields were also required in order to write a record to the database, and they are included (although remained invisible) in the Model:

rowguid, ModifiedDate, SellStartDate, and IsDeleted.

The last one, “IsDeleted”, was a custom addition to the table; its purpose is described in Additional Features section.

1. Overview

The requirements state a base-level functionality which involves a grid that presents the 7 “visible” fields mentioned above and features the following capabilities:

* Pagination – this feature was accomplished using a library and Asp.Net MVC HtmlHelper methods for generating paging control with the PagedList library. PagedList makes it easier to develop paging code by allowing the use of any IEnumerable<T> and by specifying the page size and desired page index and selecting a subset of that list.
* Sorting – This was accomplished with a GET call to the server using ASP.NET MVC.
* Column Filtering – the requirements for this feature are generically stated leaving room for interpretation when implemented. As a consequence, design choices needed to be made. See the “Additional Features” section for commentary.
* Data Validation – Validation was accomplished using primarily server-side coding to validate fields.

1. Browser Support - The application has been tested successfully with the following browsers:

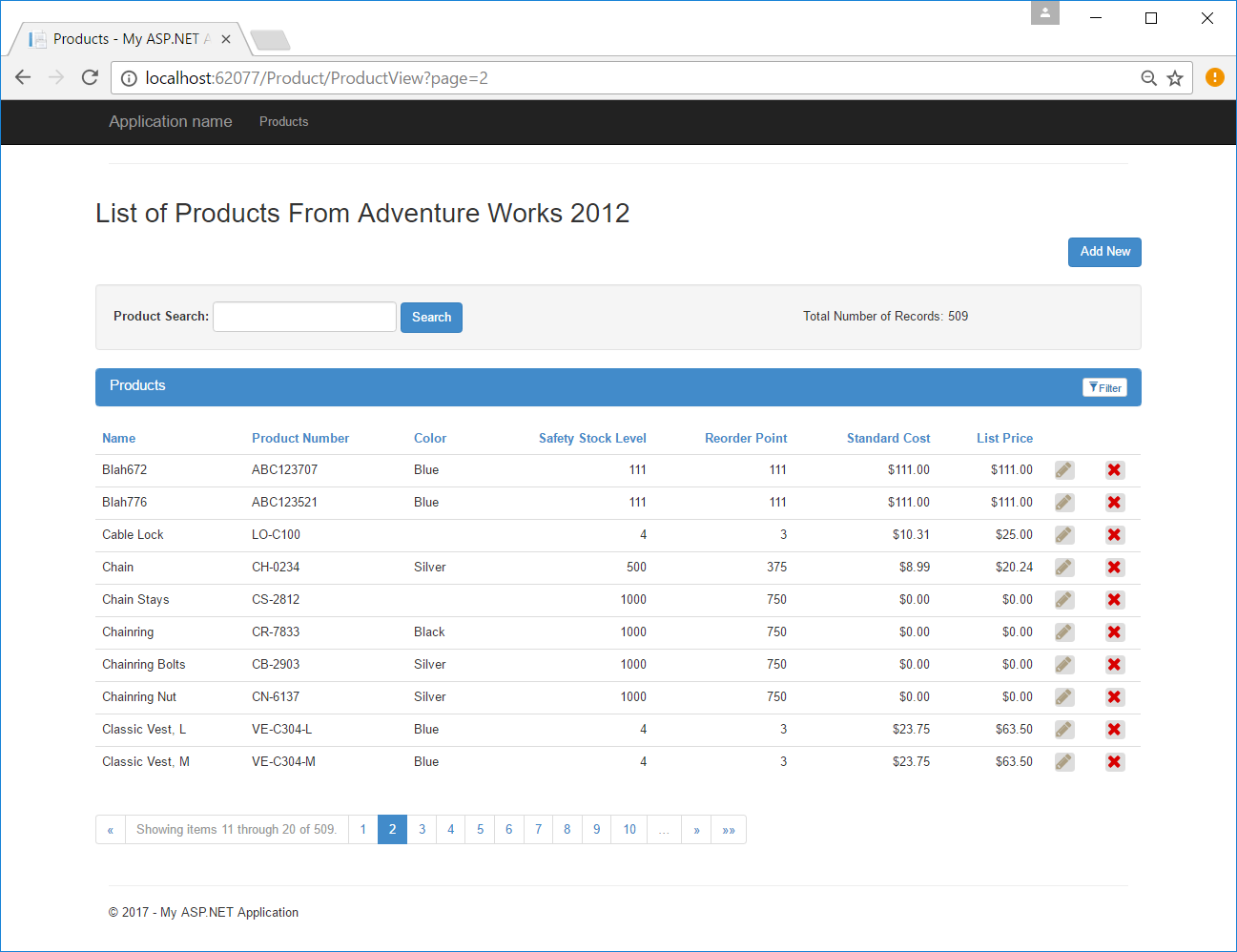
- IE 9+

- Edge

- Firefox

- Chrome

1. User Interface – below is a screen capture of the application which was created to satisfy requirements for: Adding a new Products, sorting, pagination, Search and filtering. Additional features are also pictured. See “Additional Features” Section for the discussion on those.



1. New Rows (Adding New Products)

Users are able to add a new product to the database and each field is validated prior to saving the record.

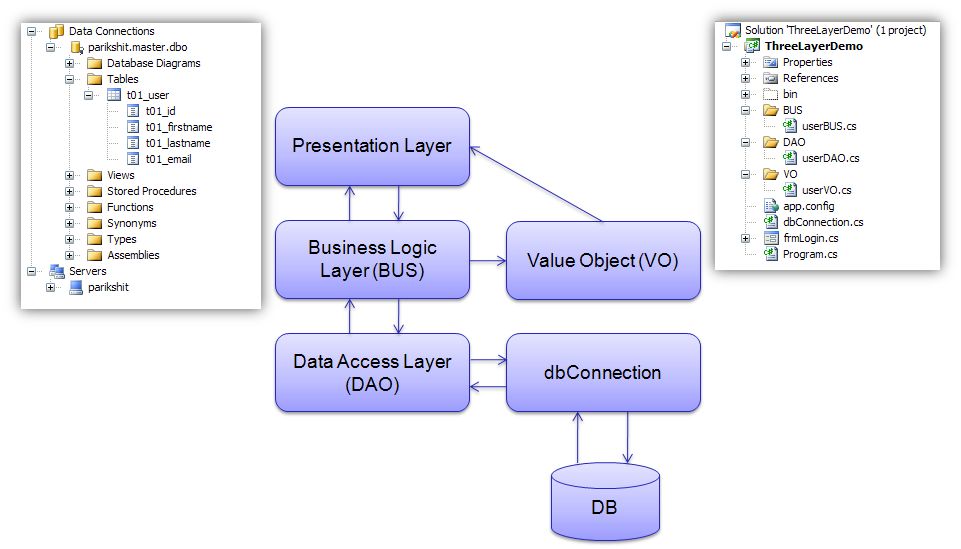
All base level requirements were completed for the aforementioned functionality; some creative license was employed where stated requirements left room for interpretation. Below is a detailed discussion of the components used to accomplish the functionality.

Description of the source code

The source code is organized in a multi-tier architecture and employs Object Oriented best practices with strict observance to SOLID software design principles (i.e. Single-responsibility, Open-closed, Liskov, Interface Segregation, Dependency Inversion).

The base coding layer, with the fewest dependencies is the Data Access Layer. It is patterned from the

[repository pattern](http://martinfowler.com/eaaCatalog/repository.html), a pattern used to separate the logic the business logic that acts on the model, from other logic that retrieves the data and maps it to the entity model. The code found in this layer is responsible for managing data as it pertains to the data repository as it contains the Domain Object(s) and the Entity Framework configuration like that of the main Context Object. Among its uses are connection with the database and provision for separation between the business logic and the persistence mechanism.



To produce the operative functionality for the Data Access Layer, Entity Framework can be used to generate the Domain Model. This layer exhibits an otherwise standard implementation of the Entity Framework Data Access ORM. Since it is designed against a single table, this implementation is a first-step toward a full-scale DAL conducive to handling a more elaborate schema.

The next layer in the “logic chain” is the Business Logic Layer or Service Layer which handles the CRUD functionality. It is dependent on the Data layer for its Domain Objects and Context object and manages application business logic and the communication between the Data and UI layers.

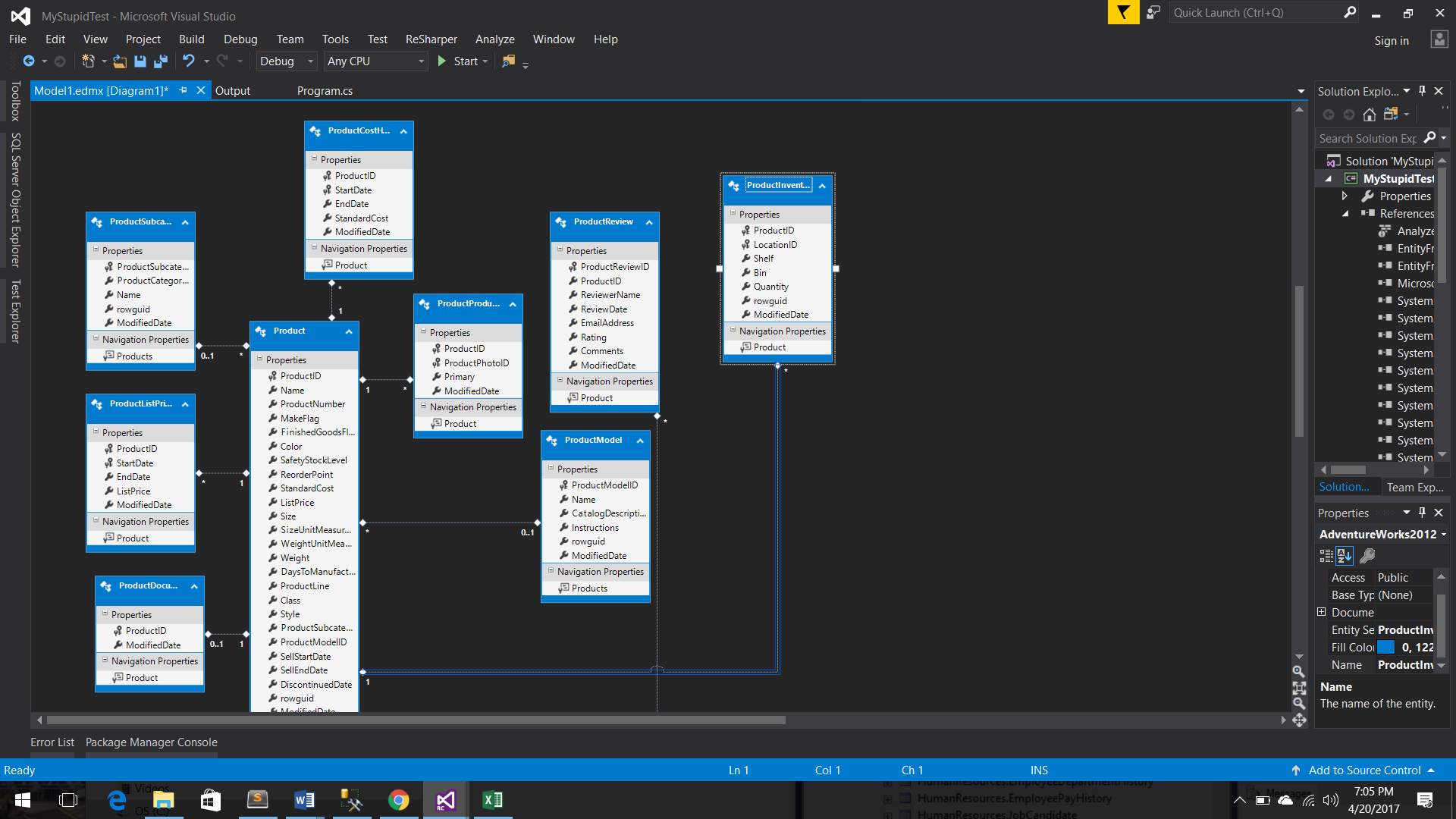
Finally the Presentation Layer contains the core Presentation Logic used in the User Interface. Built around the Model-View-Controller pattern, it is responsible for GETting or POSTing data via direct calls and Asynchronous (AJAX) calls into the Service Layer. The code in this layer demonstrates functionality that includes Server-side Controller Actions, CSS, multiple Javascript libraries (KnockoutJS, AngularJS, etc) and Asynchronous Client-Server data transfer.

Additional Features

Since the application is not limited to the features stated in the requirements document, additional features were applied to the scenario. These are as follows:

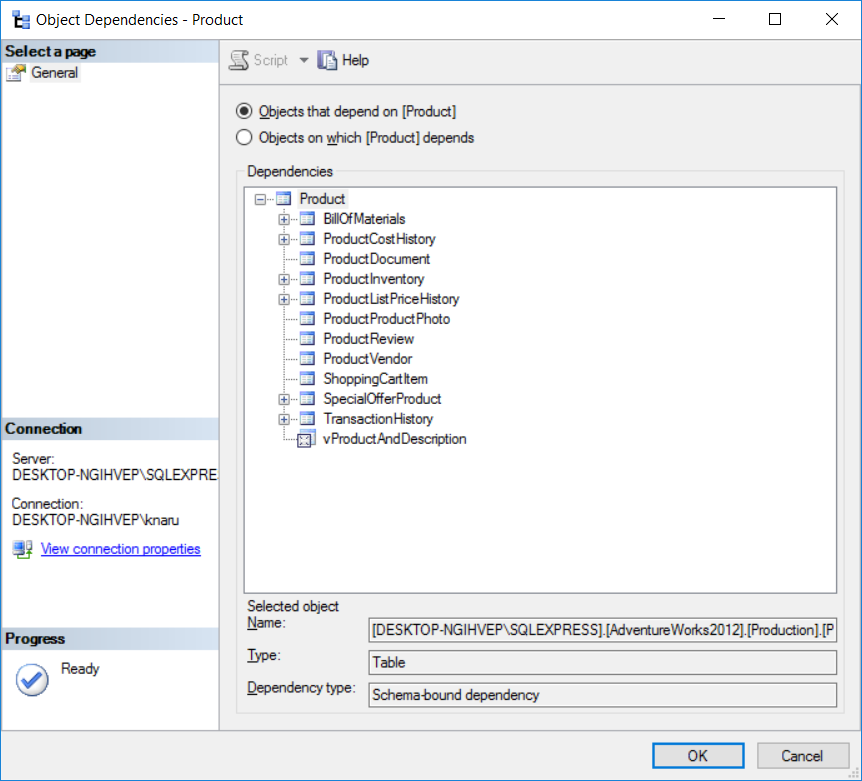
* **Edit** – The Edit makes reuse of the “New Product Entry” and the related technology which includes: HTML, JavaScript, and the MVC portion of the Server-Side functionality.
* **Delete/ “IsDeleted”** – this is a standard feature that can be implemented in a variety of ways. The following portion of the document describes this implementation “Delete” functionality and justification for the design.

To begin with, some difficulty exists with the Product table in that it has various other tables which are dependent upon it, which complicates deletion. The dependencies are depicted below in the Entity Relational Diagram.



When dependencies are this numerous there is a need to provide a design solution that carries the least impact upon an existing schema.

The problem is best stated like this:



Deleting a Product record requires an “order of deletion” routine and might disrupt other current or future applications such as a quotation process, invoicing, or auditing data. It’s also possible one could potentially “orphan” any dependent fields.

The solution to allow deletion used here was to create a new column (“IsDeleted”), provided by the SQL script which runs the following:

alter Table Production.Product

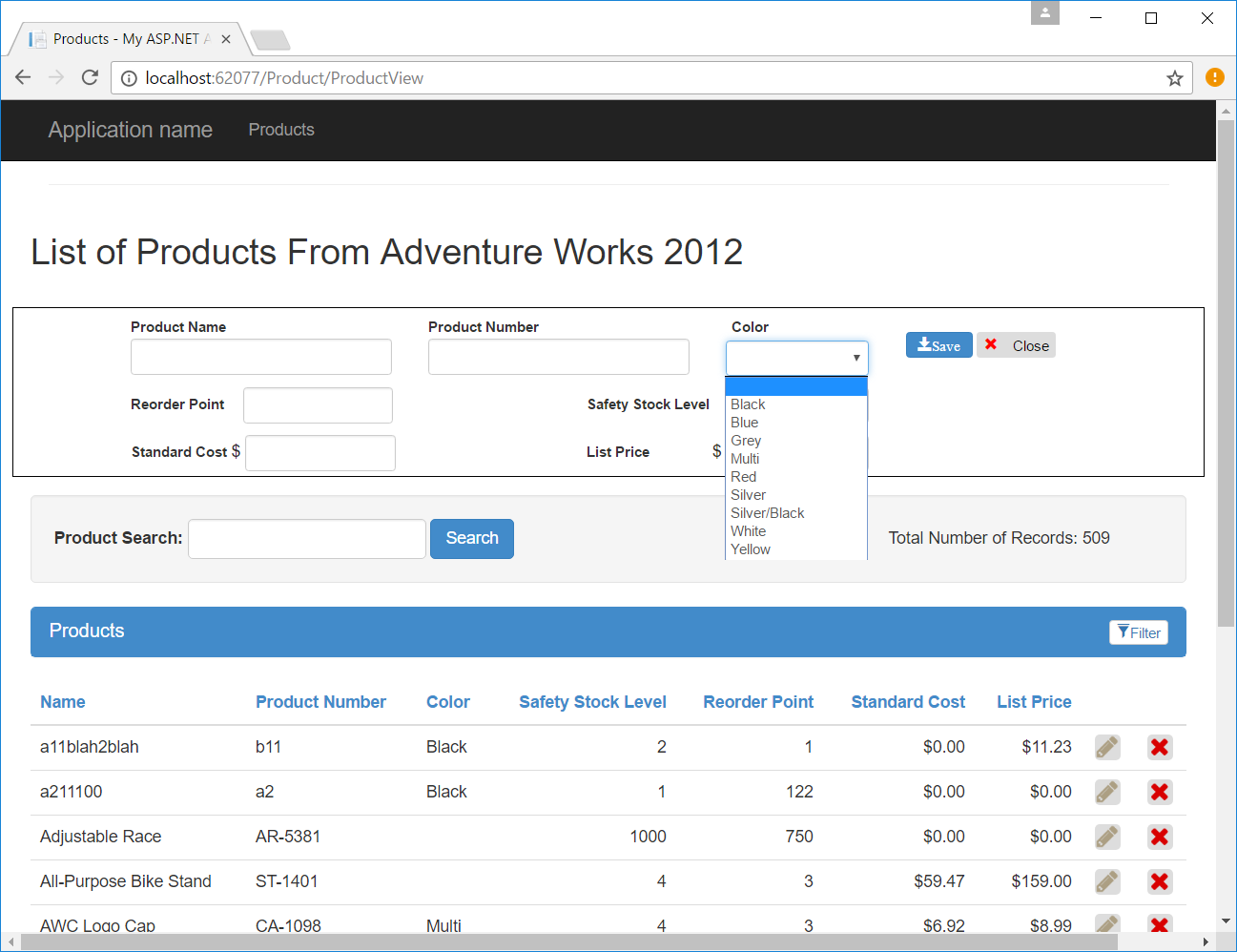
Add IsDeleted bit NOT NULL DEFAULT(0)

high dependencies.

The proposed change allows existing applications to continue without disruption and provide a relatively low-impact solution. Additionally, the change requires all database fetches to exclude records using the condition “when IsDeleted == true” in their fetch statements.

* **“Color” pull-down/ Reference Data Singleton** - Another concept was added regarding the “Color” field as it relates to how it is utilized for input. In the database, the Color field is a Nullable NVarchar(15). In this scenario, it is difficult to determine how the Product data may have gotten into the table hypothetically speaking, but one thing is clear: there are only so many colors listed in the Product table, as if certain products only come in so many colors.

Thus, an assumption can be made that the available of colors are fully represented in the existing data, in other words no other colors might exist besides what is there, but could be added through some other means. Consequently, it’s risky to allow the color field to be a free text entry where even a misspelling can introduce a new color, for example typing “Bleu” when “Blue” was intended. To reduce risk, fields like this would normally be a reference field that links to a Color Table where only so many colors can exist. A decision to allow this behavior was still possible by performing a one-time loading of “distinct” colors into a Singleton and providing an HTML <select> to be used by the Filter and New/Edit Entries (see below).



Although it might be worth it to refactor the database to accommodate a Color reference table, a simpler solution was possible to allow presentation of distinct colors based on the ones that are currently available in the Products table, thus eliminating the risk of mistyping, etc. Ideally, if there is a need to allow Colors to be entered, there should also be an “Administrative” page to manage that – in other words it should not be up to the “New Product” entry to introduce new colors of products. This is a good practice known as Separation of Concerns.

* **Data Binding** - Among the libraries that were introduced is AngularJS. Some investigation is necessary as some latency was discovered in using it for this demo. Additionally, KnockoutJS was included which offer similar two-way binding by creating “observable” objects. A decision would need to be made in the next sprint to continue using either of these. In any case, JQuery/Javascript handled the data-binding for the first sprint.
* **Test Driven Development** – With the exception of setting up the Entity Framework and generating the necessary model, the project was mostly begun by writing the tests for CRUD. As the design emerged from there, other tests evolved and were adapted. At the end of this sprint, currently 20 tests exist for Unit Testing.
* **Entity Framework** – A rapid development methodology was employed thanks to Entity Framework’s “schema-first” model generation. It allows a variety of Models to be generated (only Product Model in this case), maintaining all of its constraints if any exist and establishing relevant repository functionality that immediately connects to the database.

Commentary on Design Choices

Since the requirements lacked a true road map, concepts like implementing the filter were open to interpretation. For example, some implementers might have chosen a pull-down menu that filters on a single column at a time. It was also difficult to choose an implementation that demonstrates aptly for this exercise. Since the visibility of the 7 fields were high priority, it was determined that the filtering should be across all 7 columns as once. In any case, a design decision was made to satisfy the requirements in the best possible manner, which in hindsight, the goal of demonstration may have over-shadowed its usefulness.

Choosing to implement filtering as a real-time lookup for all fields raised immediate design issues which needed to be resolved, for better or worse, extemporaneously. For consistency, the numeric filtering was implemented with a “contains” rather than “begins with”, which upon further review neither choice seems terribly useful. The implementation that was used is a real-time search that filters records down from search results if they exist. A different option might be to implement this as a client-side operation so that searches don’t have to be re-executed or data doesn’t have to be sent back and forth to maintain its state.

Paging was a functionality that was completed quickly using the PagedList. However, once the design choice was made to favor Server-side, handling filtering and paging brought into question the wisdom of that choice. The low volume of data and the fact that the data only changes if acted on through the provided UI, fostered an extemporaneous decision to “switch” between Filter and Sorting, as pictured below.

|  |  |
| --- | --- |
|  |  |

This in turn seemed to naturally fashion the following usability rules:

1. The search begins anew covering the entire data set.
2. Filtering is secondary but maintains the data set that was searched.
3. Currently clearing the filters will not disrupt the search item.

Removing the filters will reset the page to its former state which is how “Search” might have been presenting the data. Since there was no specific requirement for this and the filtering was done using a partial page refresh, it would need to be refactored if the functionality was that filtering can be switched and paging will page through search items and filtered.

Next Steps

An issues list, database, and the Source code ready to be loaded into Visual Studio are all provided on a public unrestricted repository, which can be found at <https://github.com/knarushoff/UPMC.eBenefits>

The issues list is essentially a product backlog from which a developer/development team would vote on its viability and choose the next priority to begin its implementation. Among the items on the list, some known issues are:

* Upon collaborating with stakeholders, it is conceivable that one possible “to do” is allowing Sorting to be available for whatever data set is currently available via search and filtering.
* Currently, if it’s determined to be useful, implementation of a true “Contains” search of the dollar amounts like that of List Price and StandardCost, would need to be compared against the 2-decimal rounded number and not the full number (this was realized when numbers showed up from a search but didn’t to contain the “filter” number. When in reality the matching digit was hidden because of rounding on the display).
* Consider revising the “Contains” filtering or replacing it with another style such as “Begins with” or something else altogether.
* Investigation: consider an Inversion of Control framework such as Ninject or Autofac. The Autofac library and its preliminary configuration were included in this implementation. However, the project in its current state did not warrant the excessive layers of abstraction that might be introduced – at this point it would seem to be over-engineered.
* Consider implementing a more sophisticated repository pattern. Among the drawbacks to this are introduction of unneeded layers of abstraction. Also when “schema-first” model generation is needed for additional execution (if other tables are later introduced), then the necessary “interface” class will be over-written.
* Test using Selenium or Karma
* Clean up Javascript Code and CSS – remove Javascript functionality that present in the “View” page and instead place into the provided CSS or JS files for modularity and better organization of code.
* Refactor to more fully AngularJS, KnockoutJS or React application.
* Consider tests for preparing the Model used by the View.
* Investigation: Consider latency - sorting as an AJAX function

Instructions:

1. Run the sql script: Product.sql

USE [AdventureWorks2012]

GO

SET ANSI\_NULLS ON

GO

alter Table Production.Product Add IsDeleted bit NOT NULL DEFAULT(0)

GO

1. Alter the Connection string in App.config and web.config to an appropriate connection string.
2. The full deployment can be obtained from <https://github.com/knarushoff/UPMC.eBenefits>