Spark 4.5

Application Platform

Companion document to

Design Pattern Framework™ 4.5

by

Data & Object Factory, LLC

www.dofactory.com

Index

Index 2
Introduction 6
Document 6
What is Spark 4.5 7
Overview 7
Selecting data 7
Ad-hoc queries 9
Insert, update, and delete 9
MicroORM 10
Spark Features 11
Full stack 11
Prescriptive 11
Simple 11
Light-weight 12
Agile 12
Rapid Application Development (RAD) 12
Architecture 12
Design Patterns 15
Practices 15
Tools 16
Conclusion 16
What is Art Shop 16
Art Shop Walkthrough 16
Product Details 21
Shopping Cart and Checkout 22
Authentication and Authorization 23
Administration 27
Reports 29
Dashboard 32

Ad Hoc 32
REST 34
Solution and projects 35
Project: Art.Web 36
Project: Art.Domain 38
Project: Art.Rest.v1 40
Project: Art.Rest.v1.Text 40
Code Review 41
Database 41
Domain Generator 47
Data Layer 49
The Data Layer in detail: Db class 49
Domain Layer 51
The Domain Layer in detail: Entity class 56
Repository Layer 57
Context 59
Unit of Work 60
Building Domain projects with Spark 4.5 62
Multiple databases 62
Presentation Layer 63
App_Data and App_Start 63
Areas 64
Code Folder 69
Content Folder 72
Scripts Folder 72
Views Folder 72
Global.asax.cs 72
Web.Config 72
REST Layer 73
Design Patterns 77
Active Record 78
Repository 78

Unit of Work 78
Façade 79
DTO 79
CQRS 79
Miscellaneous Patterns 80
MV* Patterns 80
GOF Patterns 80
Enterprise Patterns 81
Best Practices 81
Convention over Configuration 82
Constraints over Lookups 82
Caching 83
JSON over XML 84
Tools and Components 84
Tools 84
Automapper 85
Json.NET 85
Bootstrap 85
jQuery 86
jQuery UI 86
History.js 86
Flot 86
Components 86
Caching 87
Custom Principals 87
Service layer 88
Error logging 88
JavaScript and jQuery 89
Building apps with Spark 4.5 90
Solution and Projects 90
Creating the Database 93
Building the Web project 95

Building the Domain project 96
Hand-coding the Domain project 98
Building the REST project 99
Hand-coding the REST project 101
Building the REST Test project 101
Summary 104

Introduction

This document describes *Spark 4.5* and its *Art Shop* reference application. Spark is a light-weight, pattern-based platform for rapid application development (RAD). It provides and enjoyable and productive developer experience that is centered around simple conventions and a small set of proven design patterns. Creating apps with Spark is quick and easy and the resulting applications you built with it are beautiful and fast.

The *Art Shop* is an MVC application built on the Spark platform. It is an online store that sells art reproductions of famous classic artists, such as Van Gogh, Cezanne, and Vermeer. It features elements that are common in modern online stores: a product catalog that the users can browse, search, sort, filter, and page through; a shopping cart with payment and checkout; an authentication and authorization system; and a secure administration area with reporting, charting, and order management, as well as an ad-hoc reporting page.

Spark 4.5 addresses the needs of .NET developers by making you highly productive. Many projects are under tight deadlines and users are expecting high quality applications. As a developer this leaves you with little room for experimentation or mistakes; you have to get the architecture and design right from the start. Spark 4.5 offers a clear path with a great architecture and proven patterns and best practices.

Document

There are 8 sections in this document:

What is Spark 4.5: Here we dive right into the code by showing you some Spark data access examples. This is followed by a review of the design principles that underlie the Spark platform.

What is Art Shop: This section will walk you through the Art Shop reference application. This should give you a sense of what is possible with Spark. You will see the shop's user interface, the shopping cart as well as a large administrative area which is accessible by administrators only.

Solution and Projects: This section reviews the Art Shop solution and projects in Visual Studio 2012. You will it has only two projects. And two more to implement a REST interface.

Code Review: Here we explore the code and analyze the four layers in detail. It starts with the database moving up through the data and domain layers to the presentation layer. We will also discuss the REST implementation.

Design Patterns: In this section we will review each of the core patterns of Spark. They are DTO, Façade, Repository, Unit of Work, Active Record, Lazy Load, and CQRS.

Best Practices: Spark follows the Convention over Configuration (CoC) paradigm. In this section, you will learn what this means and how it affects the design and architecture of Spark apps. Two additional best practices are mentioned as well.

Tools and Components: In .NET development, open-source tools and libraries cannot be ignored; they are just too valuable and useful. Spark uses some of these including AutoMapper, Bootstrap, and

jQuery. Spark components that are discussed here include Caching, Error logging, and custom Principals.

Building apps with Spark: Once you are familiar with Spark and you are ready to apply Spark techniques to your own projects, this section will show you how to get started.

Without further ado, let's get started.

What is Spark 4.5

Overview

Let's start off with some Spark code. This will give you immediately a feel of its ease-of-use and its flexibility.

Very important to Spark is the Active Record design pattern. This pattern states that for each table in the database there is a corresponding class (called a domain object). At runtime, a domain object instance corresponds with a record in the table. The convention in Spark is that the table and the class names are the same. Furthermore, column names are the same as the property names.

Let's look at an example. Suppose the database has a Car table with columns Id, Model, and Make. In Spark you will have a corresponding Car class with properties named Id, Model, and Make. What's happening is that the data model and the object model are the same, essentially solving the 'impedance mismatch' problem which has hampered developers and slowed projects for a long time.

Let's look at selecting data first. Note that the code samples are in C#, but VB developers will follow along as they are nearly the same.

Selecting data

The Art Shop database has a table called *User*. To retrieve a user by Id you simply pass the value to the Single method.

```
var user = ArtContext.Users.Single(id);
```

This returns a User domain object. ArtContext represents a database connection to a database named Art. Users is the User repository. Single is a method on the Users repository which returns a single domain object.

As an aside: ArtContext can be shortened (to *db* for example) with an alias which would make data access expressions even more concise, like so:

```
var user = db.Users.Single(id);
```

The alias option is explained later in this document, but it has not been used in the Art Shop.

To retrieve all users from the database you use the All method:

```
var users = ArtContext.Users.All();
```

The Art Shop database has another table named *Product*. To get all products you use a different repository name, like so:

```
var products = ArtContext.Products.All();
```

The All method has several optional parameters. For example, to get all users in the USA you add a *where* snippet with a parameter, like so:

```
var users = ArtContext.Users.All(where: "Country=@0", parms: "USA" );
```

The *where:* and *parms:* are named parameters. Working with parameters has the benefit that it prevents Sql Injection.

Sorting is easy. To get all users sorted by email you use an *orderBy* snippet:

```
var users = ArtContext.Users.All(orderBy: "Email Asc");
```

Where statements with multiple parameters are issued like this:

```
var where = "Country=@0 OR LastName=@1";
var parms = new object[] { "USA", "Smith" };

var users = ArtContext.Users.All(where: where, parms: parms);
```

As you can see, Spark gives you the full flexibility of Sql, but without the need to write lengthy Sql statements.

Pagination is common in applications and the Paged method makes it easy. Here is how to select page 2 of a list of users with 20 records per page.

```
int totalRows;
var users = ArtContext.Users.Paged(out totalRows, page: 2: pageSize: 20);
```


The totalRows *out* variable will have the total number of users meeting the criteria. It is used to display the proper pager controls on the View.

The Paged method supports the same parameters that are available in the All method. Here is an example in which all parameters are provided (note: the named parameters are used for readability, but they are optional).

```
int totalRows;
var users = ArtContext.Users.Paged(out totalRows,
                                   where: "OrderCount BETWEEN @0 AND @1",
                                   orderBy: "OrderCount desc",
                                   page: 1, pageSize: 15,
                                   parms: new object[] { 2, 10 });
```

The methods Single, All, and Paged return domain objects which have all their property values set. In other words, they represent complete table records.

If you need fewer columns, or you need more control, such as SQL joins, use ad-hoc queries. They are explained next.

Ad-hoc queries

The Query method accepts any Sql select statement and returns a collection of .NET *dynamic* objects. Parameters are optional. Here is a parameterless query that also shows how to access the Country and Number properties.

```
string sql = @"SELECT Country, COUNT(Id) AS Number
               FROM [User]
               GROUP BY Country
               ORDER BY Number DESC";

var countries = ArtContext.Query(sql);

foreach (var country in countries)
{
    string name = country.Country;
    int number = country.Number;
}
```

Similarly non-query commands can be executed with the Execute method (non-query commands return no values):

```
string sql = "DELETE [User] WHERE TotalOrders=0";
ArtContext.Execute(sql);
```

Insert, update, and delete

The Art Shop has an Artist table and here is how a new artist is inserted.

```
ArtContext.Artists.Insert(artist);
```

The artist argument is an Artist domain object (business object). Following the Insert it will have the newly created Id value.

Updating an artist is as simple as this:

```
ArtContext.Artists.Update(artist);
```

And deleting works the same:

```
ArtContext.Artists.Delete(artist);
```

In summary, data access in Spark is simple and close to the metal, meaning it is close to the SQL that is sent to the database. Yet, the amount of Sql you write is minimal with just a few '*where*' and '*orderBy*' snippets and optional parameter values.

MicroORM

If you are familiar with MicroORMs, such as Dapper and Massive, you may have thought there are similarities with Spark. And you would be correct, because Spark was inspired by other MicroORMs. However, Spark's ORM implementation is better (we think). In addition, Spark is more than a MicroORM. It also offers guidance, patterns, and practices to the entire application stack of any .NET application.

Data access in Spark is extremely fast and performance-wise Spark beats the Entity Framework hands-down. Unlike Entity Framework, Spark doesn't have to go through numerous parsing and mapping layers: it sends the Sql straight to the database. Data retrieval is highly optimized: records are retrieved with the *firehose* DbDataReader and the resulting domain objects are streamed back immediately to the caller (using *yield return*) without the need to store these in a list of similar structure.

An apples-to-apples comparison between Spark and Entity Framework is not currently available, but here is an article that comes close: https://code.google.com/p/dapper-dot-net/. This is a study which compares similar MicroORMs, including Dapper, Massive, and PetaPoco to the Entity Framework.

The results are impressive: with 500 select mappings (a select query returning a business object) they are roughly a factor 10 faster than the Entity Framework, that is, they take only 1/10th of the time! Performance in Spark will be in the same ballpark.

Although similar to other MicroORMs, Spark has some distinct advantages. The most important ones are: 1) type-safety and 2) minimal SQL coding. Massive is built around .NET *dynamic* objects meaning

their queries return dynamic .NET objects whereas Spark's domain objects are type-safe. Type-safety is quite important because it allows Spark 1) to detect certain errors at compile time and 2) Intellisense works as expected (the data objects in Massive are dynamic and Intellisense is not available).

Both Dapper and PetaPoco require that you write entire SQL statements which can be rather tedious. In the examples you've seen that in Spark you only need small 'where' and 'orderBy' snippets while still maintaining the full flexibility of the SQL syntax.

As mentioned, Spark is more than a MicroORM. It is a comprehensive platform that includes guidance on architecture, patterns, practices, and tools for the entire application stack. It goes beyond the data access by including architectural patterns such as Repository, Unit of Work, Caching, Lazy loading, CQRS and also the use of JavaScript and jQuery.

Now that we have seen some code and compared Spark to some other MicroORMs let look at what Spark really is.

Spark Features

So here is a definition of Spark: Spark is a full-stack, prescriptive, simple, light-weight, agile, rapid application development (RAD) platform based on a proven architecture with patterns, practices and tools. That's a mouth full. Let's parse this sentence and review each of the characteristics, one at a time.

Full stack

Full-stack means it defines the entire stack, that is, all layers in the application, starting with the data model in the database all the way up to the presentation layer. Spark specifies the layers, their roles, their interactions, and the patterns and practices used to build these. It is more than a concept; it includes actual code that makes it all possible.

Prescriptive

Spark is prescriptive, meaning it prescribes the architecture, the patterns, and numerous other code decisions. Another word for prescriptive is *opiniated*. Opiniated software comes with thoughts about how to best build an app. It may sound restrictive, but in reality it is refreshing. The prescriptions and opinions are nothing more than conventions, guidelines, and best practices.

With a clear set of guidelines, developers don’t have to worry about making the right coding decisions. Instead, they can focus on the real task: building great apps quickly and effectively. By following the conventions, developers will be faster and more effective. One of the goals of Spark is to make development frictionless, fast and simple.

Simple

Simple does not mean simplistic. The conventions used in Spark are based on patterns and best-practices which help in making the code easy-to-understand and easy-to-use. At the same time, Spark is elegant and powerful allowing you to build modern systems that perform well and delight the users.

Light-weight

Spark is light-weight. The number of lines of the code varies with each application, but on average it is somewhere between 1500 - 2000 lines. Note: foundational code is the core-engine that drives the entire platform. This is remarkably small if you consider the functionality and flexibility it offers (you will see this a bit later in the document).

Agile

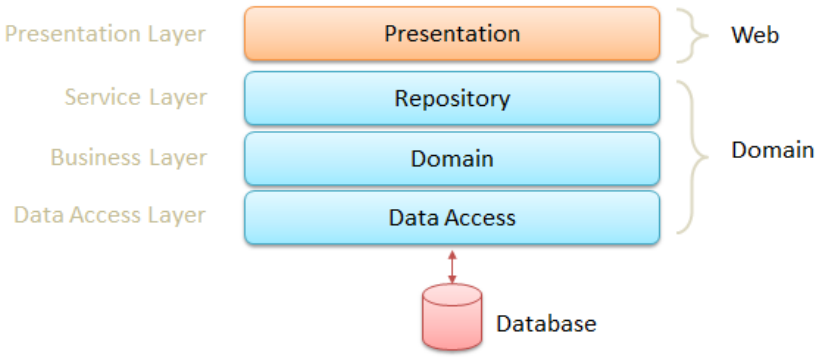
Most projects today follow some variety of Agile methodology. Agile development allows developers to focus on what is most important, which is delivering great apps quickly and effectively. This is exactly in line with the objectives of the Spark platform.

Rapid Application Development (RAD)

Spark applications are developed much faster by using a proven architecture and a limited set of patterns, practices, and tools. The Spark code base is lightweight and very small (about 1500 - 2000 lines depending on the model), but it allows for effective and rapid development practices.

Architecture

Spark is based on a proven pattern-based architecture. You will not be surprised to hear that it is same 4 layers we presented in *Patterns in Action 4.5* although they are named and organized slightly differently. The entire solution with 4 layers is spread over just 2 projects in Visual Studio. Below is a diagram of this setup:



On the left are the layers as presented in *Patterns in Action 4.5*. In Spark, instead of Service and Business layer, we have Repository and Domain layers. The differences are minimal. The Repository layer is a Service layer in which the Service is partitioned in separate Repositories, that is, each business object has its own dedicated 'service' which is called a Repository. The Business layer is exactly the same, but is now called Domain layer.

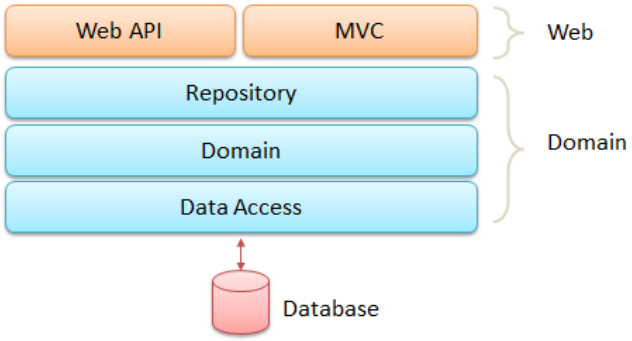
On the right in the diagram are the two Visual Studio projects in which these layers reside: Web and Domain. The Web project contains the Presentation layer and the Domain project contains the Repository, Domain, and Data layers.

Developers that use the Spark platform spend most of their time in the Web project. If you think about it, this really is the preferred place because your application is unique and this is where you express that uniqueness. Issues such as data access, domain object creation, and transaction management are more mundane in nature and should be automated whenever possible.

The conventions in Spark make it possible to automatically code generate the entire Domain project with *all* three layers. We offer a separate product called *PRO Design Pattern Framework 4.5* which does exactly that: it reads the tables in the database and then automatically generates the Domain project with Repository, Domain, and Data Access layers in just a matter of seconds. This includes all Domain objects and their repositories. No coding is involved at all. The PRO edition comes with two code-generators: one that creates the Domain project and another code-generator that creates a REST project.

Having said this, the *PRO Spark 4.5* code generator is only a convenience -- albeit an amazing one. There is nothing that prevents you from coding your own Spark-based apps by hand, just like you normally would. Part of the Spark platform is foundational code, that is, reusable code that does not require any change and is ready to be used. What the PRO edition offers is the automatic creation of the domain and associated repository objects which can be rather tedious. An advantage of PRO over hand-coding is that the ability to automatically update the code when database changes are made. If this is not clear, don't worry, we will get into more detail later in this document.

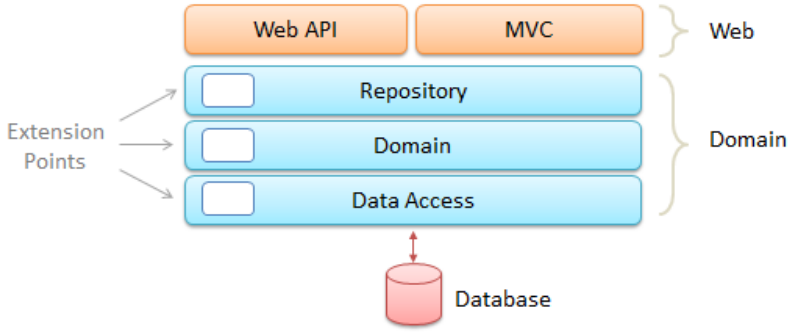
Art Shop is the reference application built on Spark. The diagram below depicts the layered structure in this application. The presentation layer in Art Shop uses both Web API and MVC. Web API is used to build the REST interface, and MVC to build the actual Web application.



As mentioned, the bottom three layers are either fully hand coded or code generated. In either case, each layer has extension points in which you can customize functionality for the application you're working on. For example you can add a custom method to the Users repository, such as GetByEmail

which retrieves a User domain object by email. Or, you can adjust the Sql just before it is sent to the database. This is mostly implemented through partial classes. Please note that these extensions are optional and in many cases they are not used. Once you see the actual code this will all become clear.

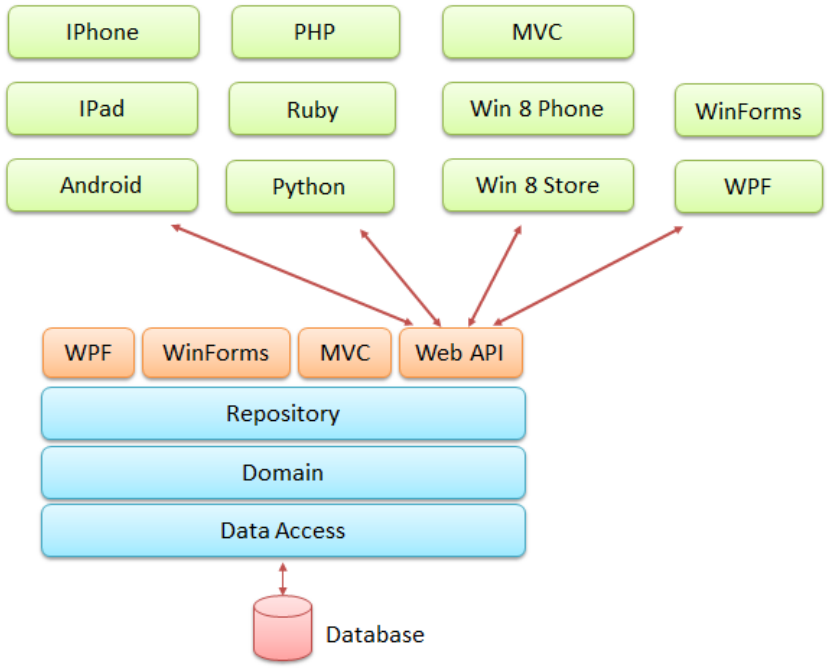
The diagram below shows these extension points as 'holes' in the layers in which you can add your custom code snippets.



As an aside, these extension points are very much like the GoF Template pattern.

Spark includes numerous patterns and components, such as a Cache, CQRS, Transaction Management, OAuth, and JavaScript / jQuery code. Spark also helps with the construction of a Web API REST interface to allow external clients to reach the system. As you go through this document you will discover the broad scope of Spark as a robust application platform.

Spark is not limited to MVC and Web API. It can be used by any client technology, including WebForms, WinForms, and WPF. In fact, when building a REST interface (with the Web API) you open the application up to the rest of the world, both Microsoft and non-Microsoft client technologies; the possibilities are endless as you can see in the diagram below:



Design Patterns

Spark uses numerous patterns. You know by now that patterns are proven solutions to common software problems. Two decades of real-world pattern experience has shown that some patterns are more useful than others. Spark incorporates only the most useful patterns that offer a clear benefit to the application. Core patterns include Façade, Repository, Unit of Work, DTO, Active Record, and CQRS. You may or may not be familiar with these, but they are explained later in this document. Of course, other patterns are used on an 'as needed' basis. In the Art Shop app you'll find several other patterns used as well.

Practices

A best practice is an approach to software development that is proven to be useful. A core best practice in Spark is *Convention over Configuration* or CoC for short. CoC simplifies and standardizes software development. Research has shown that 70% of software projects are too late, over budget, or get cancelled altogether. The money lost is staggering. CoC brings a level of common-sense and simplicity to development.

Convention over Configuration originated in Ruby on Rails, but is now common in the .NET world, particularly MVC which has many similarities to Rails. Other terms related to CoC are *lean* development, *RAD* (Rapid Application Development), *YAGNI* (You Aren't Gonna Need It), and *KISS* (Keep It Simple Stupid). All of these express a common desire for simplification. The best way to achieve this in software development is with a set of well-designed and well-defined conventions and a platform, which is exactly what *Spark 4.5* offers.

Tools

Spark 4.5 uses several open source tools and libraries. Nuget makes managing third party tools very easy. They are:

- Bootstrap
- jQuery and jQuery UI
- History.js
- Automapper
- Json.NET
- Flot

Nothing prevents you from adding additional tools and libraries. For example, you may want to consider adding a JavaScript MVC library, such as Backbone, Angular, or Knockout. They integrate well with Spark and it makes sense to add these to bring structure to the UI as you add more and more JavaScript and jQuery to your own projects. Unfortunately, the learning curve for the JavaScript MVC libraries is fairly steep and their use is outside the scope of the Spark platform. Again, they are easy to integrate with Spark.

Conclusion

This completes our 30,000 feet overview of Spark 4.5. In summary, Spark is a lightweight application platform designed to allow .NET developers to build apps quickly and easily. The *Design Pattern Framework 4.5* package includes the entire Spark 4.5 platform with 100% source code, so you can immediately start building your own projects with this powerful tool.

A separate product offering called *PRO Design Pattern Framework 4.5* is available for purchase on our website at www.dofactory.com. It includes Code Generators for the Domain and REST projects which allow you to develop you apps up to 4 times faster. Check it out!

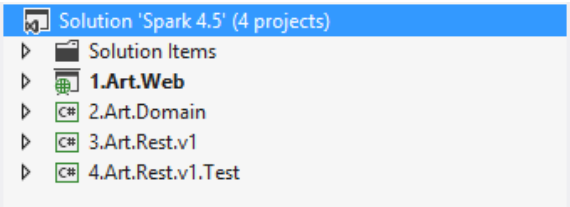
What is Art Shop

Art Shop is a reference application that demonstrates the capabilities of Spark. It is a modern, real-world online store where users can purchase art reproductions. It includes a store front with shopping cart, checkout pages, an authentication system, and a large administrative area that allows administrators to perform maintenance and reporting functions.

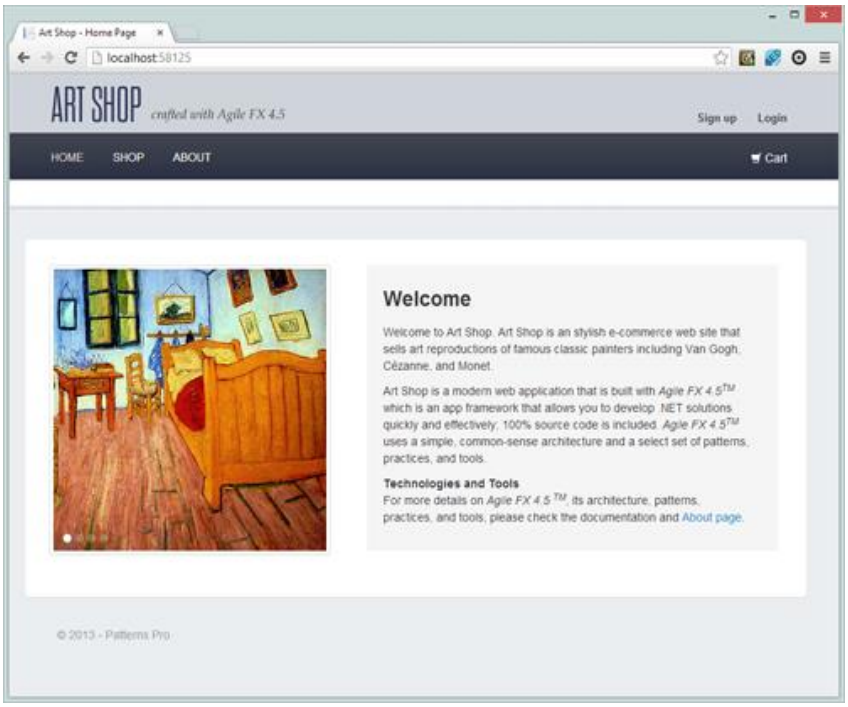
Before getting into the technical aspects, it is important that you understand the features and functionality of the Art Shop. This will allow you to better appreciate the Spark architecture, patterns, practices and tools that went into building the application. So, let's walk through the running application and see what it offers.

Art Shop Walkthrough

Open the Spark 4.5 solution. There are 4 projects.

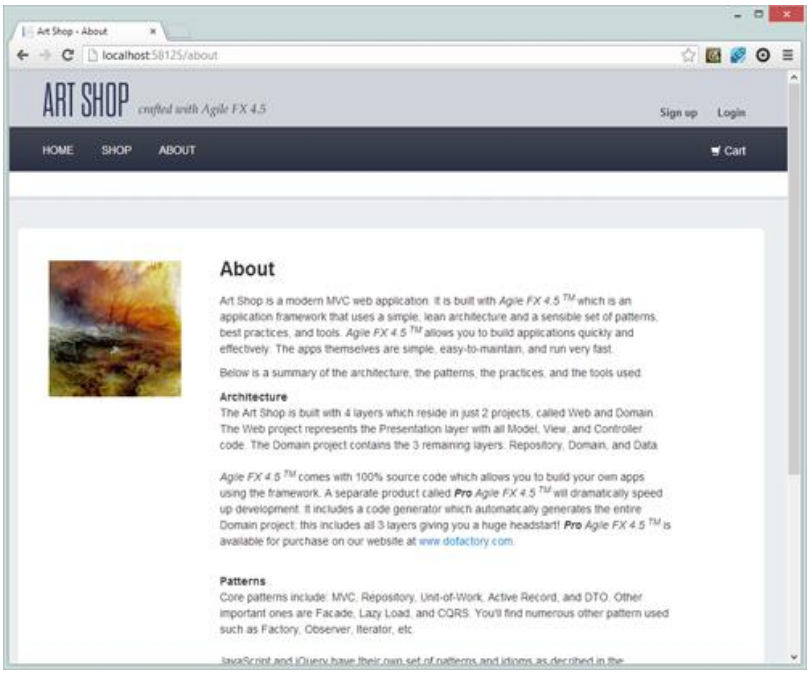


For now, ensure that the Pro.Web project is the startup project (i.e. shows in bold) and select Run. The application will launch and the home page will open.



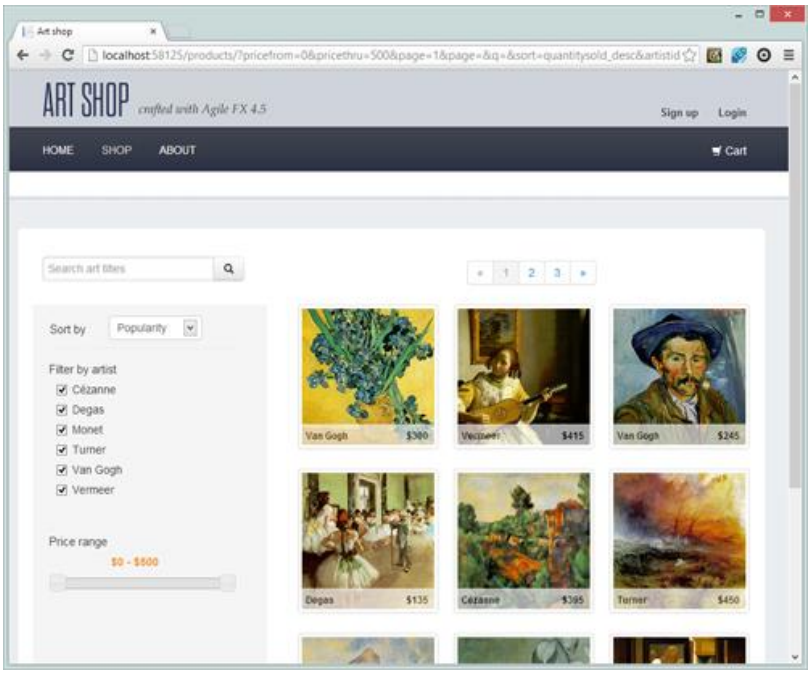
The home page contains a carousel control in which four different art images rotate every 4 seconds. By the way, the carousel lazy loads 3 out of the 4 images (Lazy load is a design pattern). This will allow the page to render quickly and once it is up and running the remaining 3 images are loaded. This is implemented with JavaScript and jQuery located on the page.

Read the welcome message on the page and also visit the About page. They will give you a high level introduction of the architecture, the tools and the technologies used to build this app.



The black menu bar along the top has 3 menu items and a cart status control to the right. Signup and login links are accessible on the top right. Clicking on the Art Shop logo will bring you back to the home page.

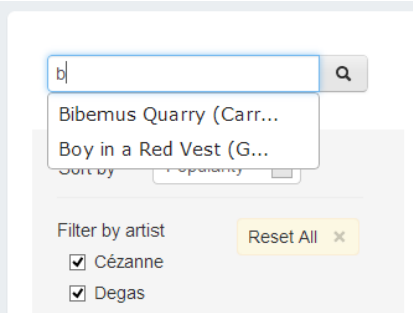
Click the Shop menu item to start shopping for art.



You are now in the art shop where you can search and browse art reproductions. This page features all the common UI list operations including: Search, Sort, Filter, and Pagination. By the way, from a programming perspective this page is one of the more interesting and rich ones in the app.

The Art Shop was designed for ease of use. With the help of JavaScript, jQuery, and Ajax a wonderful user experience is created. The page renders very smoothly and quickly. Whenever the value of a control is changed, the list with art images will be updated immediately, but without page flicker. Let's review some of the controls.

We'll start with search. Search is built with the autocomplete feature in which suggestions display as you type. This particular database has only 24 product items (art reproductions) which limits the effectiveness of this demo. Enter the letter 'b' and notice that two suggestions appear in the dropdown. Press the search button and only those two titles that start with a 'b' will display.



Anytime you search, sort, filter, or page through the art work, a beige "Reset All" button will display (see image above). Click the button and this will reset all search and filter criteria. Again, due to the limited number of art items in the database the true power of the search cannot be shown.

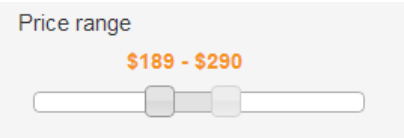
Click Reset All and the focus on sorting. The sort drop down offers 3 options: Popularity, Rating and Price.

Popularity is how many items have been sold for each product. *Rating* is a 0-to-5 star rating; users can provide feedback and rate a product (not implemented in the app) and each product item maintains its average star rating. The *Price* is the price for the products which ranges from $125 to $495.

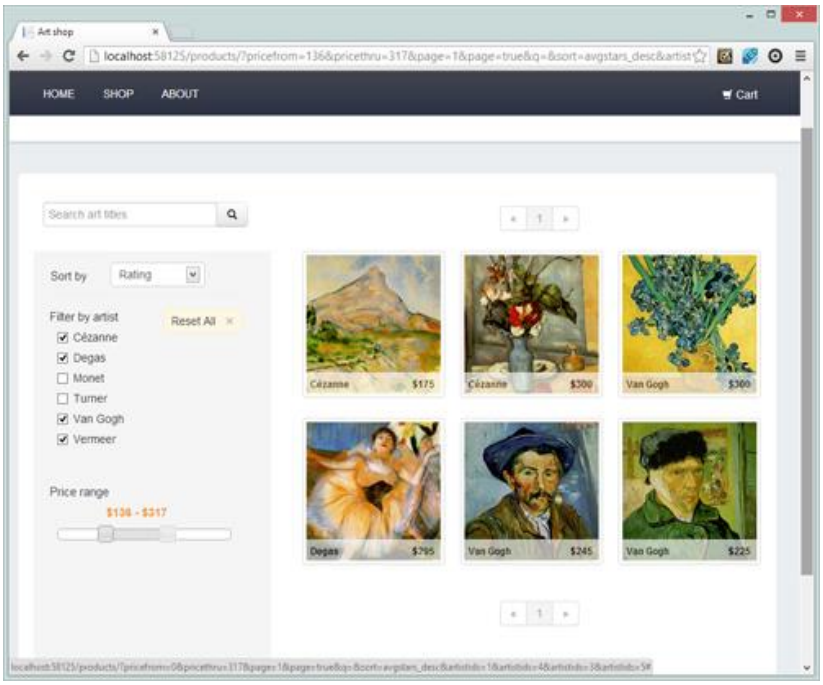
Select a sort item and notice the speed at which the page responds. There are no full page refreshes; only the art work is refreshed via an Ajax call to the server.

This page supports two filters. You can filter by artist, of which there are 6, and you can filter by price using a range control. Check or uncheck any of the artists and their work will be included or excluded depending on the checkbox setting. Unchecking all will result in no artwork displayed. Again notice how snappy the response is.

The price range slider lets you filter by price.



Grab the right hand side slider and bring it down so say, $300. The list of art items will be reduced. Tighten the range even further by sliding the left slider to the middle. Only a few paintings will fall within that narrow price range. The image below shows that only 6 items meet the artist and price filter criteria.



Pagination is the next UI action to explore. Start off by selecting 'Reset All' (or click the Shop menu item) so that all products display. There are two pager controls: one above and one below the art images. The number of products per page is 9, so with a total of 24 products there are 3 pages (the last one only with 7 products).

Click on each page number and again notice the fast page response. Now apply some filters and notice how the pager changes from 3 pages, to 2 pages, and, if the filters are very tight, just 1 page.

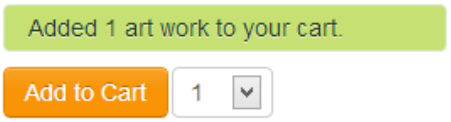
One more interesting fact related to pagination. To see this, first click Reset All (or select the Shop menu item). You will see three product pages. Select the 3rd page. Now, whenever you apply a Search, a Sort, or a Filter action, the current page always jumps back to 1. This is necessary because the pagination system needs to adjust to the new arrangement (in case of sort) or the new number of items in the list (in case of a filter or search).

In summary, all the browse and search actions are Ajax based which delivers a wonderful, highly interactive user experience.

Product Details

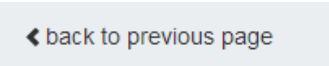
Suppose you find an art piece that you like. To see more information click on the item and you are taken to the product detail page that has a large image, title, star rating, and information about the artist.

Here you can select the number of items you'd like to purchase in a small dropdown. By selecting the quantity and clicking on the Add to Cart button these items are added to your cart (without going to the shopping cart page).



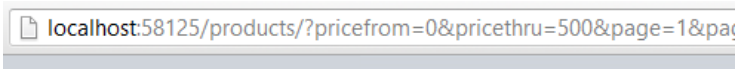
The cart is immediately updated which is clear from the orange cart control in the menu bar and the green confirmation message. On the background a new cart is created and the items are added with an Ajax call to the server.

At this point you can visit the cart by clicking the orange button or go back to the shop by selecting the back link at the top left.



There is something interesting taking place when going back to the shop. When selecting the 'back to previous page' link, you will notice that the original state of the shop page is restored, that is, the search, sort, filter, and page selections are all restored to the state in which you left it. This creates a very natural experience for the users because they are going back to the same place they came from.

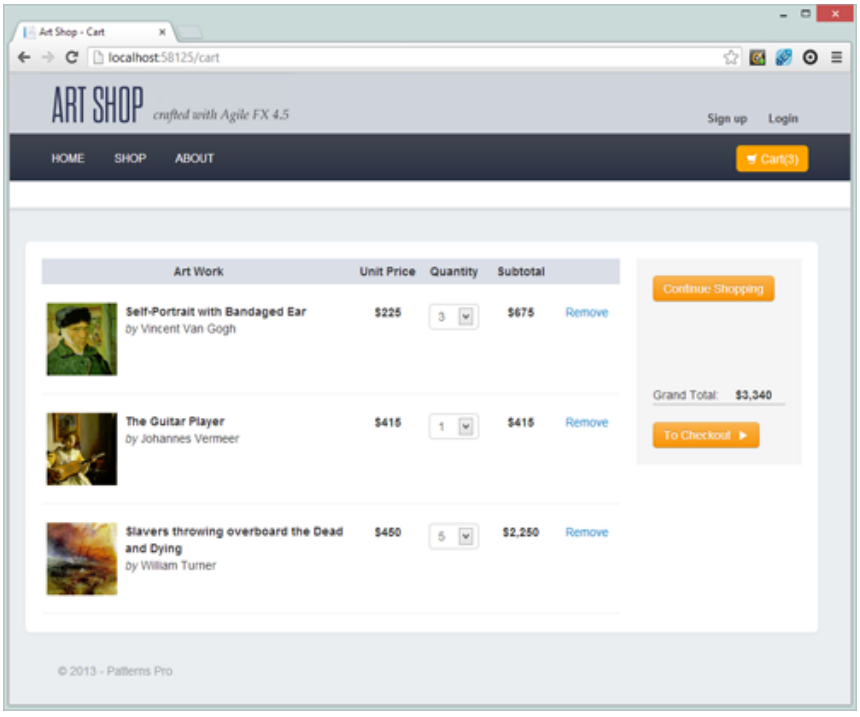
This 'magic' is coded in JavaScript which updates the browser history each time a search, filter, sort, or paginate event occurs. You can actually see this in the command line where url changes take place when you perform sorts, filters, etc.



Manipulating the browser history is explained in detail later in this document. Remember that in master-detail scenarios updating browser history makes sense. However, if there is no detail page, then there is no need to manage the browser history.

Shopping Cart and Checkout

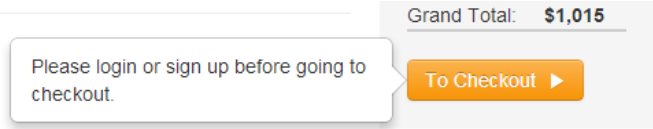
Add a couple more art pieces to your shopping cart. Then click on the orange cart control in the menu bar to see your shopping cart.



The shopping cart page is where you can adjust quantities and/or remove line items entirely from the cart. Change the quantity of an item and notice that the Grand Total reflects that change by fading out and then back in. The change takes effect immediately, there is no page refresh; all courtesy of Ajax which does all the heavy lifting in the background. Click a 'remove' link and the line item will be removed from the cart. Again, all totals are immediately updated without page refresh, including the cart control when the cart is emptied out.

An interesting pattern called CQRS is at play here. It is an optimistic pattern which sends a simple command to the server and then assumes that the rest of the transaction (cart database change) will be handled successfully without further checking for the results. We will explain CQRS later in this document.

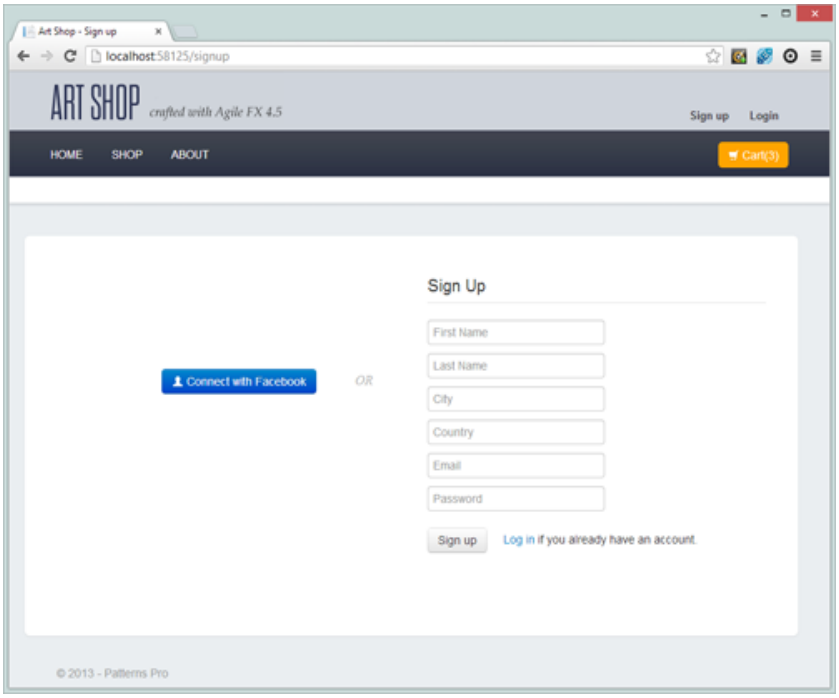
Select checkout and you'll see that you need to authenticate yourself before being able to checkout.



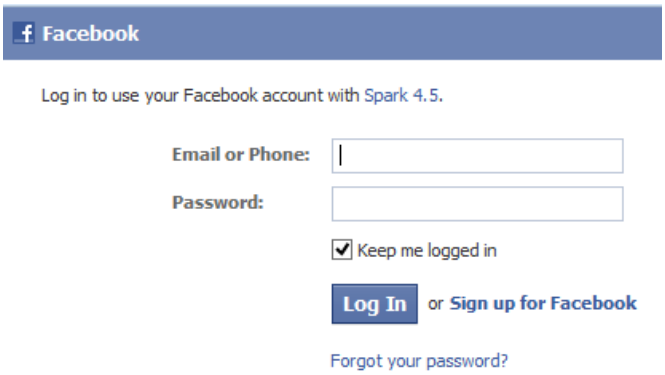
So, next let's review the user signup and login pages.

Authentication and Authorization

On the top right of the page you see two links: signup and login. Select sign up and you will have the opportunity to either sign up with Facebook (using OAuth) or create a local account stored in the local membership database.



Selecting 'connect with Facebook' allows you to use your Facebook account. Please be aware that to make this work you need to create an app on Facebook. This is what the Facebook login looks like:

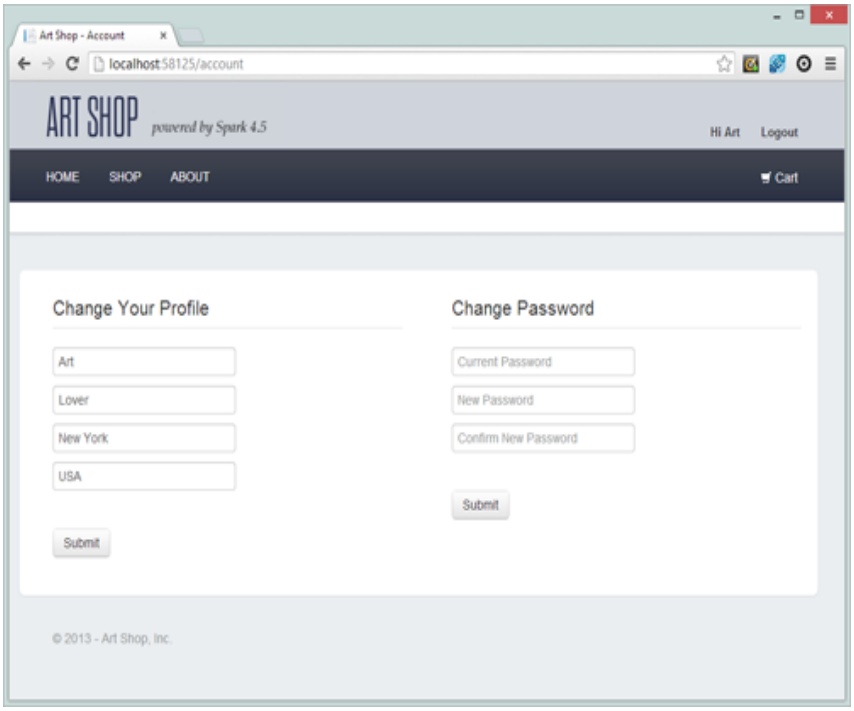


After you login to your Facebook account it will ask you for permission to send your personal data, email, and location. Approve and you will be redirected back to Art Shop where your account is created and you'll be logged in automatically.

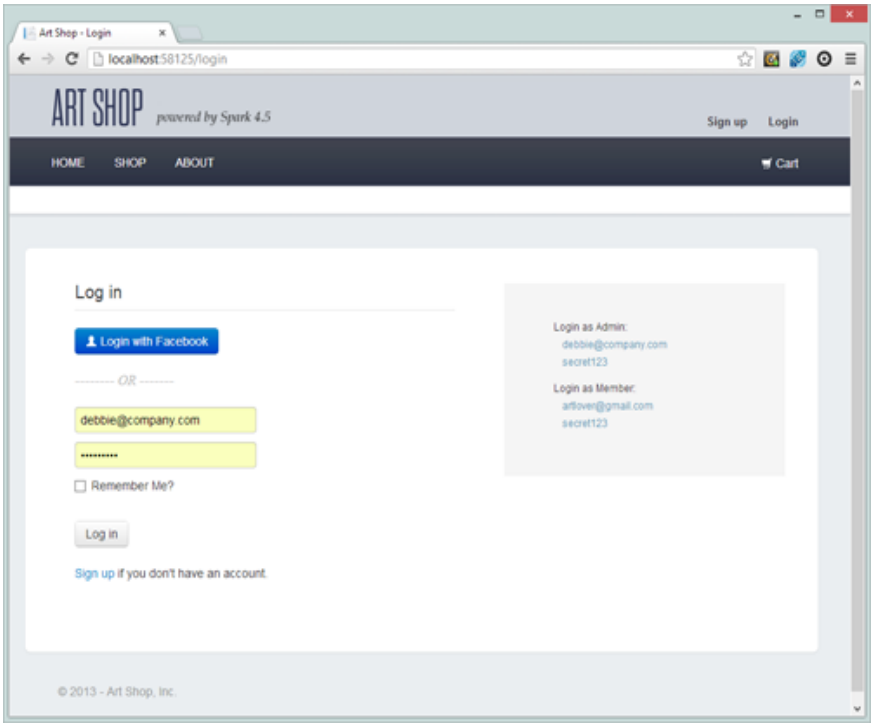
Again, in order to support Facebook authentication you will need to create an application on Facebook which requires that you have a Facebook developer account. It is easy to do and it allows you to use OAuth in your own applications.

For now, let's skip Facebook and create a local account instead. It will ask for your name, city, country, email and password. Your account will be activated immediately and you'll be logged in automatically.

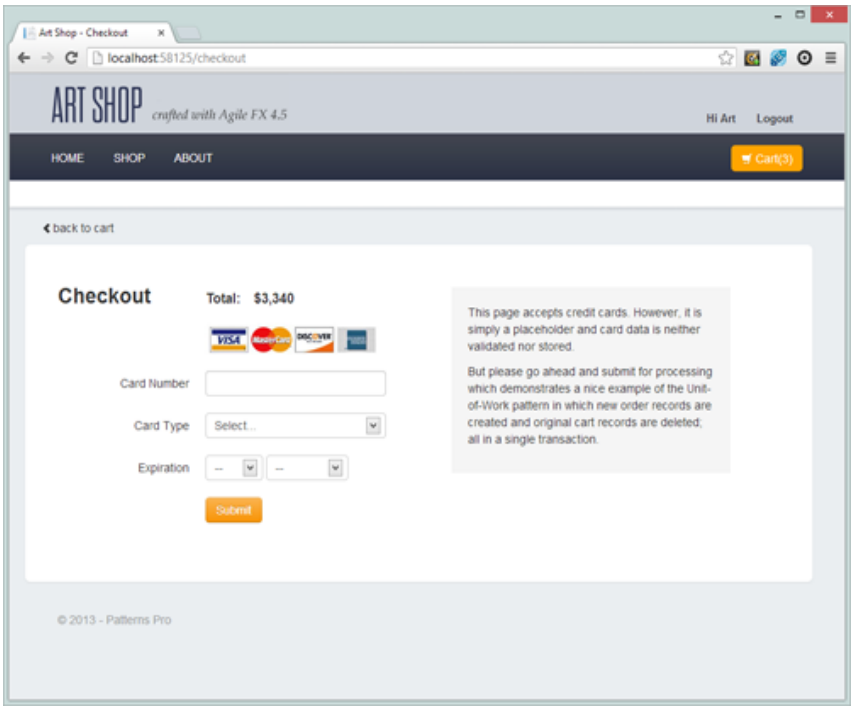
You can maintain your account and change your password by clicking on the 'Hi [name]' link at the top of the page. Of course, the Change Password option is not available for users authenticated with Facebook.



By the way, you can also login with two other accounts since Art Shop has two built-in accounts: one for Debbie which is an administrator, and one for Art Lover which is a regular user. To assist you with their logins the login page displays their full credentials (to the right on the page below).

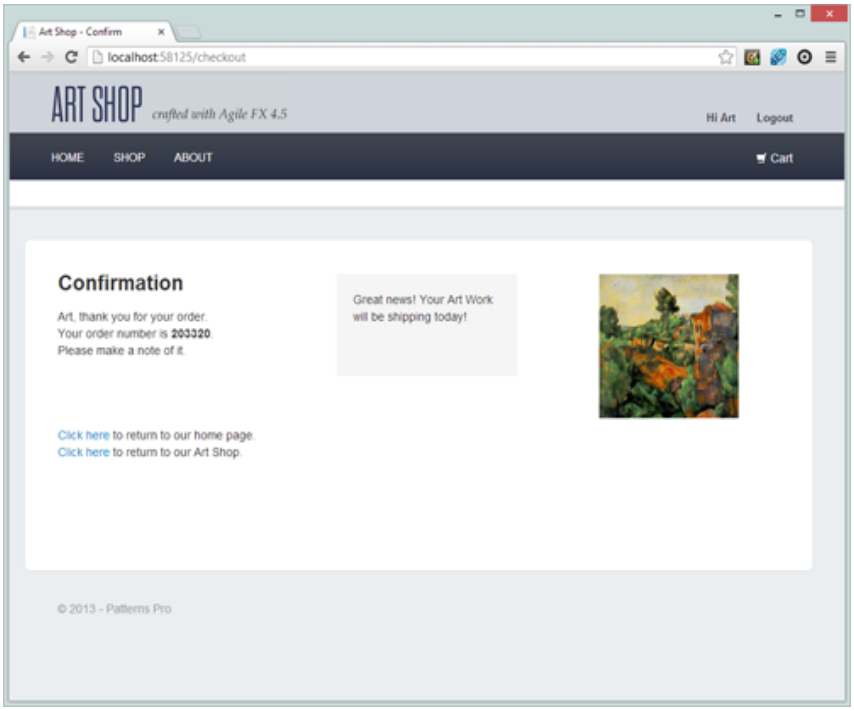


Once logged in you can go back to your cart. Then select Checkout which will bring you to the checkout page.



This is where you would make your payment and enter credit card information. This page is not validating or processing any credit card data. Simple enter some made up data and press Submit.

Your order will be created and your cart will be cleared. The next page shows you a confirmation number for your art purchase.



Notice that your shopping cart is now empty.

Changing a shopping cart into an order is an interesting process. It involves 4 tables: the Cart and CartItem and the Order and OrderDetail tables. The entire transaction has to succeed or fail as a single unit. We cannot allow partial cart changes and/or incomplete orders. This transaction is managed by the Unit-of-Work design pattern which is discussed later.

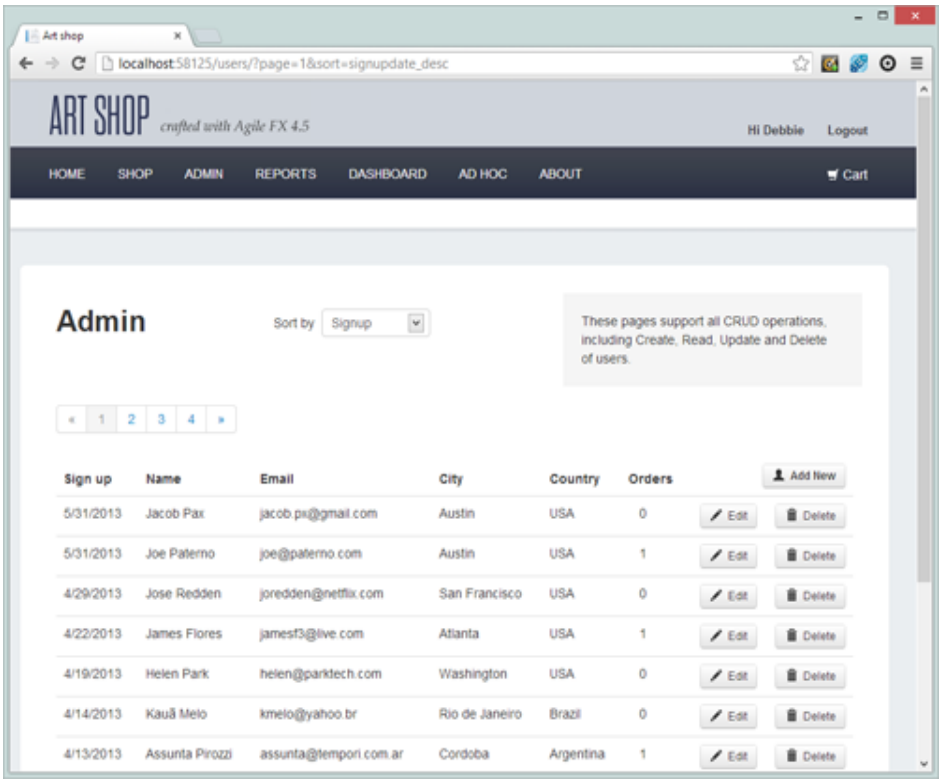
Administration

Up to this point we have shown functionality that is available to regular users. They can create an account, browse the shop, add items to a shopping cart, and place an order. That's all.

There is another role, called Admin, which has access to a wide range of administrative tasks that are available in the Art Shop application. To access this, you need to logoff and then log back in as the administrator. The admin's email is *debbie@company.com* and her password is *secret123*. In case you forget, these credentials are also listed on the login page.

Once logged in as an admin you will see four additional menu items: Admin, Reports, Dashboard, and Ad hoc.

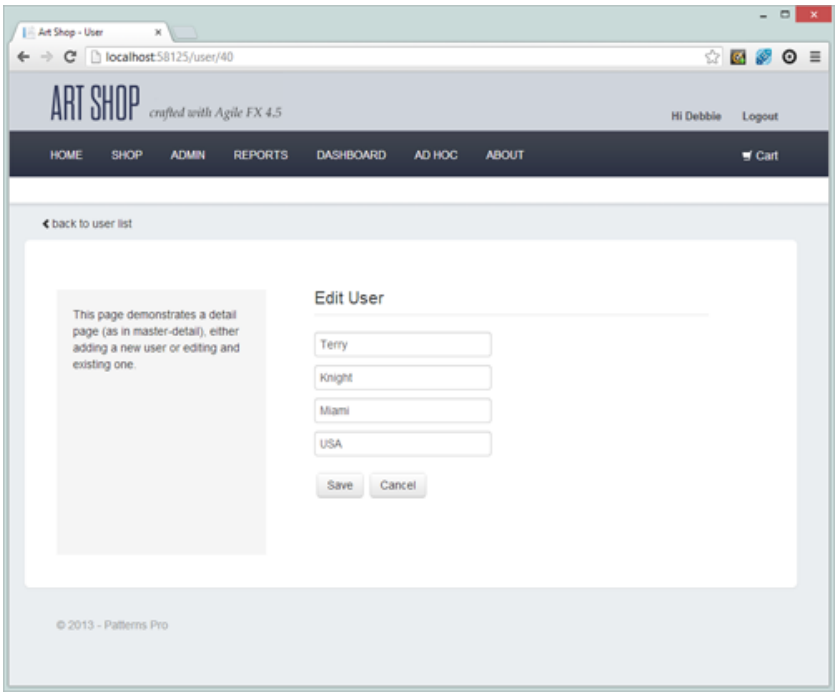
The Admin menu opens the admin page from where users are managed. It allows you to add, edit, and delete users – essentially the basic CRUD (Create, Read, Update, and Destroy) operations.



This page supports sorting and pagination. All sorting and pagination in Art Shop is performed with Ajax meaning no full page refreshes are performed. This includes this page.

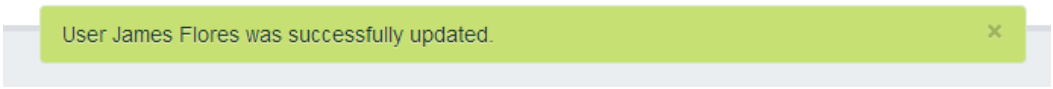
The browser history is manipulated as you can tell from the changing urls when sorting or pagination takes place. Browser history is managed because this is a master-detail scenario where the administrator can go to a user detail page and then come back to this master page. Any time you encounter master-detail it is best to also manage the browser history.

Let's see the browser history in action. Sort the page by Country and select page 3. Notice the url in the command area. It includes the page and sort order. Then click on any Edit button to edit a user.



Without making a change, select the top-left back link which will return you to the list of users. Notice that your page has the same sort and page settings before you left. This is the magic of browser history. We use the open source JavaScript library called History.js to help implement this.

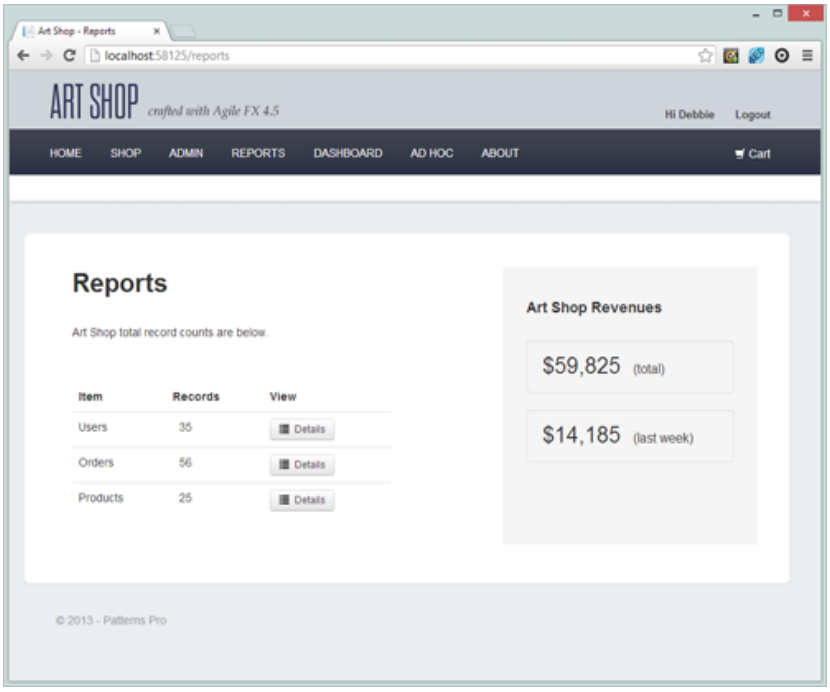
All CRUD operations, i.e. Create, Read, Update, and Destroy, are supported with the Users. Go ahead and edit an existing user, add a new one and finally try deleting an existing user. Notice the green confirmation bar that temporarily displays (for 4 seconds) on the master page when a change is successful. It will be red in case of a failure or error.



Only users who have not placed any orders yet can be deleted. Deleting users that have placed orders is not allowed because it would violate referential integrity in the database – and it does not make sense from a business perspective.

Reports

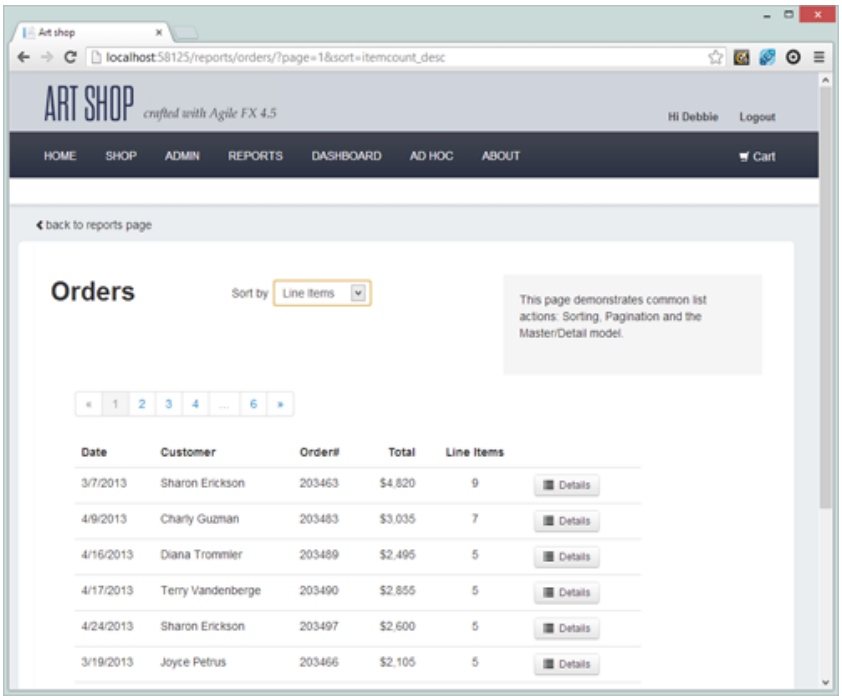
The Reports menu item opens the reports page from where you can create custom reports for users, orders, and products.



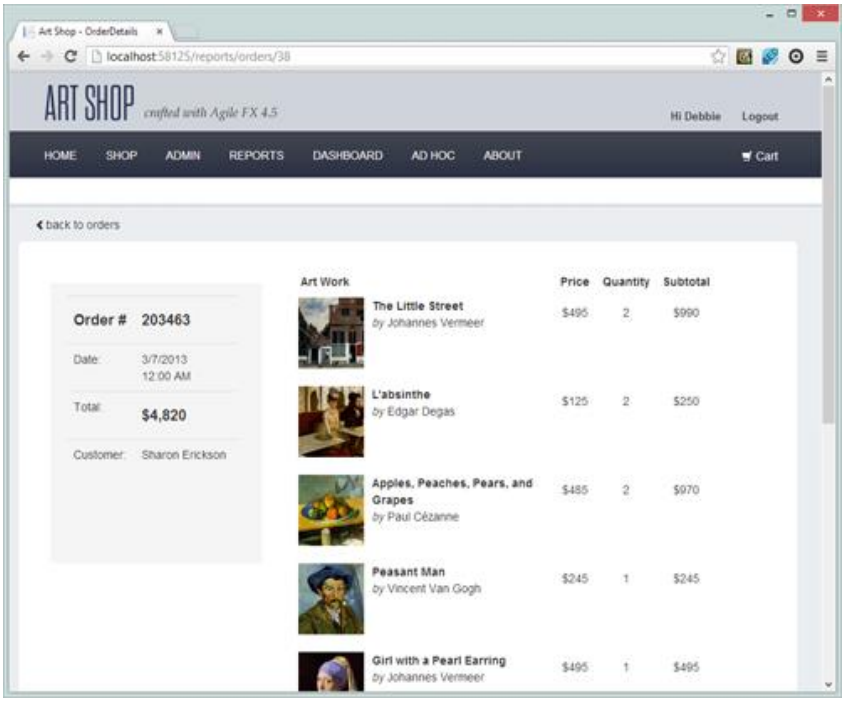
This page displays the total revenues since the launch of the site and last week's revenues. Note that the date range in Art Shop is hardcoded to the first 4 months in 2013; there is sufficient data in this period to display meaningful results, but it would be trivial to change this to a real-time, dynamic date range (when real users and orders are added to the database).

The user report page supports sorting, pagination, and filtering. However, browser history is not managed because it is not a master-detail scenario.

The orders report page supports sorting and pagination.



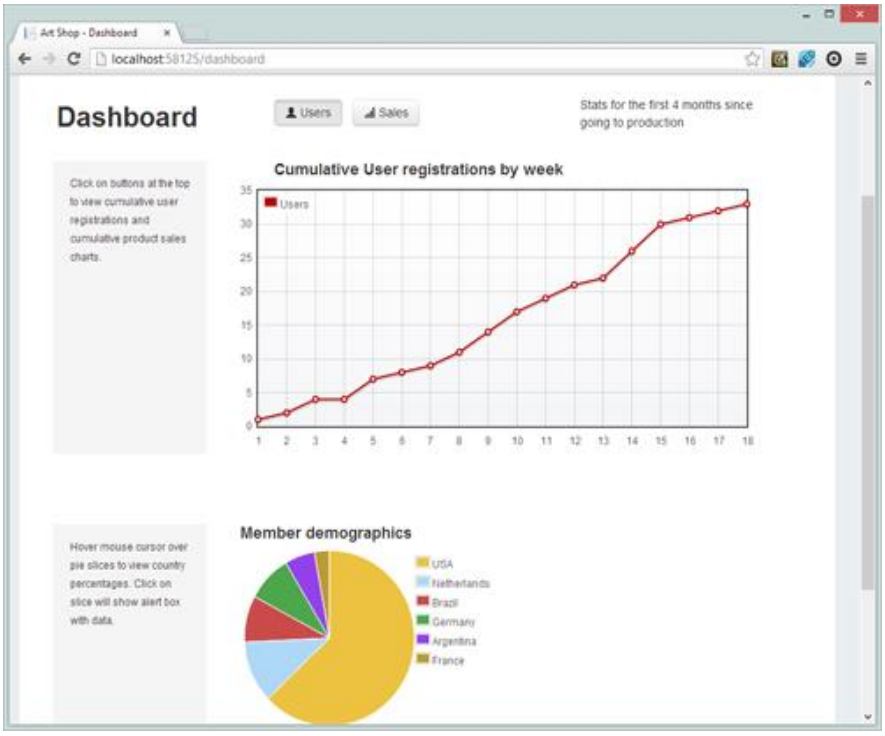
Orders have order details which is why we manage the browser history. For example sort by Line Items (# of line items) and then select page 2. Select any Details button which will bring you to the order details page. It contains order information and associated line items. Then select the 'back to orders' link at the top left of the page and you will see that the prior page's state is the same as when we left.



The product report page is relatively simple. It supports sort and pagination. Since there is no detail page there is no need to maintain browser history.

Dashboard

The Dashboard page displays performance charts of the web site.

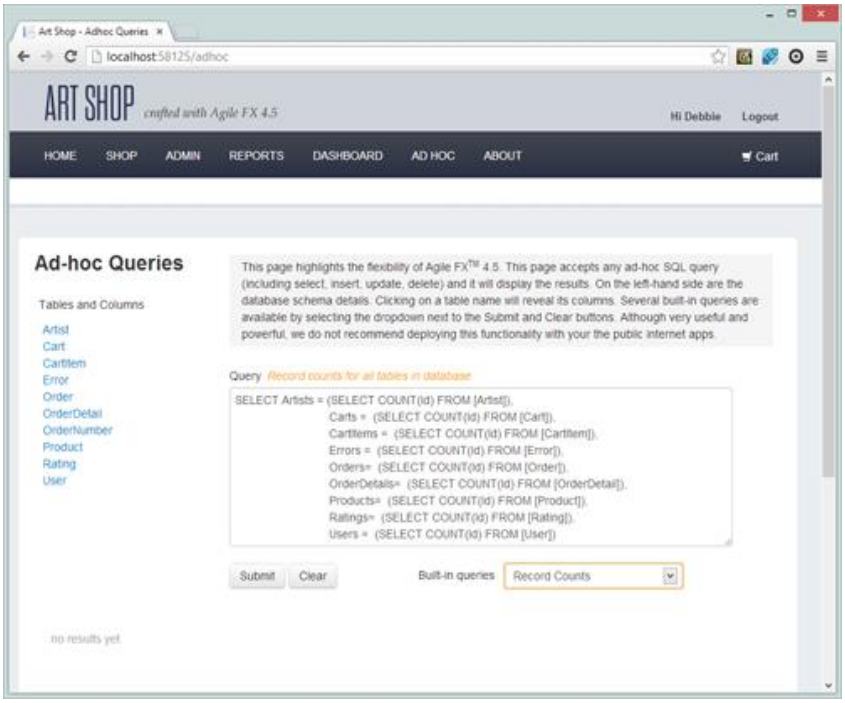


There are three charts. The first line chart displays the cumulative number of users that have signed up since the launch of the site. The numbers are grouped by week: 1, 2, 3, etc. The second line chart shows the cumulative sales in dollars by week. The Pie chart at the bottom shows user demographics, i.e. the countries the users are coming from.

The charts are implemented with an HTML5 charting package called Flot. If HTML 5 is not supported a message informs the user that a more modern browser is required.

Ad Hoc

The Ad hoc page is a powerful administrative tool.



It demonstrates the flexibility of *Spark 4.5* data access. It allows you to enter any Sql and execute it on the fly. With SELECT statements the results are displayed back onto the page. All CRUD operations are supported; SELECT, INSERT, UPDATE, and DELETE as well as any other valid query command that can be sent directly to the database.

To assist the administrator in building Sql queries a list of database tables is displayed on the left. A click on a table name will reveal all columns in that table. This list is dynamically created by querying the database for its schema; all using Spark 4.5.

The dropdown under the text area contains 4 pre-built queries. The first one shows the record counts of all tables. Next is one that shows all users with their orders, sorted by order date. Next is Artists with Products which shows all artists with the artworks they created. Finally, the 'Update Statistics' will update the aggregate statistics column in the database. Let's look at the last one in more detail.

Several aggregate values (totals, sums, averages) are maintained and updated in the database immediately when a change occurs. For example the Product table has a QuantitySold column which tracks – in real time -- the total number of items sold for that particular art piece. Doing this greatly simplifies the queries because there is no need to compute aggregate values when they are needed. Not only does this make the queries simpler (without JOINs or COUNTs), it also makes them much faster.

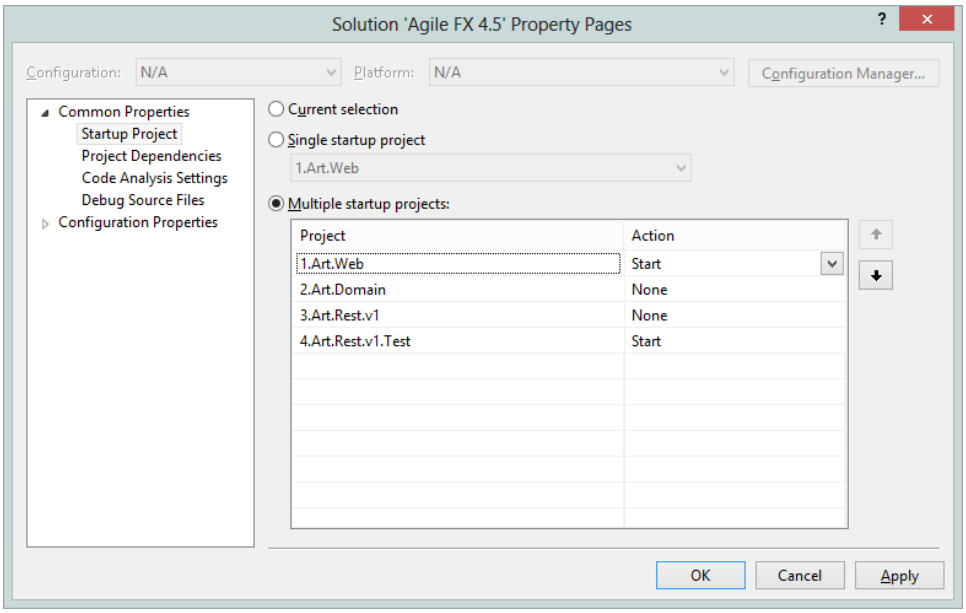
However, there may be circumstances in which these values are out of sync or become unreliable. This little Sql routine (the last query on the built-in dropdown) will re-compute and reset all these values in one fell swoop. In a production environment you could setup a scheduled task to run this, say, once every four hours, or perhaps once a day.

The ad hoc query page is a powerful tool for administrators, but for security reasons we do not recommend that you deploy with your public facing internet website (not even under a secure admin area).

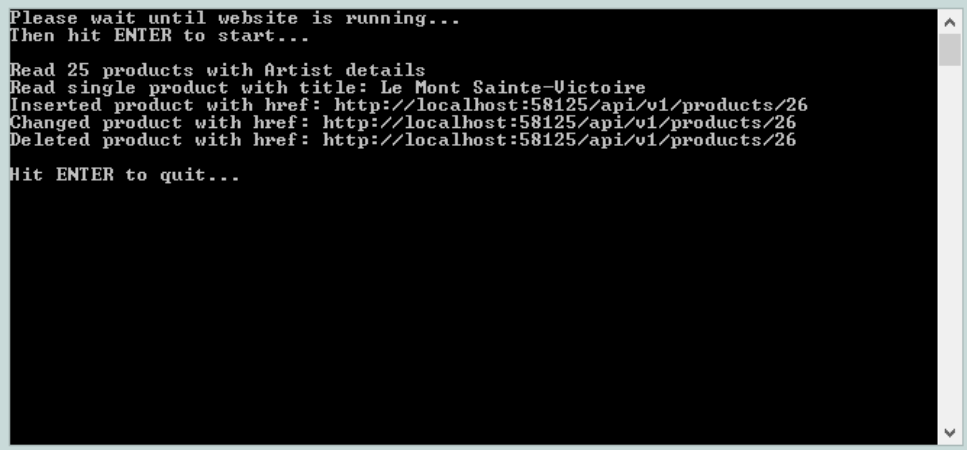
REST

The Art Shop exposes a REST based web service which is implemented with Web API. The REST API, version 1, is defined in a project named, Art.Rest.v1. Testing the REST interface is handled by a console application defined in a project named Art.Rest.v1.Test. To run the test both the Art Shop web app *and* the console app need to run concurrently.

To do this, right click on the Spark 4.5 Solution (not the project, but the solution) and select Properties. This shows the dialog below in which you select the two projects to start.



Click OK. Then run the application. The browser and the console window open. Wait until both a fully loaded. Then select the console window to give it focus and hit the Enter key. It may take a few moments for REST to start, but you should see the results of the test, like this:



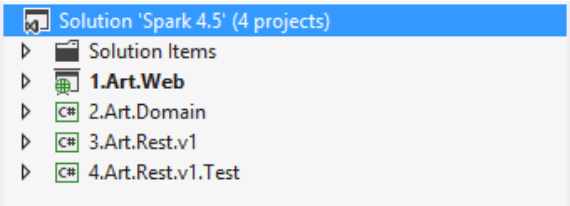
The test program displays the results; it performed two selects, an insert, update and delete. The details of REST are discussed later in this document.

This completes our tour through the Art Shop. You can now set the startup project back to the Art.Web project only.

Solution and projects

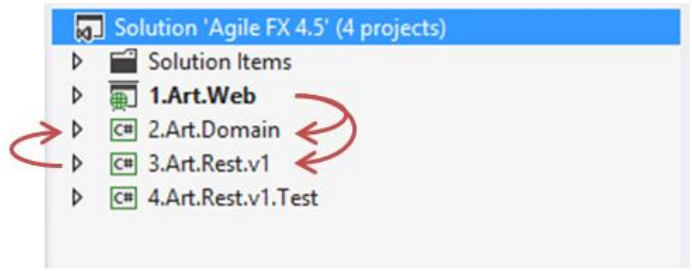
In this section we'll review the solution and projects of Spark 4.5 application in further detail.

Open the Spark 4.5 solution.



Art.Web is an MVC project. Art.Domain is a class library that supports three layers: the repository and domain objects as well as database access. Art.Rest.v1 is also a class library that implements the REST interface (version 1). Finally, Art.Rest.v1.Testdata is a console app which is designed to test the REST interface.

The dependencies among the projects are as follows:

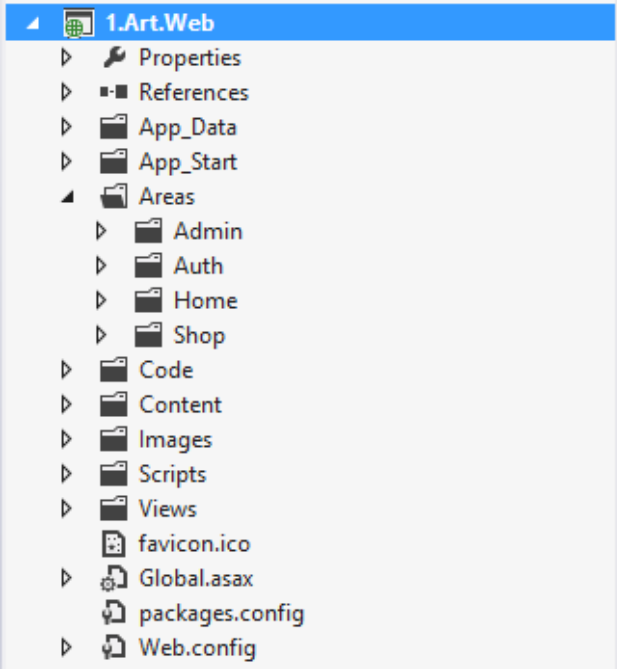


The Art.Web project references both the Art.Domain and the Art.Rest.v1 projects: Art.Domain for data access and Art.Rest.v1 for the REST interface. Art.Rest.v1 also references Art.Domain for its data access. Art.Rest.v1.Test does not reference any project but it requires that the Art.Web REST service is available when running the tests.

Let's look at each project.

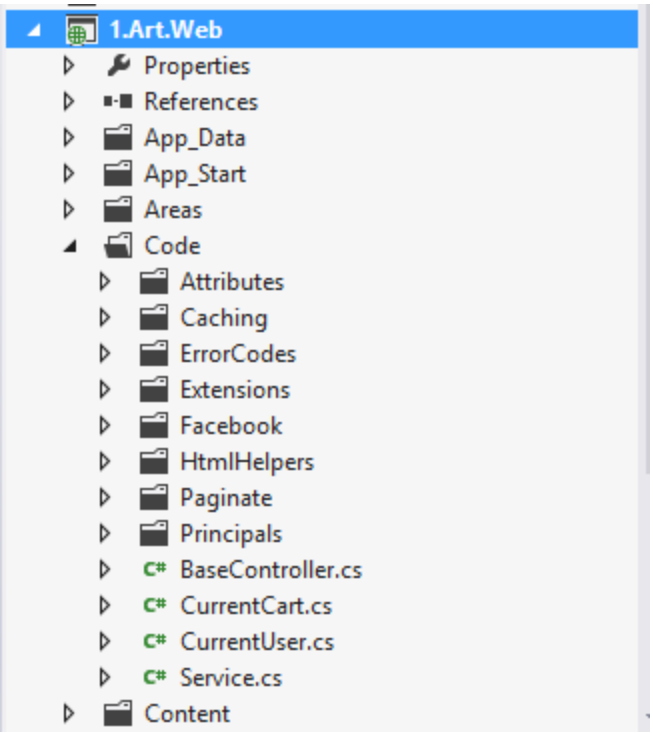
Project: Art.Web

Open the Art.Web project. This is what it looks like.



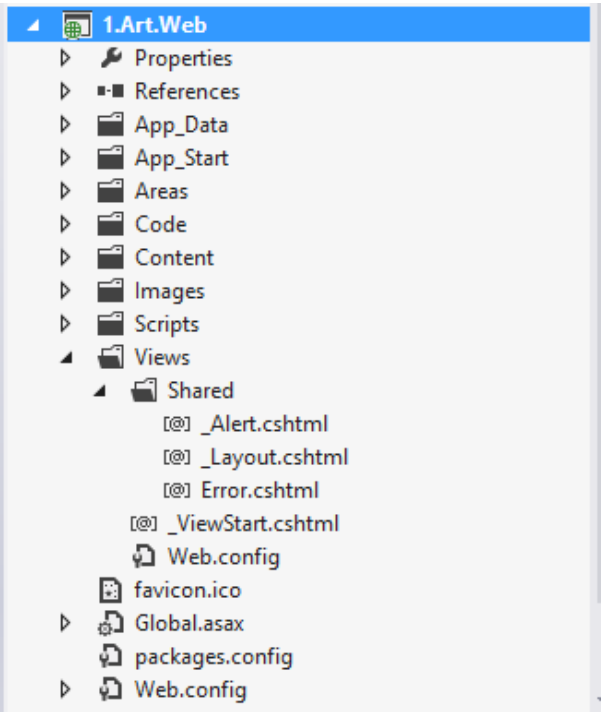
This is a regular MVC project with 4 functional areas (or Modules) under the \Areas folder: Admin, Auth, Home, and Shop. There are no controller or model files outside the \Areas, so there are no \Controllers or \Models folders in the project root folder.

Next open the \Code folder. It has several subfolders and four files in the root.



Most of the subfolders are small and have just one or two files each. Each focuses on a particular task. The four code files at the bottom are used globally, that is, they are used throughout the app.

The remaining folders and files in this project are vanilla MVC.



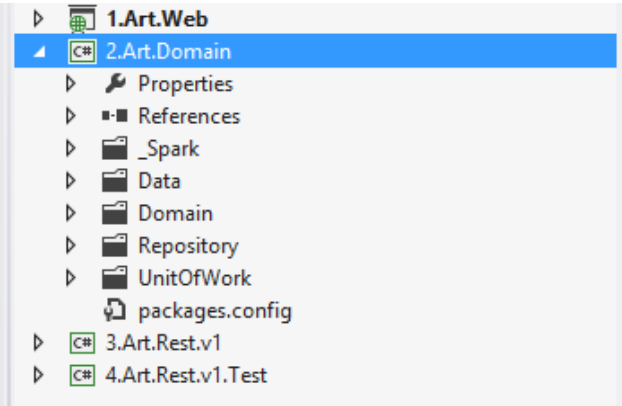
The \Content folder contains css files. The \Images contains UI images as well as images of the art being sold on the site. The \Scripts folder has all the JavaScript code. Finally the \Views folder houses the Layout file (i.e. the master page) and a couple supporting views: _Alert is a partial view which displays alert messages (success and failure) and the Error page displays when an error occurs.

The MVC project is the Presentation layer of the application. We'll get back to the Art.Web classes and components in the Code Review section.

Next we'll move on to the Art.Domain project.

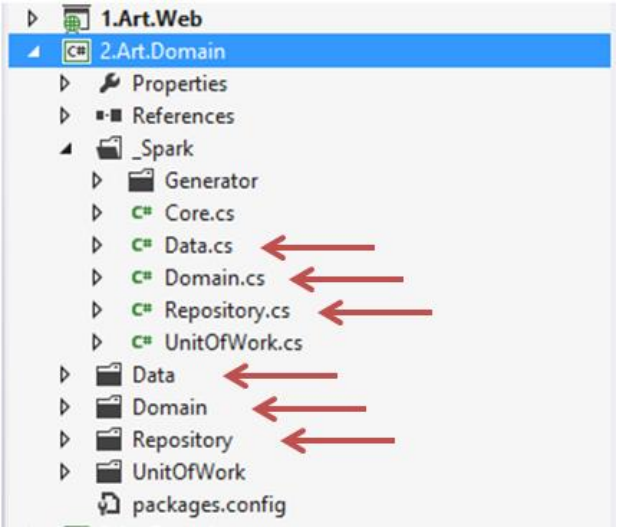
Project: Art.Domain

Open the Art.Domain project. You will see this.



Art.Domain is a class library project. It doesn't seem very large and yet it represents 3 of the 4 layers of our application stack: Repository, Domain, and Data.

Open the _Spark folder. Now you can see all layers.



The Data file *and* Data folder represent the Data Layer. The Domain file *and* Domain folder represent the Domain layer. Finally, the Repository file *and* Repository folder represent the Repository layer.

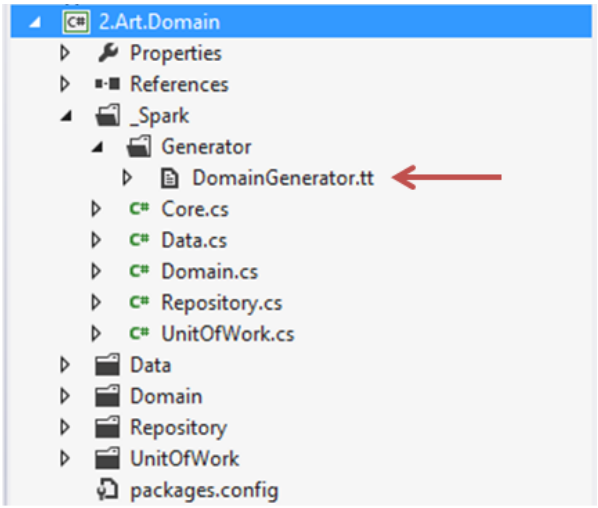
Files under the _Spark folder are not changeable, that is, once they are in place they should not be changed by any developer. This includes Core.cs which contains the core classes of Spark; they are mostly base classes to all layers: Data, Domain, and Repository. Four Art Shop specific files are also not changeable: Data.cs, Domain.cs, Repository.cs, and UnitOfWork.cs. These files all contain partial classes that have companion partials where custom code can be added.

These companion partials are located in these four folders \Data, \Domain, \Repository, and \UnitOfWork. By default these classes are empty. They are the extension points we talked about earlier. Here is an example of the Error domain object. You can see the class is empty.

```
namespace Art.Domain
{
    // Generated 05/28/2013 11:28:02

    // Add custom code inside partial class

    public partial class Error : Entity<Error>
    {
    }
}
```

Spark offers a few methods that you can override in this file. We have more to say about this in the Code Review section.

A final note about this project: if you have the *PRO* edition of the Design Pattern Framework you will see a DomainGenerator.tt T4 file which generates the *entire* Domain project: all folders and all files. In the standard edition of the Design Pattern Framework this file is empty.

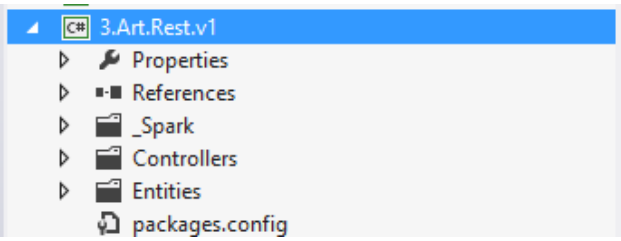


Project: Art.Rest.v1

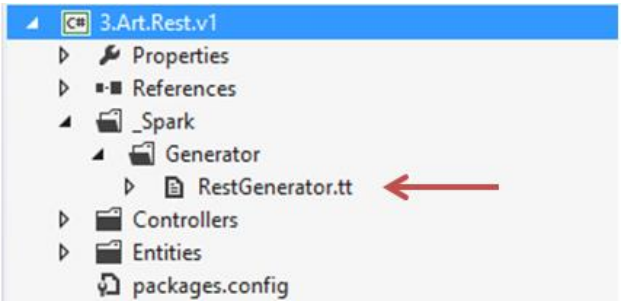
This project contains the REST interface for the Art Shop. The v1 indicates version 1. As the application evolves other versions may be required. In general it is advised to limit the number of REST versions you will release and also keep older versions in place for backward compatibility.

Web API which is new in .NET 4.5 is used to build the REST interface.

The project looks like this:



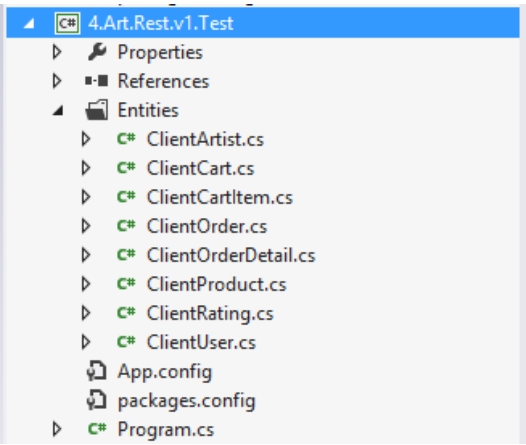
It follows a similar pattern as the Art.Domain project, that is, the RestGenerator.tt file has the ability to build out the entire project including \Controllers and \Entities folders and their files. The details of these classes are presented in the Code Review section.



The Art.Web project references this project and includes an API route entry in the WebApiConfig.cs file. This ensures that when the Art Shop is running the REST API is automatically exposed to outside clients.

Project: Art.Rest.v1.Text

This is a console project designed to test the REST API.



Under the \Entities folder are numerous client domain objects. These objects mirror the ApiDomain objects in Art.Rest.v1.

The main program, Program.cs, demonstrates how external clients consume the REST API. It runs all four CRUD operations with products, that is, it selects products, and then inserts, updates, and deletes a product. Earlier we showed the console window that displays the results of these tests.

Code Review

Now that you have an understanding of the application functionality, the Visual Studio solution and projects, it is time to delve into the code. We will explore the code layer by layer starting with the database moving all the way up to the presentation layer. Finally, we will review the implementation of the REST interface.

Database

In Spark 4.5 the data model is the very foundation to the application. Spark 4.5 uses the *Active Record* design pattern which maps each table to its own domain object. Essentially the *object model is the same as the data model*. Yes, you read that correct: the data model and the object model are the same. This greatly simplifies development because developers can focus on a single model. However, it also means that the data model must be right. We cannot overstate the importance of a good data model.

To this end, Spark introduces several conventions for the data model. These are useful in and of themselves for designing robust data models, but they also allow Spark to generate simple and clean domain objects. Spark databases are truly simple: they only have tables and indexes. There are no functions, triggers, views, cursors, or stored procedures. Data access is via dynamic Sql.

These are the conventions for the tables and indexes:

1. Table names are proper-cased and singular.
2. Column names are proper-cased like .NET object property names.
3. Intersecting table names are (usually) a combination of the tables involved (e.g. UserVote).
4. Each table has an autogenerated surrogate key (identity) of type int and is named Id.

5. Foreign key names are the 'parent table name' with 'Id' (e.g. CustomerId).
6. Each table has 4 audit fields: CreatedOn, CreatedBy, ChangedOn, and ChangedBy.
7. Favor column constraints over lookup tables. Use default values whenever possible.
8. Consider adding aggregate data fields (e.g. TotalOrders) for easy reporting.
9. Carefully add indexes based on the queries running against the tables.

We will look at each one:

1. Table names: Use short singular table names, such as User, Product, or Supplier. Proper case them just like .NET object names. If a name is a database keyword, such as User or Order then bracket their names: [User] or [Order]. Each table will have a corresponding Entity type (domain object) in the Domain Layer and each entity instance represents a single record in the table. This, by the way is the Active Record pattern, which has been popularized by tools such as Ruby on Rails and several others. Each entity represents a complete record and the entire record (row) is inserted, updated, and/or selected. We will get back to the domain entities when discussing the Domain Layer.

2. Column names: Proper case column names, just like .NET public property names. Examples are: FirstName, Email, Description, and TotalOrders. There will be a one-to-one relationship between table columns and object properties, so choosing proper names is important.

3. Intersecting table names: To model many-to-many relationships you create an intersecting table. Suppose you have students that sign up for courses and each student can take any course and any course can be taken by any student. This is an example of a many-to-many relationship. You model this with a Student table, a Course table and an intersecting table named StudentCourse. If necessary you could also name it Attend or Register, but StudentCourse implicitly names the other two tables. Other examples are: UserDocument and ProfileView.

4. Primary key: Each table has a surrogate primary key named 'Id'. This is a SQL Server identity field of type integer which is fast and easy to work with. Moreover, integers are more DBA-friendly than composite keys or uniqueidentifier. It is easier for a DBA to work on a problem and say it is user # 349901, rather than user # cd171f7c-560d-4a62-8d65-16b87419a58c. Clean and DBA-friendly models are essential if you expect effective database support. We recognize that there are scenarios, such as with distributed databases or not-always-connected clients, in which a uniqueidentifier is preferred.

5. Foreign key: Foreign key names are a combination of the parent table name and Id. For example, a Product table that references the Supplier table has a foreign key column named SupplierId. This also creates easy-to-read SQL when joining these tables; for example
SELECT * FROM Product JOIN Supplier ON Product.SupplierId = Supplier.Id.

6. Audit columns: Each table has (optionally) a set of 4 audit columns: CreatedOn, CreatedBy, ChangedOn, and ChangedBy. CreatedOn and ChangedOn are of type datetime, and CreatedBy and ChangedBy are integers, which are the Ids of the user creating and changing the record. These are the 'guilty party' fields that exist strictly for auditing purposes and should not be used in the application. So, if you have an Order table to represent orders, don't rely on the CreatedOn column to determine the

date of the order; instead include a separate OrderDate. These columns are simple to add and are of great help to identify when and by whom changes were made. These columns are automatically populated with the proper values, without developer intervention.

7. Column constraints: Suppose you are building a database with frequent flyer members. Their status can be Bronze, Silver, Gold, or Platinum. You could create a Member table with a StatusId field which references a separate table called Status that has 4 records with id values and matching status values: Bronze, Silver, Gold, and Platinum. Obviously this requires that you join the two tables for every Member select.

A better approach is to have a Status field in the Member table with a constraint for those 4 possible values and a default value. There are two advantages to this: 1) it significantly reduces the need for JOINs and 2) it is easier on the DBA. DBAs prefer to talk about member # 349901 who has a Gold status, rather than member # cd171f7c-560d-4a62-8d65-16b87419a58c with a status of 3. The 'in your face' models are far easier to work with.

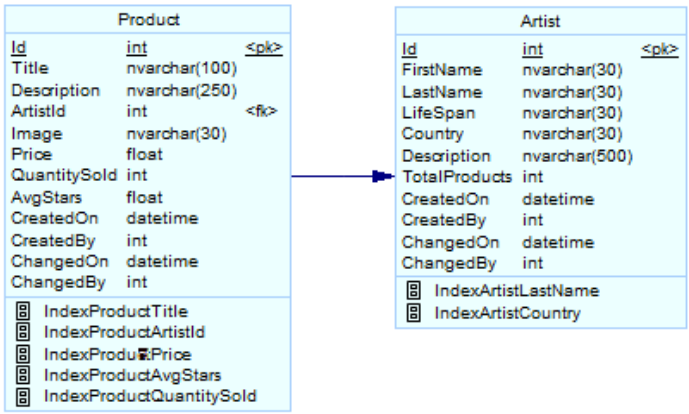
Columns with constraints usually have a default value. If there is a reasonable default value then specify it in the data model. In fact, each column should have a default value when possible in the model. Spark will automatically read the default values and assign their values to newly initialized domain objects. Whenever possible, columns of type int, float, double, datetime, should all have a default value (of course, only when it makes sense) and be declared non-nullable.

8. Aggregate fields: Aggregate fields are fields that keep live stats. Examples are: # of orders placed, # of times logged in, total amount sold, etc. This is particularly important when your app includes listings or reports that include several aggregates. Suppose you have an Employee table and you wish to list all employees and the number of Projects they are involved with as well as the total cost to the company for the time spent on these projects. Going against a transaction database (as opposed to a data warehouse) would involve a large and expensive JOIN operation.

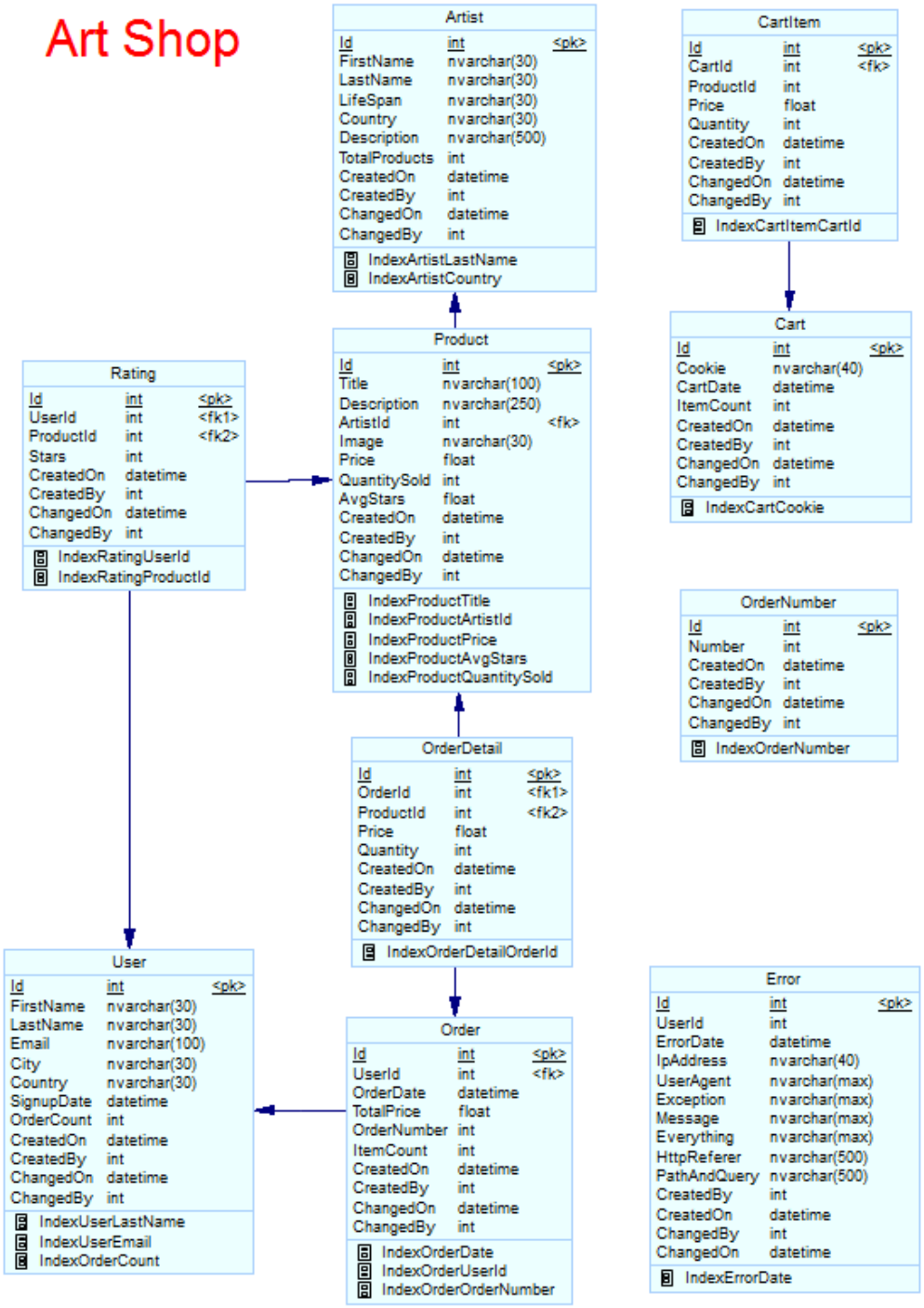
If, instead, you dynamically maintain the statistics in the primary table (Employee in this case) you won't need a single join, it becomes a straight select. If this is not immediately clear, we have several examples of this in the Art Shop. Also, Spark makes it very easy to maintain these aggregate fields with minimal impact on performance. This is done by appending and UPDATE statement to the major SQL statement. Finally if you are nervous about the potential for errors in these aggregate fields, you can simply run a SQL batch job (hourly or nightly) and re-compute all values (an example of this is available in the ad-hoc page in Art Shop).

9. Indexes: Indexes can make a huge difference in terms of performance. The larger the database, the more important they become. Indexes are frequently an afterthought, but carefully tuning a database based on the SQL queries is a very important exercise. Your users will love you for it.

In the image below you see a small subset of the Art Shop data model that demonstrates some of the data model conventions. The Artist is the artist (e.g. Van Gogh or Cezanne) of the Products (the art reproductions) being sold. Each table has an Id primary key and 4 audit columns. The foreign key in Product is named ArtistId (parent table name + Id). The QuantitySold and AvgStars are aggregate fields that are immediately updated when users purchase a product (quantity is added to QuantitySold) or offer feedback (they rate the work between 1 and 5 stars and the average of all votes is immediately updated in AvgStars).



Let's explore the actual Art Shop database. It is named Art.mdf (ArtVb.mdf in VB) and resides in the Art.Web project (you can also use Sql Server if you prefer). Below is a complete ERD diagram of the model. The DDL is available as Art.sql under the Solution Items folder in the Visual Studio solution.



There are 10 tables and they all follow the Spark database conventions. Each table lists their name at the top. Columns are next with primary and foreign key indicators <pk> and <fk>. At the bottom you see the indexes. From their names you can derive the columns they index. The arrows indicate their relationships.

The User table contains the users that have registered with the application. The Product table represents the art reproductions being sold on the site. The Artist table is the artist that made the paintings which include Van Gogh, Monet, and others. The Cart and CartItem tables represent the shopping cart. CartItem holds the line items in the cart. When a User makes a purchase, the Order and OrderDetail tables are populated. The Order table is the order which has one or more associated lines items. Order line items are stored int the OrderDetail table.

The remaining 3 tables have more supporting roles. The OrderNumber table has just a single record and maintains the next available order number. With each order this number gets bumped up. The Rating table collects user feedback in which they rate their purchase with a value between 1 and 5 (stars). Finally, the Error table collects errors that occur in the application.

Let's look at the tables and their columns. All tables have short singular names. Each table is properly indexed with indexes that support the access queries (note: properly indexing the model is critical to good performance – we have found that this is often overlooked). The column names are proper-cased and are short and descriptive. Each table has an identity field named Id and is of type int (this is a surrogate primary key).

Each table has 4 audit columns: CreatedOn, CreatedBy, ChangedOn, and ChangedBy. Foreign keys are named according to their parent table name with an Id suffix. Two foreign key examples are ArtistId in the Product table (the artist that made the product, i.e. the painting) and UserId in the Order table (the user who placed the order).

As you can see, the data store as a whole is simple. There are no views, no stored procedures, no triggers, no functions, no granting of privileges, etc. -- just tables and indexes. Despite its simplicity it is a robust, rock-solid data model that has no problem supporting the needs for the Art Shop (even when the Art Shop becomes a huge success).

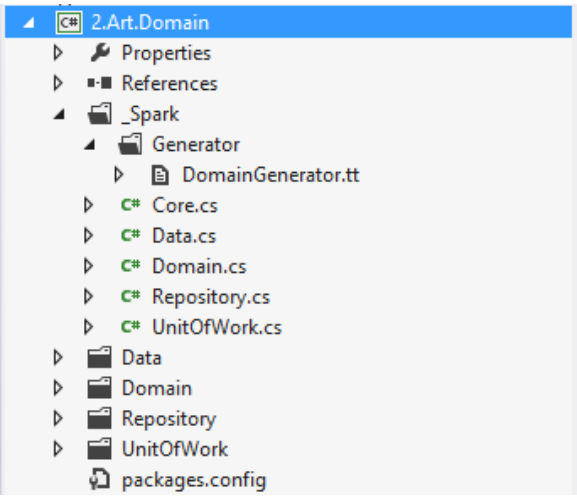
Notice also that there are no lookup tables that maintain type or status information. Spark favors column constraints over lookup tables. We mentioned it earlier but suppose there was a column named Status on the Order table with possible values: Ordered, Approved, Packed, Shipped, Canceled, and Completed. We would not create a Status lookup table and then store a StatusId in Order. Instead Spark recommends that you create a Status column with 6 column constraint values and a default value. This reduces the need for JOINs and the values are displayable directly onto the UI.

This data model combined with the Active Record pattern makes it possible to access the data almost entirely without JOINs. In fact, the entire store does not use a single JOIN (this excludes the Admin area). The result is that the application is 1) much simpler and 2) much faster; exactly what we as .NET architects and .NET developers are aiming for.

Next we will explore the Data, Domain, and Repository layers which all reside in the Art.Domain project. But before doing this we need to show a little more about the Art.Domain project and the code generator which is the topic of the next section.

Domain Generator

In the Art.Domain project you will notice a \Generator folder.



The DomainGenerator.tt file in this folder is a T4 code generator file. And here comes the amazing part: this single file has generated the *entire* Art.Domain project.

We started this project by creating a blank project and then placing the DomainGenerator.tt file in the _Spark\Generator folder. Next three values were entered: 1) a database connection, 2) a namespace name, and 3) an app key name (which is *Art* in this case). Then we ran the code generator and the entire project with all the folders and files were created! It took no more than a couple minutes.

In the standard version of the Design Pattern Framework 4.5 this file is empty, but the T4 code generator code is available in a separate product called ***PRO Design Pattern Framework 4.5*** which can be purchased on our website at www.dofactory.com. However, it is not a requirement that you have the PRO version. Using the Art Shop example you have all the relevant information and files to be able to hand-code the Domain and Repository objects yourself. The details of this are discussed later in this document.

How does the Spark code generator know what domain objects and repositories to create? First, it reads the schema information from the database which includes table names, column names, data types and defaults. Then, as a second step, it creates all data, domain, and repository classes by mapping the schema information to properties in the newly created data, domain and repository classes.

There are 2 partial classes per generated object: one that cannot be changed and another one to which you can add custom code. The partial classes under the _Spark folder are not changeable. Their counterparts (i.e. the other half of the partial classes) are located in 4 separate folders \Data, \Domain, \Repository, and \UnitOfWork. These are the extension points of Spark where customizations can be made. Based on our experience with Spark, there is usually very little code in these extension points:

you create it once and it just its work forever. You will understand the power of these extension points better once we get into their details.

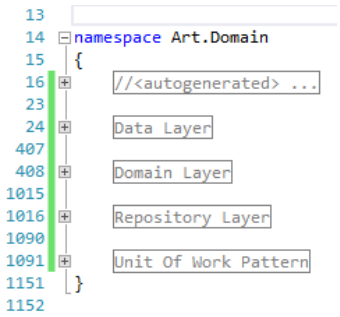
When the data model changes you will want the same changes to be reflected in the code, that is, the domain objects. Re-running the T4 DomainGenerator will quickly and easily handle this. It will refresh and update all classes under the _Spark folder, but the extension point files are not touched, that is, your custom code will not be overwritten. Only when a new table is introduced, will there be new files in the \Domain and \Repository folders.

Now that you know how the 3 layers, Repository, Domain, and Data were created we will discuss each of the layers.

Let's start with a quick look at the Domain projects under the _Spark folder and notice a file named Core.cs. It contains the foundational code of Spark, that is, base classes for all three layers and a small number of supporting classes. Here is a quick overview of the main classes and the layers they represent.

Class	Layer/Pattern
Db	Data Layer
Entity	Domain Layer
Repository	Repository Layer
UnitOfWork	UnitOfWork Pattern

To help you locate the classes the file is partitioned in 4 regions.



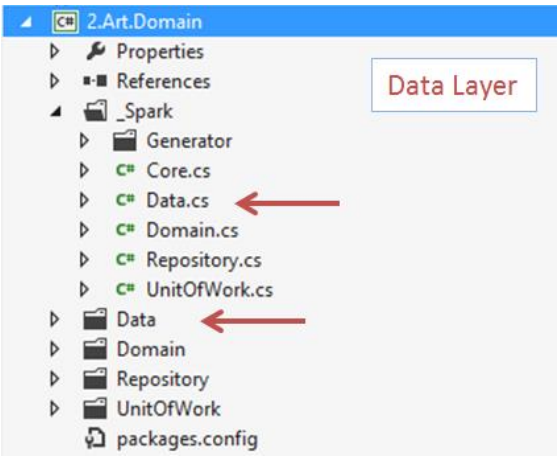
Notice the file size: about 1150 lines. It is amazing to think that this is the entire foundation of 3 out of 4 layers. When using Spark you will discover that any custom code you add to these layers is minimal. Spark is truly a simple, lean, and lightweight platform (especially when compared to the Entity Framework).

When using Spark you will never change or touch any of the core classes. They are just doing their work. But understanding them and their roles will help in your understanding of how to build lean, pattern-based systems.

Data Layer

The Data Layer, also called Data Access Layer (DAL), provides access to a database. It must do so efficiently and securely. Its goal is to handle data access generically so that if application requirements change it can handle these without any changes to this layer.

The data layer in Art.Domain is found in the Data.cs class and in the \Data folder.



Just a heads up: the following section gets into the details of the Data layer. As a developer using Spark there is no need to understand all details. However, if you are interested in learning how to build a fast, pattern-based applications then this discussion will be of interest to you. Depending on your interest you may skip to the next section which discusses the Domain layer.

The Data Layer in detail: Db class

The Db class (in Core.cs) is very important and handles all data access in Spark. It is extremely fast by using a forward-only stream of records using DbDataReader. While iterating over the records, the *yield return* keyword ensures that the newly created domain objects continue to be streamed to the requesting source without the need to build an intermediate list of objects. In summary, it is super optimized.

Scrolling down a bit in the Db code you find comments indicating that the CLR does not allow the *yield return* statement to appear inside a try or catch block. To circumvent this limitation a couple wrapper methods have been added. The try catch blocks are very important in these methods because, in case of an error, they allow us to capture both the SQL and parameters. This is really helpful when debugging a problem.

Class Db uses ADO.NET and it is the *only place* in the entire app in which ADO.NET objects are referenced. Many programmers don't like the repetitive nature of programming data access; with Spark you don't have to think about it at all. You won't deal with ADO.NET at all.

Next, let's step through the different methods in Db. This class is very important because it represents pretty much the entire Data Layer.

The Db constructor gets the connection string from web.config. The Read method accepts a parameterized sql statement and an optional set of parameters. It also accepts a *make* delegate that transforms a table row into a domain object. The Query method is similar to Read except it takes in any parameterized SELECT statement with parameters and returns a collection of .NET *dynamic* objects. The difference between Read and Query is that Read will return a collection of type-safe domain objects, whereas Query returns a collection of *dynamic* objects. The Read method is always preferred, but the Query method gives you more querying flexibility.

The Scalar method returns any scalar value. It is used for aggregate SQL queries, such as SUM, COUNT, MIN, MAX, etc.

The methods Insert, Update, and Delete do exactly as they say. All accept a parameterized SQL statement and an optional set of parameters.

The next group of methods is designed to support transactions. They are BeginTransaction, EndTransaction, TransactedInsert, TransactedUpdate, and TransactedDelete. These methods work together and are invoked by the UnitOfWork class which coordinates the transaction taking place.

Three DataSet based methods are available for easy access. They are GetDataSet, GetDataTable, and GetDataRow. They make data access and data manipulation extremely easy and are included for convenience. However, they are much slower than all other data selection methods in this file. The DataSet methods are not used in Art Shop.

CreateConnection, CreateCommand, and CreateAdapter are private helper methods that create ADO.NET objects to support all data access methods in this file.

AddParameters is an extension method that adds parameters to a command object. For SELECT statements we use ordinal parameters (i.e. they are numbered) and for all other statements named parameters are used.

Another extension method called ToExpando iterates over the fields of a DataReader and returns an ExpandoObject which is an object whose members can be dynamically added or removed at runtime.

Finally, DbException is a custom exception object designed to capture the Db context, including SQL and parameters, in case of an exception. All exceptions are logged into a table called Error in the database.

Db maintains a connectionstring which is initialized when the constructor is called. To make this process easier we derive from Db with a specific connection string name. In the Art Shop the derived class is called ArtDb which is used by all domain objects. The code for ArtDb resides in the data layer in a file named Data.cs. Here is the code; it could not be any simpler.


```
public partial class ArtDb : Db
{
    public ArtDb() : base("Art") {}
}
```

This is partial class. Its counterpart is called Data.partial.cs in the \Data folder which allows you to add any custom data access functionality. In reality, this is rarely, if ever, used.

For now, all you need to remember is that ArtDb is used by all domain objects so they know which database to access. In case you're curious, the ArtDb database is assigned in the base class to all domain objects which is Entity in the Core.cs file. Here is the relevant code:

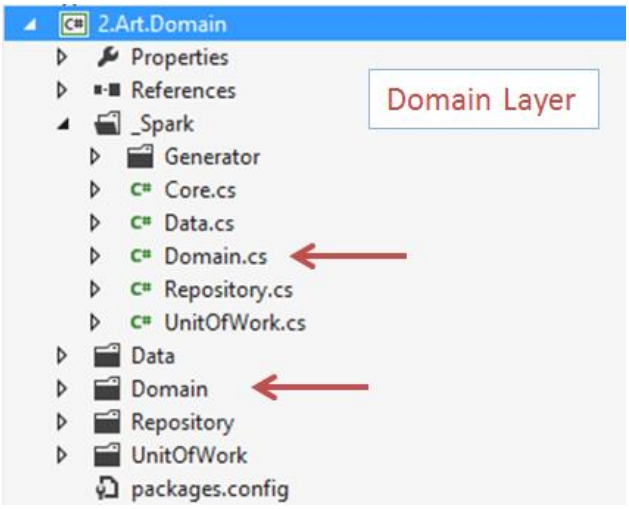
```
static void InitDb(Db _db, string table, string key)
{
    db = _db ?? new ArtDb();
    tableName = table ?? typeof(T).Name;
    keyName = key ?? "Id";
}
```

The domain objects are discussed next.

Domain Layer

The Domain layer, also known as the Business Logic Layer (BLL), is a collection of domain objects. Domain object are also sometimes called entities or business objects. Domain objects are containers of information and their rules that are relevant to the problem domain space. Examples are Artist, Product, and Cart; these are all logical entities that are relevant to the Art Shop problem domain.

The domain layer is located in the Domain.cs class and the \Domain folder.



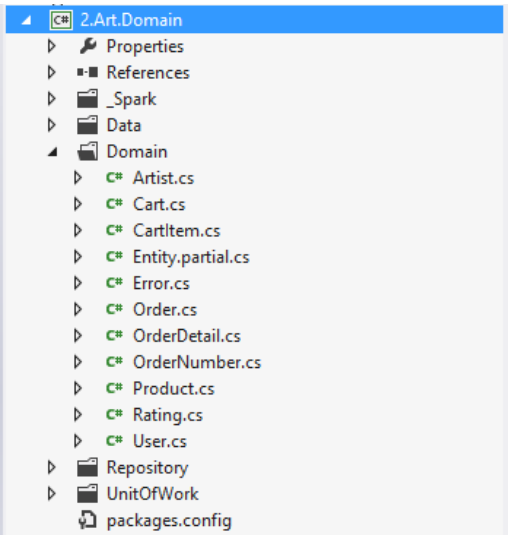
Domain.cs contains all domain classes for Art Shop. Open the file and you'll see that the domain classes have properties that are named the same as the columns of their associated tables. Here is an example:

```
public partial class Artist : Entity<Artist>
{
    public Artist() { }
    public Artist(bool defaults) : base(defaults) { }

    public int? Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string LifeSpan { get; set; }
    public string Country { get; set; }
    public string Description { get; set; }
    public int TotalProducts { get; set; }
}
```

Each domain class has two constructors, a default constructor and one that accepts a boolean to indicate whether the object should be initialized with default values or not. The latter is only used with new domain objects that are going to be saved to the database. It ensures that default values, as defined in the data model, are properly set. You can confirm in the Art Shop application code (Controller classes) that the overloaded constructor is used only when new objects are inserted into the database. In fact, this is the Prototype pattern in which you create entities (domain objects with values) according to a prototype (the table record with column defaults).

Domain objects that already exist in the database get all their values from the database. There is no need to initialize these with default values, so in these cases the default constructor is called.

All domain classes are partial and their counterparts reside in the \Domain folder; one class for each table. Here is the list for Art Shop:



The base class to all domain objects is the Entity class in Core.cs. Entity is a partial class itself and its counterpart is Entity.partial.cs located in the \Domain folder. This is where you add custom functionality that applies to *all* domain objects. This is rarely used and in the Art Shop this file is empty.

To add custom code for a particular domain type, such as Artist, Cart, or Product, you use the files in the \Domain folder: Artist.cs, Cart.cs, and Product.cs. These are the extension points we mentioned earlier.

To see how this works, open Artist.cs and you will see a utility property named FullName that concatenates FirstName and LastName.

```
public partial class Artist : Entity<Artist>
{
    public string FullName { get { return FirstName + " " + LastName; } }
}
```

Another example is the Order.cs file in which the aggregate value named TotalOrders in the User table is updated with each Order insert or delete. It occurs in two overridden methods: OnInserting and OnDeleting.

```
public partial class Order : Entity<Order>
{
    protected override void OnInserting(ref string sql)
    {
        sql += "; UPDATE [User] Set OrderCount = OrderCount + 1 WHERE Id = " + this.UserId + ";";
    }

    protected override void OnDeleting(ref string sql)
    {
        sql += "; UPDATE [User] Set OrderCount = OrderCount - 1 WHERE Id = " + this.UserId + ";";
    }
}
```

These virtual methods are called just before Inserting and Deleting their records. In it, we append a Sql UPDATE to the sql that is about to be sent to the database. This is extremely powerful. It allows you to maintain aggregate values throughout the database (or do something else), simply by appending a simple sql statement which removes the need for another roundtrip to the database.

To see how the extension points work, we need to look at the Entity class in file Core.cs.

Half way the class you will find two pairs of 6 methods that allow you to customize the code, either just before or just after insert, update, and delete commands.

```
// partial methods that allow custom coding (for all entities) before and after

partial void OnInsertingAll(ref string sql);
partial void OnInsertedAll();
partial void OnUpdatingAll(ref string sql);
partial void OnUpdatedAll();
partial void OnDeletingAll(ref string sql);
partial void OnDeletedAll();

// virtual methods that allow custom coding (by entity type) before and after al

protected virtual void OnInserting(ref string sql) { }
protected virtual void OnInserted() { }
protected virtual void OnUpdating(ref string sql) { }
protected virtual void OnUpdated() { }
protected virtual void OnDeleting(ref string sql) { }
protected virtual void OnDeleted() { }
```

The first block contains partial methods that you can implement in Entity.partial.cs. Implementing these will affect *all* domain objects. In Art Shop these are not used (and since they are partial methods the compiler simply removes these methods). The second block contains virtual methods that you can

override in a particular domain type (entity). This will affect only those particular domain objects, such as Artist, Product, or Order.

Let's scroll down in Entity.cs to the Insert, Update, and Delete methods and see how these custom methods are invoked:

```
// inserts current entity instance

public virtual void Insert()
{
    string sql = sqlInsert;

    OnInsertingAll(ref sql);
    OnInserting(ref sql);
    this[keyName] = db.Insert(sql, Take());
    OnInserted();
    OnInsertedAll();
}
```

The actual Insert call (db.Insert) is wrapped by two pairs of methods: OnInsertingAll and OnInsertedAll which will execute for all domain objects, if implemented. The other pair is OnInserting and OnInserted which can be overwritten at a particular domain class such as Artist, Product, or Order.

Open Order.cs and see the OnInserting and OnDeleting overrides:

```
// Add custom code inside partial class

public partial class Order : Entity<Order>
{
    protected override void OnInserting(ref string sql)
    {
        sql += "; UPDATE [User] Set OrderCount = OrderCount + 1 WHERE Id = " + this.UserId + ";";
    }

    protected override void OnDeleting(ref string sql)
    {
        sql += "; UPDATE [User] Set OrderCount = OrderCount - 1 WHERE Id = " + this.UserId + ";";
    }
}
```

Appending the proper SQL to the original incoming sql string will send two sql statements in a single trip to the database. The result is that the aggregate values are always up to date and this, in turn, avoids the need for complex joins and/or calculations. In the example above, the User table does not need to join with Order to find out how many orders each User has placed.

The Entity.cs base class also offers a Validate method that you can implement for each domain object type. The entity named Product has an example.


```
public partial class Product : Entity<Product>
{
    protected override void Validate()
    {
        // simple validation rules

        if (string.IsNullOrEmpty(Title)) Errors.Add("0001", "Title is required");
        if (string.IsNullOrEmpty(Image)) Errors.Add("0002", "Image is required");

        if (Title.Length > 100) Errors.Add("0003", "Title cannot be longer than 100 ch
        if (!string.IsNullOrEmpty(Description) && Description.Length > 250) Errors.Add

        if (ArtistId == null || ArtistId < 0) Errors.Add("0005", "Invalid ArtistId");

        if (Price < 0) Errors.Add("0006", "Price cannot be negative");
        if (AvgStars < 0) Errors.Add("0007", "Avg Stars cannot be negative");
        if (QuantitySold < 0) Errors.Add("0008", "QuantitySold cannot be negative");
    }
}
```

Each domain object has an IsValid property that internally calls Validate. If there are errors they are logged into an Errors dictionary in the domain object. When the error count is greater than zero the validation fails. Note that validation is not called automatically; the client needs to explicitly check the IsValid property. As an example, the AdminController in the Art.Web project has a PostUser action method where you see an example IsValid call (note that the code does not act upon the results).

The Domain layer holds the domain objects. Spark supports the Active Record pattern which means that for each table there is a matching domain type. Using Active Record, the object model is the data model and vice versa. It totally solves the *impedance mismatch* problem which exists between the relational databases and object oriented class structures. If you are not familiar with this problem, search Google for *impedance mismatch*.

The Active Record pattern also removes the need for complex object trees where parent objects have multiple child objects that have child objects themselves, etc. Frequently, these child objects are lazy loaded but, unbeknownst to the developers, this can lead to inefficient SQL data access in which a separate query is issued for each child. With a long list of parent objects this will result in an explosion of queries to the database. We once encountered a project where it took 50,000 queries to render a single page. The developers knew the page was not performing well, but did not quite understand why.

With the Active Record pattern each object knows how to Select, Update, Insert, and Delete itself. It is the opposite of the Data Access Object (DAO) Pattern in which each entity has a dedicated helper object (the data access object) that handles the saving and retrieval of the data to and from the database.

By the way: the following discussion gets into the details of the Domain layer and Entity which is the base class to all domain objects. As a developer using Spark there is no need to understand the full details of the foundational code. However, if you are interested in learning to build a fast, pattern-based

applications then this discussion will be of interest to you. Depending on your interest you may skip to the next section which discusses the Repository layer.

The Domain Layer in detail: Entity class

Entity is the base class to all domain objects. It resides in Core.cs. Knowing the internals of Entity will help you in understanding how the Domain layer works.

Entity is a partial generic class where T is Entity<T>. This is called a self-referencing type constraint. This constraint allows each type that derives from Entity to reflect over its properties just once and then keep this information in cache for the lifetime of the application. Reflection is slow and caching this information makes a huge difference.

Each domain type goes out to the database and gets the column information for the table that matches the domain object name. This is done lazily (i.e. Lazy Load Pattern), that is, only when the type is first used will the schema information be retrieved. Lazy loading is very beneficial because it prevents a delay at application startup time. It also prevents querying the database for object types that are never used during the application session.

The column default values are stored and whenever the overloaded constructor is called with a *true* value, an instance will be created with all its properties initialized with the proper default values (this is called the Prototype Pattern).

Upon retrieval of the table schema each entity type creates and stores a number of sql statements for each of the CRUD actions; these include select, insert, update, delete, and paged (i.e. paginated select). Note that this is done for the object *type*, and not for each object *instance*.

Domain objects know how to insert, update, and delete themselves; the relevant methods are Insert, Update, and Delete. Domain objects also expose methods to select data from the database and create entity instances. The relevant methods are Single, All, Paged and several methods that return scalar values: Scalar, Count, Max, Min and Sum.

All these methods can be called directly on the domain objects, but it is far better to access these via a Repository (the Repository Pattern). This keeps the interface tidy and simple. In Art Shop none of the domain object methods are called directly. Each domain object has a matching Repository; for example the Artist has a matching Artists repository which as you will see resides on a Context object. This will be explained in the next section which presents the Repository layer.

Another set of methods allows transacted operations. A transaction is a group of operations (updates, inserts, etc.) that either succeeds or fails as a group. The methods are TransactedInsert, TransactedUpdate, and TransactedDelete and they all participate in a database Transaction. These methods are not meant to be called directly, as they are used by the UnitOfWork class which is responsible for coordinating transacted actions.

As mentioned earlier, the Entity class offers a Validate method that can be overridden in the domain type classes. Examples are available in Product.cs and User.cs. It is designed for object property validation before the data gets sent to the database.

You may be wondering "when do I use this kind of validation?" This is a reasonable question because in MVC apps there are already two levels of validation: one at the UI level (with JavaScript) and another by checking the ModelState in the Action methods when data is posted back to the controller.

However, the database is also accessible via the REST interface which totally bypasses the two aforementioned validations, so there certainly is a need to validate the data before it gets sent to the database. This is what these validations can be used for.

In general, most MVC applications have 2, 3, or even 4 places where validation takes place: the UI, the controller, the service layer, and possibly just before new data gets sent to the database. This sounds like a code smell in need of some serious refactoring, but to give users the best experience while keeping the applications safe and secure this is (unfortunately) what is required.

Entity is also the place where the audit fields are managed and updated. The datetime columns, CreatedOn and ChangedOn, are easily updated. The UserId fields, CreatedBy and ChangedBy, require that Entity has access to the current user. A virtual method named TryGetUserId extracts the value from the current Thread using a Custom Principal. This works hand-in-hand with the Custom Principals used in the Presentation layer (which is explained later in this document). If necessary, you can overwrite TryGetUserId and provide your own implementation.

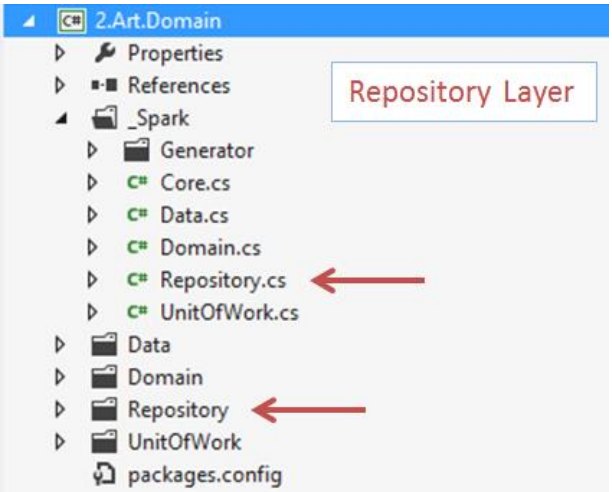
In Spark the use of the audit fields is optional -- only when Entity detects that all four fields are present for a particular domain type will they be maintained. However, audit fields are very helpful in tracking and solving database activities, so we always recommend using these.

Repository Layer

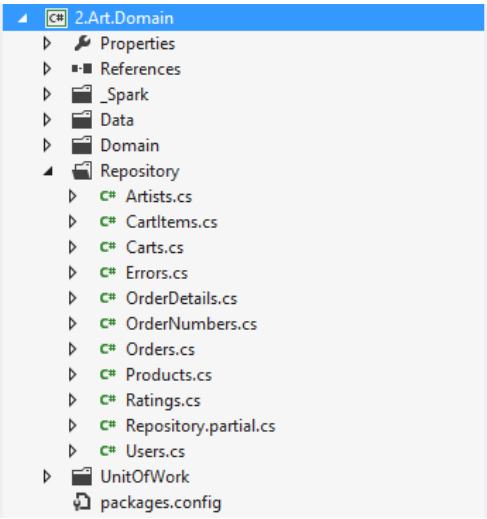
The goal of the repository layer is to provide a simple and consistent interface between application code and the logic that retrieves the data from a database and creates domain objects.

The Spark repository does exactly that. Each domain type has its own repository which returns and accepts objects of that type. A convention in Spark is that a repository name is the plural of the domain name. For example the Product type has a repository named Products and the Order type has an Orders repository. Repositories are typed and work only with their assigned domain type.

The repository layer is located in the Repository.cs file and the \Repository folder.



They follow the same pattern as the Data and Domain layers. The non-changeable Repository.cs class is located under the _Spark folder. The \Repository folder provides the extension points in the form of partial classes.



The Repository.partial.cs allows you to add global repository functionality to all repositories. This is not used in the Art Shop.

The Artists.cs, and CartItems.cs, etc. (notice these are plural) files are the places where you can add custom methods to the specific repositories. Two classes in Art Shop's \Repository folder have custom code: Users and OrderNumbers. Open Users and see that it adds a ByEmail method which was added because Users are frequently retrieved by email and this makes doing so simpler. OrderNumbers has a Next method which returns the next available order number which are generated when a user checks out their shopping cart.

All repository files derive from a class named Repository. Open the Core.cs file and locate Repository which is a generic class that handles Entities, that is, domain objects. The Single and All methods are used to retrieve a single and a collection of domain objects respectively. The Paged method returns a

paged list of objects that are used in pagination. The Insert, Update, and Delete methods do what they say. Several scalar methods are available such as Scalar, Count, Sum, etc. which return aggregate scalar values from the database. Two general purpose methods, named Query and Execute, allow ad-hoc queries against the database; Query returns a collection of *dynamic* .NET objects.

This list supports any possible scenario, so the chances of you having to add a global custom method to all repositories (in Repository.partial.cs) are slim to nothing.

Context

Repositories are accessed all the time in the client code. To make them easily accessible a context object is created that wraps all repositories. In Art Shop it is called ArtContext. For each domain object there is a matching repository on ArtContext. The context file is located in Repository.cs under the _Spark folder. Here is the code:

```
// Art Context

public static class ArtContext
{
    static Db db = new ArtDb();

    // entity specific repositories

    public static Artists Artists { get { return new Artists(); } }
    public static Carts Carts { get { return new Carts(); } }
    public static CartItems CartItems { get { return new CartItems(); } }
    public static Errors Errors { get { return new Errors(); } }
    public static Orders Orders { get { return new Orders(); } }
    public static OrderDetails OrderDetails { get { return new OrderDetails(); } }
    public static OrderNumbers OrderNumbers { get { return new OrderNumbers(); } }
    public static Products Products { get { return new Products(); } }
    public static Ratings Ratings { get { return new Ratings(); } }
    public static Users Users { get { return new Users(); } }

    // general purpose operations

    public static void Execute(string sql, params object[] parms) { db.Execute( sql, parms ); }
    public static IEnumerable<dynamic> Query(string sql, params object[] parms) { return db.Query( sql, parms ); }
    public static object Scalar(string sql, params object[] parms) { return db.Scalar( sql, parms ); }

    public static DataSet GetDataSet(string sql, params object[] parms) { return db.GetDataSet( sql, parms ); }
    public static DataTable GetDataTable(string sql, params object[] parms) { return db.GetDataTable( sql, parms ); }
    public static DataRow GetDataRow(string sql, params object[] parms) { return db.GetDataRow( sql, parms ); }
}
```

The top half contains static properties with the same name as the Repositories. They return a newly created repository of the same name. Below that are 3 general purpose static methods that execute a sql command, retrieve a collection of dynamic objects from the database, and a scalar value. These are designed to handle unusual or complex situations that are not easily handled by the typed repositories. Notice that they go directly to the db object (i.e. Data layer) bypassing the domain layer (db.Execute, db.Query, etc.).

The bottom 3 methods support the older ADO.NET DataSets. These are slower than any of the other methods which are all highly optimized. They are only included for convenience and offer flexibility and easy access to the data. The DataSet methods are not used in the Art Shop reference application.

You may notice that the same methods exist on Repository and Context. Examples include Query, Execute, and Scalar methods (although their arguments are a bit different). However, there is a difference. The Repository returns specific domain types whereas the Context is general and will return *dynamic* or *object* types as well as data in DataSets (using the last three methods). Whenever possible use the (typed) Repository methods because domain objects are much easier to deal with than dynamic objects.

Next, let's look at an example in the Art Shop where ArtContext is used (there are many other examples in Art.Web project).

```
[HttpGet]
[ActionName("User")]
public ActionResult GetUser(int? id = null)
{
    var model = new AdminNewUserModel();

    if (id != null)
    {
        var user = ArtContext.Users.Single(id);
        model = Mapper.Map<User, AdminNewUserModel>(user);
    }

    model.HttpReferer = Request.UrlReferrer.PathAndQuery;
    return View("User", model);
}
```

This is the GetUser action method in the AdminController. In the middle you see a User repository call: ArtContext.Users.Single(id) which retrieves a single user by id. There is no need to explicitly new up a repository because this is handled inside the static Users repository property.

Tip: ArtContext is frequently used and to reduce typing you could create an alias with a using statement at the top of the controller class, like so:

```
using art = Art.Domain.ArtContext;
```

This will allow you to shorten your queries to this:

```
var user = art.Users.Single(id);
```

In multiple database scenarios you can use multiple short names: *art* for ArtContext and *inv* for InvContext (inv stands for Inventory).

The context classes themselves are not given short names to avoid naming conflicts.

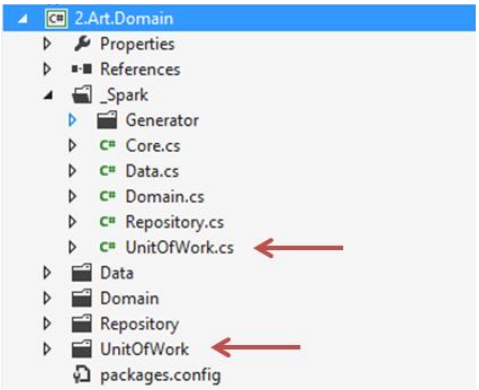
Unit of Work

In Sql Server, by default all database actions are performed in *autocommit* mode, meaning each action is committed automatically to the database (committed means physically written to disk). You don't have to do anything special for this.

However, if you have a series of database operations (insert, update, etc.) and they need to either succeed or fail as a group you need to manage these as a single transaction (also called a single unit-of-work). Suppose you have 5 actions and the last one fails, then the entire transaction is 'rolled back' and the database will restore the already changed records back to their original state. This prevents the database from ending in an unstable or incorrect state.

The Art Shop has a nice example: when a user makes a purchase the app needs to ensure that the cart's contents are moved over to the order tables and that the cart records are cleared. This involves 4 tables (Cart and CartItem, and Order and OrderDetail) and all operations need to succeed. Code examples of how this is accomplished are available in Service.cs in the Art.Web project.

Note that Unit of Work is not a layer, but rather a class that supports the Unit of Work design pattern. The relevant files are highlighted below.



The Unit of Work pattern is available as two separate classes: UnitOfWork and DistributedUnitOfWork. You'll find them at the bottom of the Core.cs file. The UnitOfWork class manages transactions in a single database, whereas the DistributedUnitOfWork class manages distributed transactions which involve one or more different data stores.

The DistributedUnitOfWork requires that MSDTC (Microsoft Distributed Transaction Manager) is running. MSDTC is a COM component that is turned off by default; you need to start it. For this reason the line scope = new TransactionScope() in this class is commented out. Once MSDTC is running you can uncomment this line back in. If you have a single database use UnitOfWork as it is considerably faster than DistributedUnitOfWork.

Open the file Repository.cs and you will see that it has an ArtUnitOfWork class that is initialized with the Art database. The \Repository folder contains a file that allows you to add custom code to the UnitOfWork or DistributedUnitOfWork classes. They are rarely used.

To see the Unit Of Work pattern in action we refer to the Service.cs class in Art.Web which represents a small service layer in the Art Shop application. You will notice that DistributedUnitOfWork is used despite the fact there is just one database. The reason is that there are multiple connections to the database: our own and the SimpleMembership. Since we have no access to the latter we are forced to use the distributed unit of work.

Building Domain projects with Spark 4.5

As mentioned, the *PRO Design Pattern Framework 4.5* product, which is available for purchase on dofactory.com, includes a code generator that automatically creates the entire Domain project including the Data, Domain, and Repository layers. However, you can also create your own Domain projects without the code generator, but it requires that you create these classes by hand. Let's look at what is involved.

Suppose your database is called Sales. It adheres to all Spark data model conventions listed earlier in this document. Start a new class library project named Sales.Domain. Then create the exact same folder structure as in Art.Domain. Copy the file named Core.cs to _Spark folder. Inside Core.cs change the namespace to Sales.Domain and on line 446 or thereabout change 'new ArtDb()' to 'new SalesDb()'. Save and close.

Copy the Data.cs, Domain.cs, Repository.cs, and UnitOfWork.cs files and inside these files change all occurrences of the word 'Art' with 'Sales'. The Domain.cs and Repository.cs files require more information about the database tables which is discussed shortly.

Copy the content of Art Shop to the \Data, \Domain, \Repository, and \UnitOfWork folders. Remove the application specific files in \Domain and \Repository, such as Artist and Artists, User and Users, etc. You will replace these with your own based on the tables in your Sales database. In general replace all instances of 'Art' to 'Sales'.

Next make a list of all database tables and their columns. Consider how the columns map to properties. Follow the pattern as outlined in Art.Domain. The \Data folder should have SalesDb.cs and Db.partial.cs. The \Domain folder should have Entity.partial.cs and a number of partial domain objects that match the tables in the Sales database. Likewise \Repositories will have Repository.partial.cs and a number of partial repository files for each table with the name pluralized. And \UnitsOfWork will have just UnitOfWork.partial.cs file with two empty classes. Again remember to set the namespace in all these files to Sales.Domain.

Finally, return to Domain.cs and include all Sales classes and their properties. Simply follow the pattern as outlined in the Art Shop. In the Repository.cs class adjust the static SalesContext class to include all repositories relevant to the Sales applications. This completes the creation of the 3 layers.

So, it is a bit of work, but it is a rather mechanical process and entirely feasible. Simply follow the patterns laid out by the Art Shop. The advantage of using the **PRO** code generator is that 1) it is extremely fast, 2) there will be no errors, and 3) subsequent data model changes are quickly reflected in the domain classes by simply re-running the T4 code generator. The challenge with manually maintaining the code is when the data model changes and you need to re-adjust the domain and repository objects.

Multiple databases

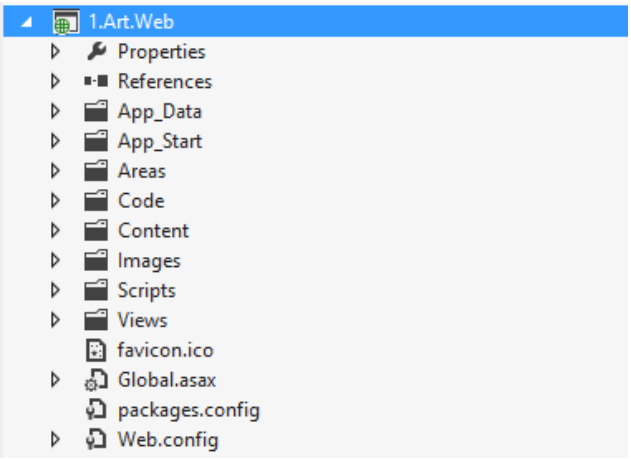
Accessing multiple databases is very easy in Spark because each database will have its own Domain project. Suppose that the Art Shop needed access to a second database which is an Inventory database,

named Inv. We would simply create a new project named Inv.Domain and point the PRO Design Pattern Framework 4.5 code generator to this database (or hand code the domain and other objects) and voila, you're ready to access both the Art and Inv databases.

In your client code you access these databases with the ArtContext and InvContext static objects (or art and inv aliases as explained earlier). The `UnitsOfWorkDistributed` class will easily handle distributed transactions that span the two databases. The code that accesses both databases will most likely reside in the Service.cs file in the MVC project. MVC and the Presentation layer are discussed next.

Presentation Layer

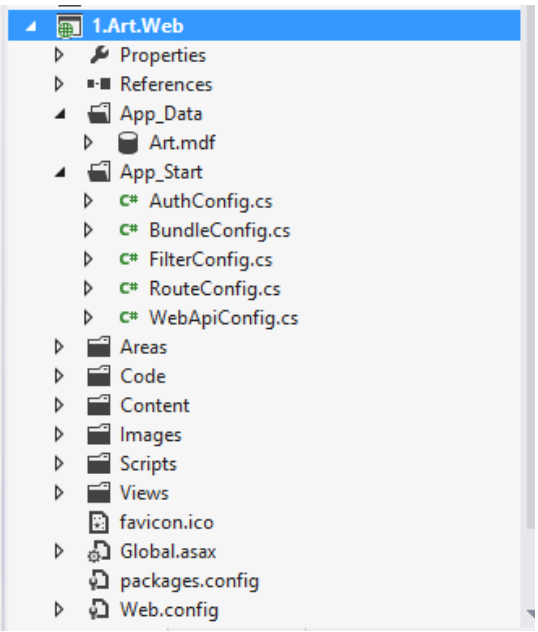
The Presentation layer contains the components that implement and display the user interface and manage user interaction. The UI in the Art Shop is built with MVC and resides in the Art.Web project shown below:



This should look familiar as it follows the standard MVC project structure. Three new folders are Areas, Code and Images. The \Areas folder contains MVC components (Models, Views, and Controllers) for the different areas (modules). The \Code folder contains a series of custom components organized in separate sub folders. The \Images folder has images and art work used in the application. We'll review the folders from top to bottom.

App_Data and App_Start

Open the \App_Data and \App_Start folders.



The Art Shop LocalDb database is called Art.mdf (or ArtVb.mdf) and resides in \App_Data. We discussed the database earlier.

There are five configuration files in \App_Start. Three of these contain custom code for the Art Shop: AuthConfig, BundleConfig, and WebApiConfig. RouteConfig is empty because routes are defined inside the Areas themselves in their respective AreaRegistration files.

AuthConfig is where the Facebook client is registered. This allows users to authenticate via Facebook. The Facebook authentication is fully implemented in Art Shop.

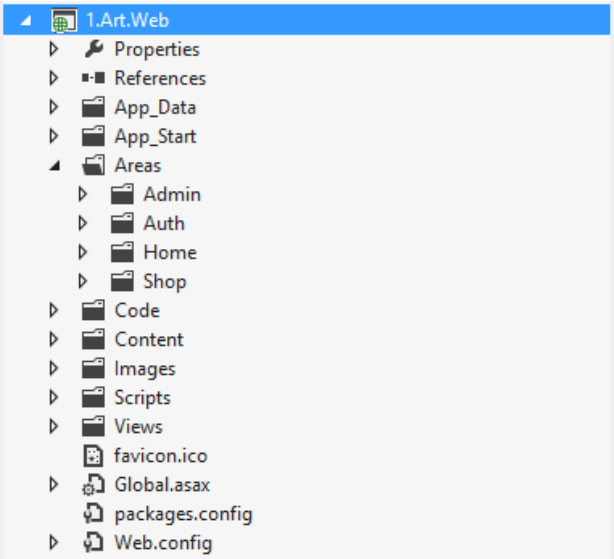
BundleConfig is where css and js bundles are configured. Bundles are a new feature in MVC 4.

WebApiConfig is where the Web API (i.e. REST) routes are registered. The routing rules in the REST interface for the Art Shop are simple and easy to follow. It follows this template: api/v1/{controller}/{id} which represents version 1 of the api. The controller name is a plural domain object name, so we may have /artists, /products, and /users. The id is the domain object identifier. Here is an example url: api/v1/artists/5 which refers to the artist with id 5.

The REST functionality is coded in project Art.Rest.v1, which is referenced by the Art.Web project. The WebApiConfig settings ensure that when the Art Shop is up and running it also exposes the REST api routes. The REST projects are discussed shortly.

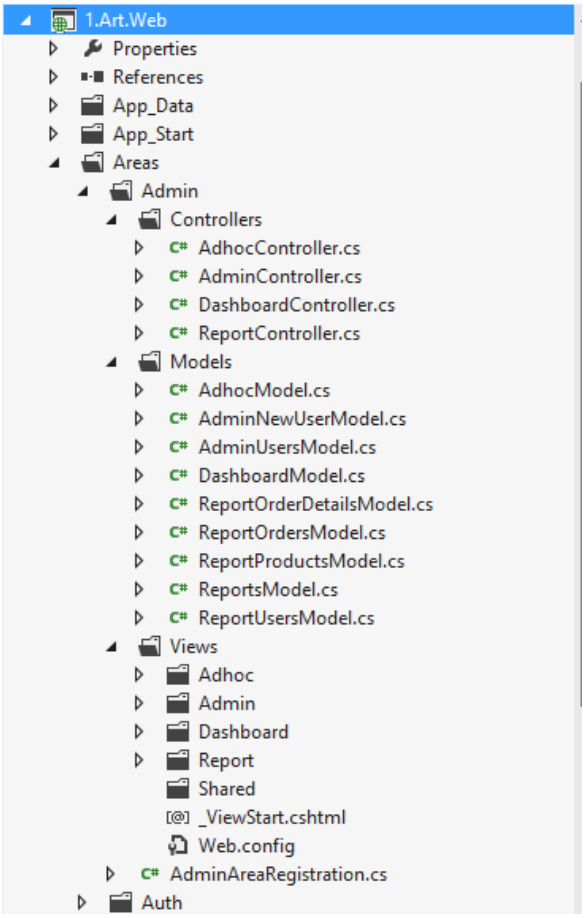
Areas

Next, we'll look at the \Areas folder.



The application contains 4 modules in the \Areas folder: Admin, Auth, Home, and Shop. Admin is the administration area, which is only accessible to the Administrator. Auth contains the authentication code, such as login, logout, and signup. Home contains the home page and the about page. Finally Shopping contains the store, the shopping cart, and the checkout pages.

As an example, see the Admin area.



It contains 4 controllers, each managing a particular sub area within the administrative area (which is fairly large). Each view has its own Model class and the view pages are in the Views folder.

Models

The Model classes are implementations of the DTO (Data Transfer Object) pattern. Their purpose is to transfer data from one point to another, from the Controller to the View. Model objects have no methods, just data. Frequently the data is already formatted, so for example many monetary fields are of type string, and have values such as $43,210.99, ready to written on the page. This reduces the need for logic on the View.

Controllers

Open AdminController; this controller is fairly representative for all other controllers in the app. Here is the code:

```
[Authorize(Roles = "Admin")]
public class AdminController : BaseController
{
    static Dictionary<string, string> UserSortItems { get; set; }

    static AdminController()
    {
        UserSortItems = new Dictionary<string, string>();
        UserSortItems.Add("signupdate_desc", "Signup");
        UserSortItems.Add("lastname_asc", "Last Name");
        UserSortItems.Add("email_asc", "Email");
        UserSortItems.Add("city_asc", "City");
        UserSortItems.Add("country_asc", "Country");
        UserSortItems.Add("ordercount_desc", "Orders");

        Mapper.CreateMap<User, AdminUserModel>();

        Mapper.CreateMap<User, AdminNewUserModel>();
        Mapper.CreateMap<AdminNewUserModel, User>();
    }
```

First we have an Authorize attribute which will only allow administrators access to the action methods and associated views. The controller derives from BaseController which provides base functionality that is useful to all controllers, such as allowing you to set page meta data (title, keywords, description) and pass success and failure message strings to the Views.

A static dictionary holds the sort items for a sorting drop down. The static constructor initializes this dictionary only once, as well as the Automapper mappings that are used in this controller. The mappings are between domain objects and the Model objects and vice versa.

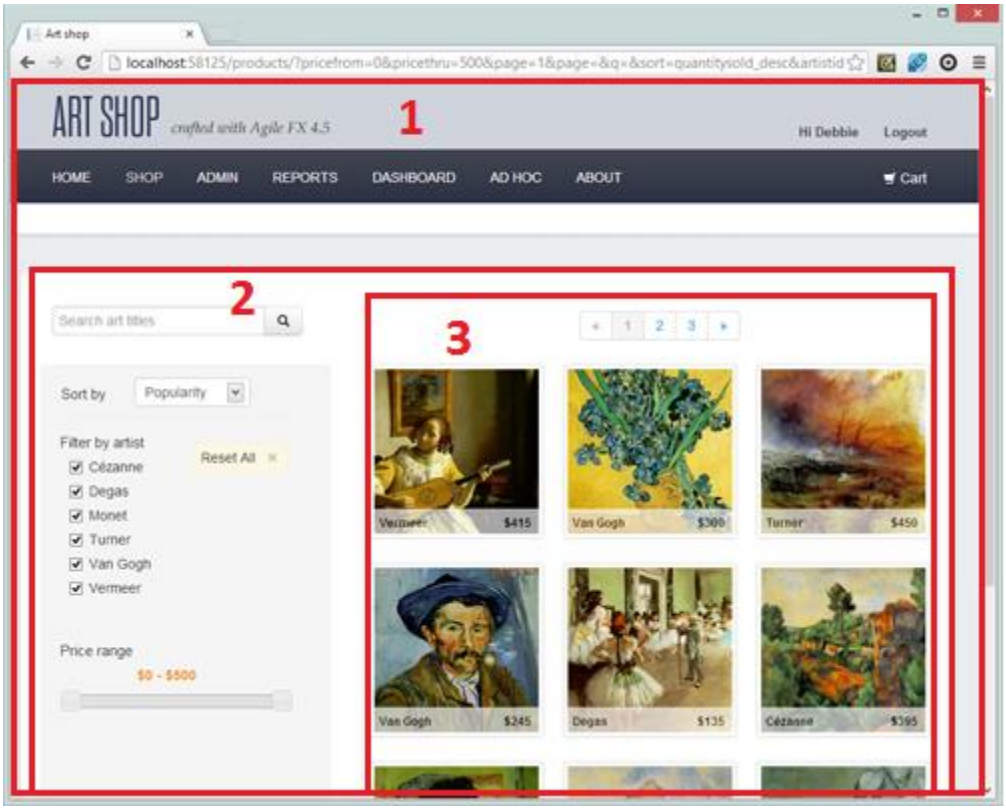
Let's examine a simple action method. This one is named GetUser.

```
// GET

[HttpGet]
[ActionName("User")]
public ActionResult GetUser(int? id = null)
{
    var model = new AdminNewUserModel();

    if (id != null)
    {
        var user = ArtContext.Users.Single(id);
        model = Mapper.Map<User, AdminNewUserModel>(user);
    }

    model.HttpReferer = Request.UrlReferrer.PathAndQuery;
    return View("User", model);
}
```

With the HttpGet attribute we indicate that the action method only accepts Http GET requests. The ActionName attribute is User which allows the routing system to send GET, POST, AND PUT requests to the same Action method name. MVC parses the Id from the url and provides it as an argument here. First a new Model instance is created. Next the User is retrieved from the database and mapped with Automapper to the model. At the end the model is passed to a View.

This was a simple action method. Let's now examine an action method that is more involved: Products in the ProductsController. So, open the ProductsController. This method renders the products View which is the main shopping page in the Art Shop. A lot is happening here because on this page users can search, sort, filter, and paginate through a list of products. JavaScript, jQuery, and Ajax are used extensively to provide the user with a great UI experience.

The Products method handles three different render scenarios: 1) the entire page, 2) part of the page (everything below the menu bar), and 3) the list of products with pager controls. These three render scenarios and their zones are highlighted below.



Number 1) is a full page render action which occurs when the page displays for the first time. Number 2) is like a full page refresh, but without the layout elements defined in the _Layout page, and Number 3) is a request for only the list of products (art works), as well as the pagers on top and on the bottom of the list. Both 2) and 3) are Ajax calls. Let's see how the Products action method handles these different scenarios.

Number 1) renders when the shop is accessed for the first time, or when the user returns from the product detail page. The Products action method has a number of parameters that match the query parameters in the url. All parameters have default values and when the page is rendered for the first time default values are used.

When a search, sort, filter, or pagination action is performed on the page, the History.js JavaScript tool will update the browser history, which is visible as a changing url in the command area.

Whenever the user leaves the page to see product details and then returns back to this page the original url is restored, meaning that all parameters in the Products action methods will receive the same values as when the user left this page.

Number 2) is when no _Layout is included. This occurs when the beige 'Reset All' button is clicked which resets all controls and the product list. This is a jQuery Ajax call. Notice that the last parameter in the Products method is called `layout`; it is a boolean that indicates whether to include the _Layout or not. The default is true, but it is set to false when the page reset takes place. The layout elements, including menu and header, did not change, therefore _Layout is not necessary. The `layout` argument value is checked at the bottom of the action method. If false, the layout will be suppressed.

Number 3) is also an Ajax call. This occurs when the user changes any of the controls on the page, such as, Search, Sort, Filter (artist and price), or Pagination. They all affect the list of products which is the only part of the page that is refreshed. By the way, when the user types in the search box it makes an additional Ajax call to the Search action method because of its autocomplete feature.

As you can see a lot of functionality exists in a single page which is becoming more and more the norm in modern Web Apps. Some apps go to the very extreme by placing the entire app on a single page. These are called SPAs, or Single Page Applications. Building SPAs can be complex, but they provide wonderful UI experiences for the end-user. The Art Shop is an example of a partial SPA.

Views

Let's look at the Products View for the Products action method. Recall that this is the main shopping page in the Art Shop. At the top you see how it handles the suppression of the _Layout file.

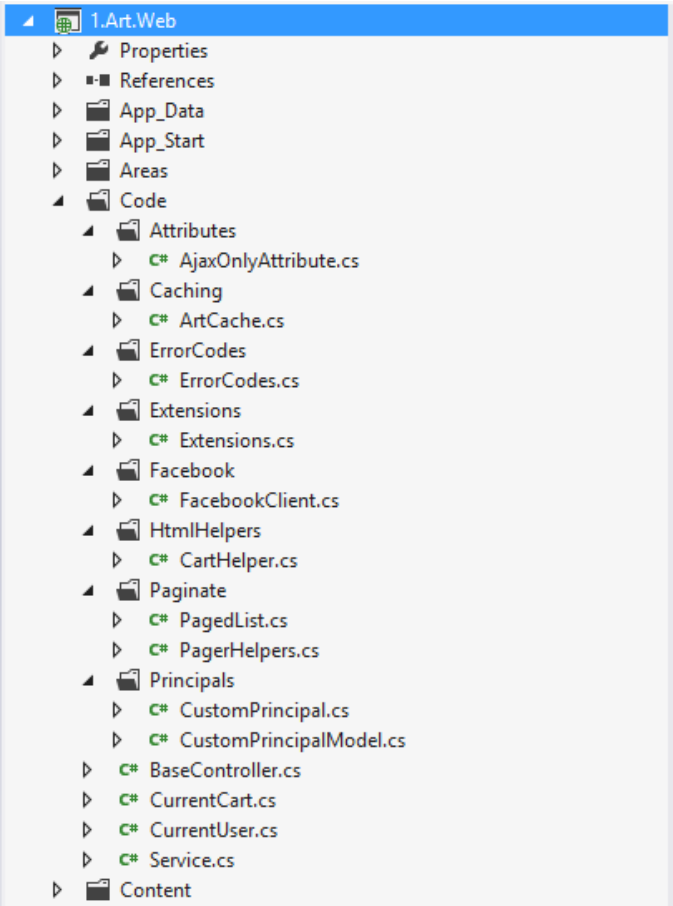
```
@{
    ViewBag.Title = "Shop";
    if (ViewBag.Layout == "No")
    {
        Layout = null;
    }
}
```

Next you see a <form> tag that uses its defaults which are: method="GET" and action="/products". Several hidden variables are inside the form. JavaScript uses these to build the query string with the jQuery *serialize* method (see JavaScript code at the bottom of the page).

A few other items of interest are: the Search box (id="q") uses the jQuery UI autocomplete feature. The Price slider div (id="price-range") uses the jQuery UI slider control. The list of products together with two pager controls is rendered by a partial view named _Products. Finally, at the bottom you find the relevant JavaScript and jQuery code. The comments in the JavaScript code clearly indicate the importance of design patterns in JavaScript and jQuery as used in modern apps.

Code Folder

The Code folder contains several subfolders and classes. The total number of classes is small; most folders contain just 1 or 2 files. Here is an overview.



The components in the \Code folder are not only useful to the Art Shop; many can be readily re-used in your own MVC apps because they solve common problems and requirements. Using the Spark components and classes can save you significant development time.

The \Attributes folder holds a single class named AjaxOnlyAttribute which ensures that an Action method is only called from Ajax.

The \Caching folder holds a single class ArtCache which is a general purpose cache manager that keeps database records in memory. The benefits of caching are 1) increased performance and 2) simplicity. Performance because it reduces the need to retrieve the same data over and over from the database and simplicity because it limits the need for JOINs in the SQL: using cached data, JOINs are resolved in memory.

Caching plays a very important role in *Spark 4.5* and a separate discussion on caching is available later in his document.

ErrorCodes in the \ErrorCodes folder is a lookup class that maps Membership status codes to their proper error strings.

The Extensions file in the \Extensions folder contains a set of handy extension methods that are used throughout the application.

The \Facebook folder holds a FacebookClient class which derives from OAuth2Client. Art Shop supports OAuth authentication and a completely functional example of Facebook authentication is included. By the way, you will need to setup your own developer account on Facebook to get the AppId and AppSecret which you then will need to include in the AuthConfig.cs file.

The native AuthenticationClient in DotNetOpenAuth does not allow customization of the query parameters. Query parameters are required to ask Facebook for additional permissions so we created FacebookClient. In the 'scope' parameter below we are asking Facebook to also send email and user location information.

```
protected override Uri GetServiceLoginUrl(Uri returnUrl)
{
    return new Uri( AuthorizationUrl
        + "?client_id=" + this._appId
        + "&redirect_uri=" + HttpUtility.UrlEncode(returnUrl.ToString())
        + "&scope=email,user_location"
        + "&display=page"
    );
}
```

A large part of the FacebookClient class is involved with properly parsing the data that is returned by Facebook.

The \HtmlHelpers folder contains two files: CartHelper and PagerHelpers. The CartHelper displays the Cart control in the menu bar on the page (it is either black or orange, depending on whether anything has been placed into the shopping cart). The PagerHelpers file displays a pagination control that is used on all pages that show large lists. PagerHelpers.cs contains a second control called PagerText which displays a 'Page 1 of 12' text string but it is currently not used in Art Shop.

The pagination controls works hand in hand with a generic PagedList class which you find in the Paginate folder. All Models with lists of items that require pagination must derive from PagedList. This will allow the pagination controls to properly display the current page state of the page. Models that support pagination include AdminUsersModel, ProductsModel, and three reporting models: ReportOrdersModel, ReportUsersModel, and ReportProductsModel.

In the \Principals folder you find two files that help manage the Principal on the current Thread. These custom Principals allow the application to carry additional user identity information on the current Thread. The default user information is the Name (which in Art Shop is actually email), but these Principals also include UserId, FirstName, and LastName.

Using the Thread to carry additional user information has several important benefits. It reduces the need to pass user information around as function arguments; it removes the need for data base selects to get the user information based on the logged in user's Name (email in our case), and it moves us one step closer to writing stateless apps without the need for a Session object. Details of using custom principal objects are presented in a later section.

Four classes in the root of the \Code folder are BaseController, CurrentCart, CurrentUser, and Service.

BaseController is the base class to all controllers. It contains common functionality that can be used by all controllers.

CurrentCart is a thin wrapper around a cookie based counter which maintains the number of items currently in the cart. CurrentUser is another thin wrapper around the thread based user identity of the currently authenticated user. Both CurrentCart and CurrentUser are handy utility classes with easily accessible static properties. Both are lightweight Façade pattern implementations.

The final class is Service.cs. This single file represents the Service layer which is a façade (Façade Pattern) that is designed to handle more complex data operations with business rules and transactions. Most methods in Service.cs perform transactions that involve multiple domain object types. These transactions are managed by the Unit Of Work pattern. We'll explain the Service layer concept further down this document

Content Folder

The Content folder contains the css files used in Art Shop. The file named app.css is the custom application file. Another important one is the bootstrap.css.

Scripts Folder

This is where all JavaScript and Jquery files are stored. The file named app.js is the custom JavaScript file. Also included are JavaScript files for Bootstrap, History.js and Flot.

Views Folder

This holds the layout file and an error page. The _Alert partial view is the temporary alert box that shows up on the page when an action completes or fails.

Global.asax.cs

The Application_Start initializes SimpleMembership and creates the necessary roles and user accounts if not present (this executes only once, the very first time you run the app). The rest is standard registration and initialization.

The Application_PostAuthenticateRequest is where, for each page request, the authentication cookie is parsed and the data is moved into the custom principal to the thread. The data in Art Shop are: UserId, and User First and Last name. You could add more or less data if necessary. As mentioned earlier keeping user information in cookies and then as custom Principal values on the Thread has many benefits.

The Application_Error is a last resort error handler. All errors are logged to an Error table, together with Request context data. Collecting this data is very valuable and it is simple to implement. The error page will automatically display following and error. The error page is configured in web.config which is the next topic.

Web.Config

In accordance with the *convention over configuration* paradigm the web.config requirements in Spark are kept to an absolute minimum. First we need a connection string. The name should match our

application key name, which is Art. If your application key is Sales, it should read Sales. This connectionstring references the LocalDb file.

```
<connectionStrings>
  <add name="Art" connectionString="Data Source=(LocalDb)\v11.0;AttachDBFilename=|DataDirectory|\art.mdf;
</connectionStrings>
```

When you change to Sql server, then this is where you change the connectionstring.

Two other settings used in Art Shop are the login and error pages. The relevant sections are shown below:

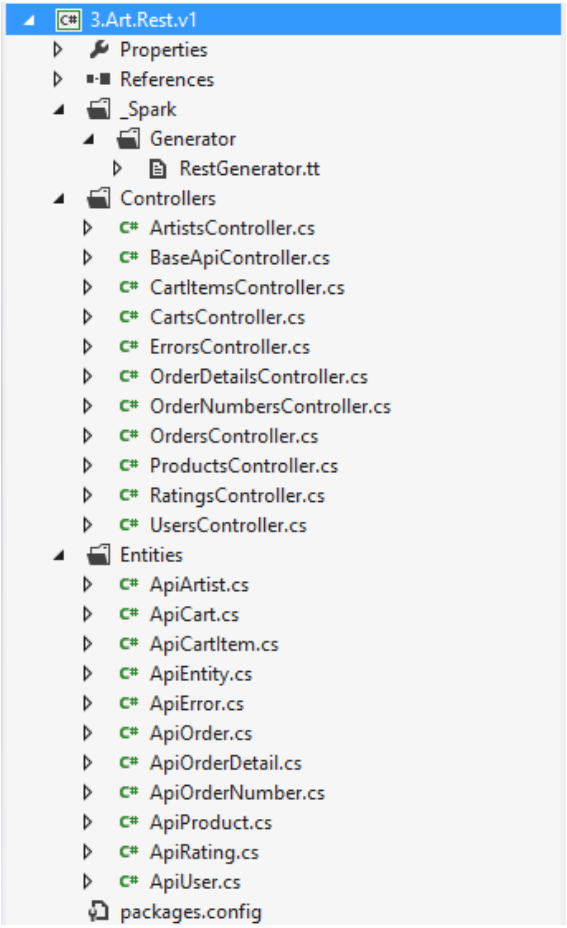
```
<!-- redirect to login page if not authenticated -->
<authentication mode="Forms">
  <forms loginUrl="~/login" timeout="2880" />
</authentication>

<!-- redirect to custom error page -->
<customErrors mode="On" defaultRedirect="~/error">
</customErrors>
```

And this is all; configuration is indeed very lightweight.

REST Layer

Art Shop exposes a public REST interface which was built with Web API. The new Web API is a great addition to .NET.

Similar to the Art.Domain project, the Art.Rest.v1 project also started out with a T4 code generator. It is named RestGenerator and is located in the _Spark\Generator folder. It builds a Controller class as well as an Entity class for each database table. You can see these classes in the screenshot below.



Only the **PRO Design Pattern Framework 4.5** version comes with a code generator, but the resulting files are all available in this REST project. Again, there is nothing to prevent you from building your own REST files without the generator.

Each entity inherits from ApiEntity which you see in the middle of the list of files in the \Entities folder. Open ApiEntity and notice that it has a single property called Href. This is the *identity* system used in Spark REST. Instead of an Id, we use the fully qualified url which uniquely identifies the entity as a resource. For example www.mycompany.com/artists/4 identifies artist with Id 4. Some developers also encrypt the Id which is something you may consider doing in your own work, like www.mycompany.com/artists/33e42f89aa2.

Each controller class inherits from BaseApiController which in turn derives from the Web API ApiController class. BaseApiController initializes Automapper and provides a helper function to construct Href identifiers (urls). The generated controllers offer skeleton Action methods for each CRUD operation or REST verb. It supports GET (single or collection), POST (insert), PUT (update), and DELETE (delete).

These Action methods are meant as templates that need to be programmed. As an example, the Art Shop has fully implemented the ProductsController. In many scenarios only the GET methods are

implemented which allows the client to read, but not write to certain tables. For example, you probably don't want to give write access to the Order table. Each app has its own requirements and typically for each domain object you need to carefully consider the REST access levels.

The optional *expand* query parameter allows the client to expand a foreign key into an object. For example if you open the ApiProduct entity, you don't see ArtistId or Artist Href. Instead it has an ApiArtist. When the client requests a product it returns an ApiArtist that only contains a Href value. This makes it easy for the client to ask for the Artist.

However, with the '`?expand=artist`' query parameter they can do this in a single step, that is, request a Product with a fully expanded Artist entity. Of course, all this is returned in Json or XML which is then transformed to a .NET object. The expand parameter is demonstrated in the Art.Rest.v1.Test project.

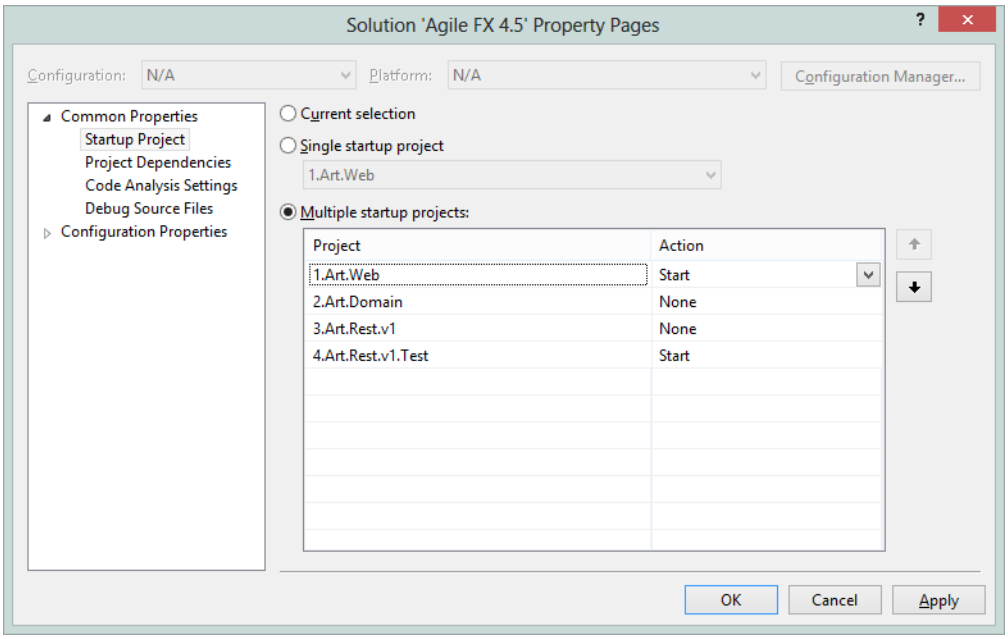
Each Api entity class fully supports this expansion model. For example ApiProduct has 3 expansion dimensions: Artist and two lists: OrderDetails and Ratings.

```
public class ApiProduct : ApiEntity
{
    public ApiProduct()
    {
        OrderDetails = new List<ApiEntity>();
        Ratings = new List<ApiEntity>();
    }
    public string Title { get; set; }
    public string Description { get; set; }
    public ApiEntity Artist { get; set; }
    public string Image { get; set; }
    public double Price { get; set; }
    public int QuantitySold { get; set; }
    public double AvgStars { get; set; }
    public List<ApiEntity> OrderDetails { get; set; }
    public List<ApiEntity> Ratings { get; set; }
}
```

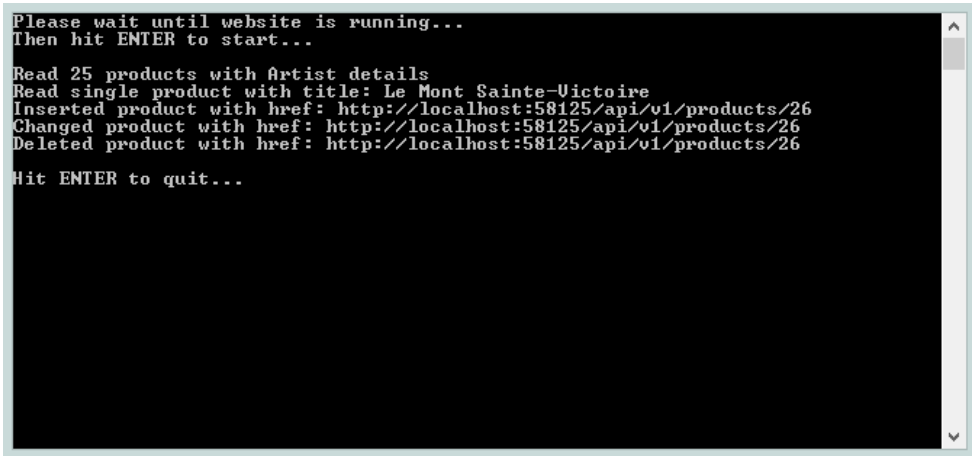
These expansion dimensions are automatically generated by the code generator by examining the primary, foreign key relationships in the database schema. Of course, this can also be done manually in case you don't have the PRO version.

The REST interface is tested by the console application in Art.Rest.v1.Test. The test program demonstrates the use of the new async / await keywords, although in this instance they don't have much effect because the .Result option implicitly invokes a wait meaning it waits for the Task (i.e. thread) to complete. In larger multicore or multiprocessor scenarios async / await can have a dramatic effect on performance.

As mentioned earlier, executing the tests for the REST interface requires that two projects are running concurrently: 1) the Art.Web project and 2) the Art.Rest.v1.Test project. To do this you right click on the Spark 4.5 *solution* and select Properties. This shows the dialog below in which you select two projects to start.



Click OK. Then run the application. The browser and the console window open. Wait until both a fully loaded. Then in the console window hit the Enter key. It may take a few moments for REST to start, but you should see the results shortly, like so:

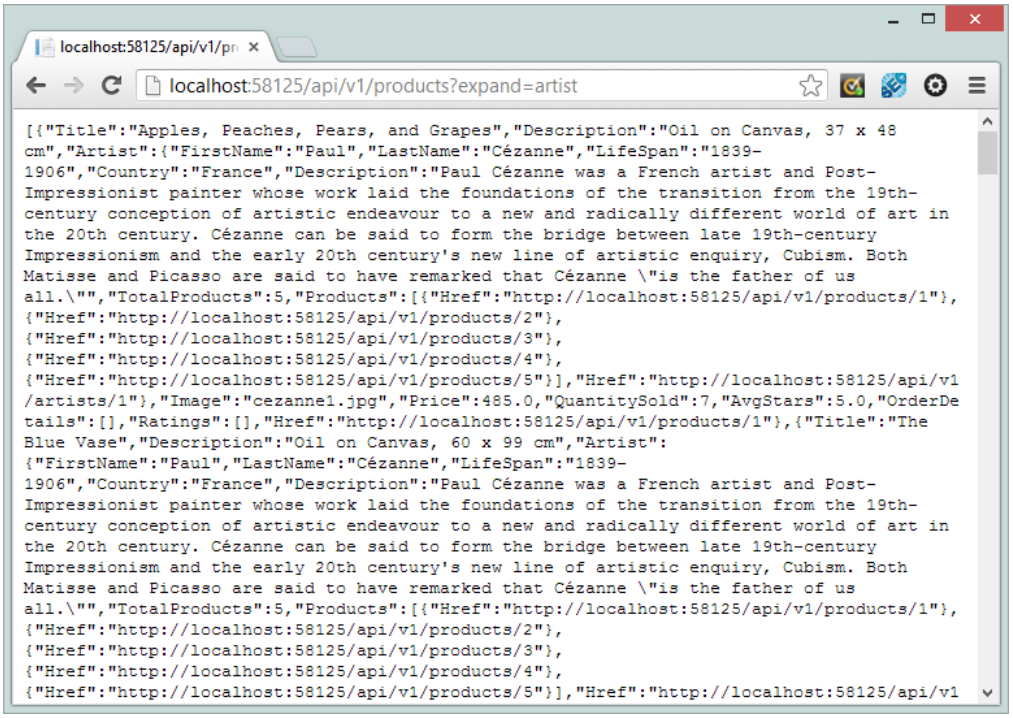


As you can see it performed two selects, an insert, update and delete. The first select returned 25 products, all with fully expanded artists. This proves the REST interface is up and running.

The REST urls are simple to understand. It follows the pattern www.domain.com/api/v1/entities/id where 'entities' (plural) is one of the business entities, such as, products, artists, users, etc. The REST verb determines the action requested. All options are listed in the table below:

Url	Verb	Result
www.domain.com/api/v1/entities	GET	Get all entities. Optionally add parameters, like expand, page, filter, and sort (the last 3 are not implemented).
www.domain.com/api/v1/entities/4	GET	Get entity with id 4
www.domain.com/api/v1/entities	POST	Insert a new entity
www.domain.com/api/v1/entities/4	PUT	Update entity with Id 4
www.domain.com/api/v1/entities/4	DELETE	Delete entity with Id 4

Another way to test the REST API is from the browser. Simply enter the above urls in the command line and you'll receive the JSON results.



By the way, several browsers request XML media type by default. JSON is somewhat simpler to view, so the XML formatter was removed from Art Shop (in global.asax.cs). Content negotiation allows the server to return another format which, if XML cannot be returned, JSON will be selected. In your own apps you probably want to include the XML formatter and support both XML and JSON.

A final note: testing REST with a browser does not require the REST test program. You can test with only Art.Web running.

Design Patterns

Next we'll review the core design patterns in Spark. The core patterns are a fundamental part of the platform. Of course, other patterns are being used but the main ones are the 'building blocks' of Spark. They include: Active Record, Repository, Unit of Work, Façade, DTO, and CQRS.

Active Record

The Active Record is an Enterprise Pattern. On Fowler's web page he defines Active Record as "an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data". That exactly describes our domain objects; one class for each database table and one instance for each table record. The importance of Active Record to Spark cannot be overstated. It is the essence of the platform.

Repository

The Repository pattern is also an Enterprise Pattern. It is defined as a group of classes where query construction takes place. Its goal is to present a clean and consistent query interface to the client. When designed properly it also helps to reduce code duplication.

In Spark 4.5 each domain object has its own Repository. For example, Product has a Products repository which accepts and returns only Product domain objects. Each repository has a well-defined interface that allows all necessary database operations including the selection of single or multiple objects as well as insert, update, and delete domain objects.

The Repository pattern is also very beneficial in testing scenarios. Repositories provide central access points to the database. By *mocking* (mimicking) repository objects (using their interfaces) we can perform tests without the need to access a real database making testing fast and repeatable.

As an aside: both repository and domain objects have public Insert, Update, and Delete methods. They both work the same and, in fact, they execute the same lower level methods. However, for consistency it is best to invoke the repository Insert, Update, and Delete methods rather than the domain versions. For example, suppose you have a domain object called 'product' that needs to be inserted into the database; instead of product.Insert() it is better to use ArtContext.Products.Insert(product).

Unit of Work

The Unit of Work pattern manages database transactions that involve multiple domain objects. It coordinates the execution of several database operations and ensures that the entire group either succeeds or fails.

Suppose a transaction involves two operations: 1 delete and 1 insert. The transaction starts and the delete is successful but the subsequent insert fails. The Unit of Work will correct this situation and makes sure the first delete is rolled back (i.e. undone), so that the database does not end up in an incorrect or unstable state.

The Art Shop includes a schoolbook example of the Unit of Work pattern, namely when the user places the order. The UnitOfWork class coordinates several delete and insert operations for 4 different domain objects (Cart, CartItem, Order, and OrderDetail). If something goes wrong during the transaction, the entire order is cancelled and the shopping cart tables are restored to their original state.

Façade

Façade is a simple but very effective GoF pattern. It offers clients a simple API to a complex set of subsystems. The Service layer in the Art Shop is a good example: it hides the complexity of intricate business rules and transactions.

The details of the service layer are discussed later in this section.

DTO

DTO (Data Transfer Object) is an Enterprise Pattern. A DTO is an object whose task it is to carry data from one place to another. They have no methods, only fields and properties.

The Model objects in MVC are DTOs (note: the Models are also referred to as ViewModels, but they live in a folder named \Models). Their only purpose is to carry data from the Action method to the View. Many properties are strings holding formatted numeric values such as "$2,322.99" or "300psi". These values are ready to be inserted at the appropriate places in the View without any further processing or formatting on the view.

CQRS

CQRS is a pattern that was inspired by the concept of CQS (Command Query Separation) which was devised by Bertrand Meyer, an expert in programming languages. He observed that methods come in two flavors: those that performs actions, called a Command, and those that returns results, called a Query. Command methods don't have return values, whereas the Query methods do return values.

CQRS implements the CQS concept and it is typically promoted for use in large-scale, distributed systems. These systems are built around DDD (domain driven design) and contain implementation details such as Message Queues and an Enterprise Service Bus; in short, big stuff.

However, the CQRS pattern is very interesting for any application, large or small. It has been described as a *fire-and-forget-it* pattern: in other words, issue the command and forget about the results. This is best explained with an example from the Art Shop.

The shopping cart in the Art Shop allows the users to adjust the quantity of the cart items or delete the line item altogether. In most web apps the change would be posted back to the server and a new shopping cart page will be rendered. On the server, the request will wait for the database change to take effect. If the database is busy the user will have to wait. This is exactly the problem that CQRS solves.

With CQRS, instead of waiting for the database, it spawns a new thread that will perform the database action, and returns immediately without confirmation of the results. This is the fire-and-forget notion. So, it *assumes* everything is going to work just fine, so it is an optimistic pattern. All this is based on real-world experience, because, in reality the chance of success is very high and, if it fails, well, you 'deal with it'.

There are ways to ensure that a request (i.e. command) ultimately happens, now or later, by using an ESB (Enterprise Service Bus) with Queuing system. For many applications, including Art Shop, this would be overkill.

CQRS chooses to provide all users a better interaction experience by taking a (tiny) risk of an action not working. This seems like a good tradeoff to us.

The shopping cart in the Art Shop implements CQRS. The CartController has three Cart methods: the HttpPost Cart method adds a new item to the cart, the HttpPut Cart method updates the cart, and the HttpDelete Cart removes a line item. All are AjaxOnly methods, but this is not a requirement. Each implements the CQRS pattern by updating the relevant variables and then spawning a new thread in which the database is updated that *hopefully* will succeed. In all our testing of the app, we have never encountered a failure and the interaction experience of the cart page is phenomenal.

Miscellaneous Patterns

Here is a list of other patterns found throughout the Art Shop code base. They are grouped by category: MV patterns, GoF patterns, and Enterprise patterns.

MV* Patterns

MVC: You couldn't miss this one. MVC is the pattern on which ASP.NET MVC is based.

GOF Patterns

Iterator: Anywhere you see a foreach or a LINQ statement an Iterator pattern is involved.

Observer: Anywhere you see events and event handlers, an Observer pattern in involved.

Factory: The GetFactory in Db.cs in Art.Domain is an implementation of the Factory pattern. In fact it is a meta pattern: it is a factory of factories. The DbProviderFactory instance named *factory* also is an implementation of the Factory pattern. It creates Connections, Commands, and DataAdapters.

Prototype: The overloaded constructor of the Entity classes (domain objects) is a type of Prototype. It returns prototyped objects that are initialized with the default values of the corresponding database table. It is used, for example in the Checkout action method in the CartController.

```
// ** Prototype pattern

var order = new Order(true) { OrderDate = DateTime.Now, TotalPrice = grandTotal, UserId
var details = new List<OrderDetail>();
foreach (var item in items)
{
    // ** Prototype pattern

    details.Add(new OrderDetail(true) { Price = item.Price, ProductId = item.ProductId,
}
```

Notice the boolean true arguments in these constructors.

Singleton: ArtContext is a Singleton. Although implemented as a static class with static methods, only one instance exists for the entire duration of the application.

Enterprise Patterns

Identity field: Saves an Id value in a domain object to maintain the identity between the domain object and the database table. All domain objects implement this pattern with their Id fields.

Foreign Key mapping: Maps a foreign key relationship to an association between domain objects: the parent and the child. As an example the ApiProduct class in the Art.Rest.v1 has three of these mappings: an association to Artist and to two lists: OrderDetails and Ratings. These are all objects that in their database tables have a ProductId.

Service Layer: The Service layer offers services to the client and coordinates their actions. In Art Shop the services are the more complex actions, those that involve multiple domain types and transaction management. The Service layer lies as a thin veneer over the Repository layer. The Service.cs file is where this layer lives.

Lazy Load: Lazy loading takes place in an object that knows how to load the data when called upon. The goal is to delay the data loading as long as possible. There may even be cases where it is never called, so no resources were wasted. The ArtCache object is a good example. It only loads the Artist cache when needed. The Entity base object in Art.Domain project also initializes itself only when called for the first time. Both of these involve data access and memory. What better way to conserve resources when not used?

JavaScript and jQuery: This is not a pattern per se, but we like to stress the importance of JavaScript and jQuery in any modern web application, including Spark apps. The Art Shop shows how to build highly interactive, beautiful apps. In the future the use of JavaScript is only going to increase, because it is the only language that runs on the client.

There are two ways to look at JavaScript: as a rinky-dinky scripting language or as a rich language that supports many modern language features. In reality there is a big difference between applying some script to change the colors of a DOM object, versus building complex JavaScript apps based an in-depth understanding of the language with features such as such as closures, prototypes, chaining, etc.

JavaScript is a small but flexible language. Just recently, dozens of Patterns and Idioms have evolved to make the language more manageable. It is important that JavaScript developers have a good grasp of these patterns. There are similarities between patterns in .NET and patterns JavaScript, but there are many significant differences. To learn more we suggest you explore our *Pro JavaScript and jQuery Design Patterns*, which is available on our website at www.dofactory.com.

Best Practices

This section discusses the best practices of Spark 4.5. They are Convention over Configuration, Constraints over Lookups, Caching, and JSON over XML.

Convention over Configuration

Developers have been building more and more flexibility into their systems by moving behavior, preferences, and settings into XML configuration files. This reached a point where some developers felt it went too far as it became nearly impossible to correctly configure an application or be able to simply start an application. *Convention over configuration* attempts to correct this by using reasonable defaults, i.e. conventions, over explicit configurations.

Convention over Configuration has its roots in Ruby on Rails (RoR). The idea is to simplify application development by following some well-defined standards and conventions. The success of RoR hasn't gone unnoticed to other systems, including PHP, Java, Python, and .NET. These are now embracing the same paradigm and this is also clearly visible in MVC.

You may not be aware of the conventions in MVC because they seem so *natural*. Here are some: 1) all controller classes are suffixed with 'Controller', but you reference them without the word Controller (for example in route definitions and HTML helpers). 2) controller action method names are the same as their Views. For example, action method Widget will have a view named Widget. 3) the ApiController (in Web API) maps the HTTP verbs GET, POST, PUT, and DELETE to action methods that start with the same words, i.e. Get, Post, Put, and Delete. So, for example, a POST request will go to Post or PostWidget. 4) at a larger scale, the routing system in which urls are mapped to Controllers and Action methods also follow some clear conventions.

The MVC conventions are simple rules to live by. They feel natural and you will probably never have to change these and if you do you *go against the grain*.

Spark embraces convention over configuration; these are some of its core conventions: 1) a set of database modeling and naming rules, 2) a 4-layer architecture build around these patterns: Repository, Façade, Unit of Work, DTO, and CQRS, and 3) adoption of the Active Record pattern in which the domain objects and their properties are named according to the database tables and their columns. All in all, a simple, light-weight package, that is easy to learn, easy to apply, and easy to be successful with.

Spark 4.5 has no configuration requirements other than a database connection string in web.config. Art Shop only adds a default login page and error page to the configuration.

Constraints over Lookups

The *Constraints over lookups* convention pertains to the data model. The concept may seem trivial but it has important benefits. Here is a simple example. Suppose you have a User table with a Gender field. The possible values are Male and Female. To ensure that no incorrect values are entered you put a column constraint on Gender only allowing Male and Female (or M and F).

You could also model this with a separate lookup table called Gender and replace Gender in the User table by GenderId. The Gender table has only two rows: Male and Female. The Id of Male is 1 and Female is 2. GenderId in User now has either a value of 1 or 2. Which one is best: constraints or lookup?

You will probably agree that the first one is better because 1) it is simple, 2) does not require table JOINS, and 3) there are only two values, so there is no reason for a separate table. These are all valid reasons.

Now suppose that the User table has a Role field. Possible values are Member, Reviewer, Approver, and Admin. Perhaps in the future some other roles will be added. Many will argue that a Role lookup table is the better model and therefore the User table needs a foreign key RoleId field. However, this is *not* what Spark recommends. It favors constraints over lookups. If a new role is added, you simply add it to the list of constraints (i.e. allowed values).

The main benefit it is that it keeps the SQL simple without the need for repeated JOINS, which in turn improves performance. DBAs will also love you for this because GenderId : 1 is more difficult to understand than Gender : Male.

Caching

Suppose that you did implement the Role lookup table (described above) rather than a constraint. How would you keep your SQL simple and limit the number of JOINs? Caching is the answer. Simply read the Role table into memory and store a dictionary (hash map) of role values. For every User domain object the role is resolved by locating the role value in cache. This is extremely fast and efficient. Role is easy to cache because the possible values never change. And if they did, you refresh the cache.

Next we go beyond lookup tables. Suppose that each user in the User table belongs to a company. The User table will have a CompanyId column (foreign key) that references the Company table. To get a list of users and their companies we need to JOIN the two tables. However, Spark prefers simple table selects, ideally without JOINs. This can be accomplished by caching. Simply keep the Company domain objects as a dictionary in a memory cache. Then for every User locate the Company from cache and you have all the data you need. You need to update the cache whenever a company gets added or changes. It is important that the cached Company objects are available as a *dictionary* with the Id value as the key which offers very fast access.

How have the *Constraints vs. Lookups* and the *Caching* conventions affected the design of the Art Shop application? The database does not have a single lookup table and the Art Shop app does implement caching albeit in a very limited way (note: if this were a real application with a larger database we would cache additional tables as well). With this in place, we were able to write the complete store without a single JOIN.

To be honest, some Reporting and Ad hoc queries do include JOINs, but this is to be expected when performing analytics rather than transactions. Interestingly, Reporting and Ad hoc queries are much faster because of the aggregate columns that are implemented in the model.

There is no need to go extreme and ban JOINs altogether, but it greatly simplifies development and accelerates data access.

Tables that are great candidates for caching are the ones that 1) don't change very often, and 2) are peripheral tables to the main tables. In the above scenario User is the main table (the focus is on the maintenance of user records) and Company is the peripheral table. In other scenarios, the User table is the peripheral table and is the one to be cached. An example of this is a discussion forum, where the focus is on the postings and the users are peripheral to them. So, users would be cached in that scenario (you can think of StackOverflow for example).

As a general rule try to cache as much as possible as this makes your application 'fly'.

JSON over XML

JSON (JavaScript Object Notation) is today the preferred data transport protocol over XML. XML is much harder to read by humans and it is bulky. JSON, on the other hand is light-weight and simple to read and understand. We favor JSON over XML, but in REST scenarios it is best to offer both which allows clients to choose their preferred format.

Tools and Components

This section discusses the tools, libraries, and components used in Spark 4.5.

Tools

Here is a list of open source tools and libraries used to build *Spark 4.5* apps:

Tool	Type	Description
Automapper	.NET	Library that helps mapping objects to other objects
Json.NET	.NET	Library for fast JSON operations
Bootstrap	CSS	Front-end CSS Framework for faster and easier development
jQuery	JavaScript	Versatile JavaScript DOM Traversal Manipulation library
jQuery UI	JavaScript	JavaScript effects, controls, and widgets built on top of jQuery
History.js	JavaScript	Supports HTML5 History/PushState on all browsers
Flot	JavaScript	HTML5 charting library

This list is a good representation of the type of tools used in building modern ASP.NET MVC apps. Some projects also introduce a JavaScript MV* library to bring more structure to their JavaScript code, especially when their JavaScript code grows large. Examples of MV* libraries are: Backbone, Knockout, and Angular. Knockout is an MVVM library that is most popular with .NET developers.

The JavaScript in Art Shop is well-structured by applying some popular JavaScript patterns. You will see that the JavaScript code has numerous pattern comments. If you wish to learn more we suggest you check out our popular *Pro JavaScript + jQuery Patterns* package (available on our website at www.dofactory.com).

Next are some details on the tools and libraries used to build the Art Shop.

Automapper

Automapper is a.NET library that does only one thing but it does it exceptionally well; it copies data from one object to another. Moving and manipulating data from one object to another is a frequently occurring process in well-architected systems. Unfortunately it leads to repetitive, boring code, but Automapper makes it fast and easy.

Automapper is used extensively in MVC controller classes where domain objects are copied to models (DTOs) and vice versa. Anytime you implement the DTO pattern you will need to have code that copies the data. Automapper keeps you from having to code it by hand which saves time and it keeps your code concise and clean.

Before it can do its job, Automapper needs to know how to map one object type to another type. This is accomplished with the CreateMap method which 'teaches' Automapper how to do this. This needs to happen only once for each pair of types. In Art Shop this is done in the static controller methods. Here is an example:

```
public class CartController : BaseController
{
    static CartController()
    {
        Mapper.CreateMap<Cart, CartModel>();
        Mapper.CreateMap<CartItem, CartItemModel>();
    }
```

As an alternative you could also run all Automapper CreateMap calls at a central place for the entire application. A good place would be an AutomapperConfig file which would be added to the \App_Start folder. Then invoke this once during application startup from global.asax.

Once Automapper knows how to map the necessary types, it is just a matter of calling the Mapper.Map method. Automapper is a valuable tool and an important addition to Spark 4.5

Json.NET

Json.NET is a JSON framework that performs fast JSON serialization and related operations.

Bootstrap

Bootstrap is a front-end CSS framework that makes the design and development of Web apps faster and easier. It is designed to help build beautiful HTML pages including page layouts, forms, buttons, lists, navigation, typography, icons, and other interface components. Bootstrap also comes with a set of JavaScript controls such as a carousel, popovers, tooltips, tabs, and more. The carousel is used on the Art Shop home page.

Bootstrap is very popular. At the time of this writing, it is the most visited project on Github. And for good reason, it significantly accelerates front-end development.

In mature development shops with an experienced team you will discover that most of the development time is spent on the presentation layer. And this is great. The reason is that development of the other

layers can be fully standardized and implemented quickly. What makes any application unique is the code in the presentation layer; not the back-end.

Spark 4.5 brings a similar level of maturity to a project. It is driven by clear standards and conventions. This allows the application foundation to be constructed quickly and easily, specifically the Repository, Domain, and Data Access Layers. In fact, with the code generator in the PRO version of the Design Pattern Framework you can reduce this time to just a few minutes.

Similar to the mature development shops, Spark allows you to focus on the Presentation layer, which is where you want it to be because this is where the *essence* of your application is being created.

What Bootstrap does is apply standards and conventions to the HTML and CSS. The applications are attractive and the time savings can be phenomenal. Not to mention the clarity and consistency of the resulting front-end code. Bootstrap can give your apps a huge front-end development boost.

jQuery

jQuery is by far the most popular JavaScript library. It is used by millions of web sites. At its core jQuery is a fast HTML traversal and manipulation library with an easy-to-use API that is based on CSS selectors. jQuery also offers animation, event handling, and Ajax. It also offers easy-to-use plugin architecture.

jQuery is an integral part of the Spark front-end development.

jQuery UI

According to the jQuery UI development team jQuery is a "curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript library." Indeed, it offers a rich set of interface components. In Art Shop the *autocomplete* and the *slider* features are used on the main Shop page.

History.js

HTML5 allows you to programmatically manipulate the browser history using methods like PushState, PopState, and ChangeState. History.js extends this functionality to all browsers, even the ones that do not support HTML5.

The Art Shop has wrapped the History.js library in a small helper JavaScript library that resides in the app.js file. The JavaScript for browser history manipulation has its own namespace: Patterns.Art.History.

Flot

Flot is an attractive and easy-to-use jQuery charting plugin that can be downloaded from google.com. It requires a modern browser that supports HTML5. To render a pie-chart you must also download the jquery.flot.pie.js file. It is already available in Art Shop if you need it.

Components

In Spark 4.5, the word 'components' refers to code elements in the Presentation layer. We like to point out the following five components areas.

Caching

The ArtCache file is a good starting point for building your own caching. It has built-in functions to clear a given cache or all caches. Dictionaries with domain objects are added to the cache through lazy loading, meaning only when the data is requested will it load the data. Subsequent calls are returned the cached data. Once this cache is cleared, the same lazy loading process applies again.

Caching works best for data that never or rarely changes. In Art Shop only the Artists are cached. However, in a real-world app with larger data sets, we would definitely have other tables in cache as well. The trick, of course, is to keep the cache in sync with the database. If, for example, your app has a User cache and a User record changes it is important refresh (or evict and then lazy load) the User object in cache also.

Although limited in Art Shop, caching is an important part of Spark. It significantly simplifies the SQL and it greatly speeds up the application. SQL data access is almost always the slowest part of any app.

In our experience, caching is a greatly underutilized technique by developers building high performance apps. Not only memory caches but also the built-in MVC output caching facilities. At Stackoverflow.com, for example, they are caching anything and everything they can.

Custom Principals

Custom principals are very useful in web applications. They allow threads to carry extra information, for example about the currently logged in user. This makes this information readily available anywhere in the code without the need to query the database or pass this information as arguments into numerous methods.

Authenticated users have their Name (email in our case) available anywhere via this globally accessible property: Thread.CurrentPrincipal.Identity.Name. But, more often you will need the UserId and we don't want to hit the database all the time to simply get the UserId via the email (or Name) value.

This requires that for each request the thread is assigned a custom principal that carries the information we need. The place for this is Application_PostAuthenticateRequest. Normally, this would require a database select, but we are avoiding this by storing the relevant values in an encrypted cookie. All this is demonstrated in Application_PostAuthenticateRequest in global.asax. As you can see there are no database requests. The user information is retrieved from the authenticated user's cookie and added to the custom Principal.

When is this information first set in the cookie? This takes place when a user logs in or signs up. You can find the pertinent code in the SetCustomAuthenticationCookie method in AuthController. It will add encrypted user information to the cookie. In a subsequent request this is then received by the PostAuthenticationRequest method in global.asax.

Where are the Principals used? The CurrentUser static class offers easy access to the user information that is stored in our custom Principal. To find all places where this is used simply search for

CurrentUser.Id. Note that CurrentUser is only available to Art.Web whereas the custom Principal is useful also to the Art.Domain project.

To see where it is used in the Art.Domain project open the Core.cs. It has a virtual TryGetUserId method which, as its name implies, tries to get the current UserId. The UserId is used to populate the CreatedBy and ChangedBy audit columns in the database. With non-authenticated users there is no UserId so they default to null. However, authenticated users have their Id now on the Thread. This also works in multithreaded situations (which is important because the original Request object or the Session object is not available at that time). With the help of the custom Principal the UserId is always easy to get.

Service layer

The Service layer in Art Shop is represented by the file Service.cs in Art.Web. Its methods perform actions involving one or more repositories. These actions are managed as a single transaction by one of the Unit-of-Work classes. There are no hard-and-fast rules about what goes into a Service, but in general these include application specific data operations that involve multiple domain objects and complex business rules.

The business rule requirements in the Art Shop are minimal, but if they had been more complex the Service layer is where you would these. Art Shop does perform multiple transactions that are managed in the service layer. The service layer is also the ideal place to manage situations where you need to access multiple databases at the same time. The DistributedUnitOfWork pattern will help with that.

What is the best place to keep the Service.cs file? In Art Shop the Service layer is used only by the MVC presentation layer, so having this file in the same project is fine. However, there are situations where the services are shared by multiple projects, such as the MVC project and the REST project. In that case you would transfer the Service.cs file over to Art.Domain.

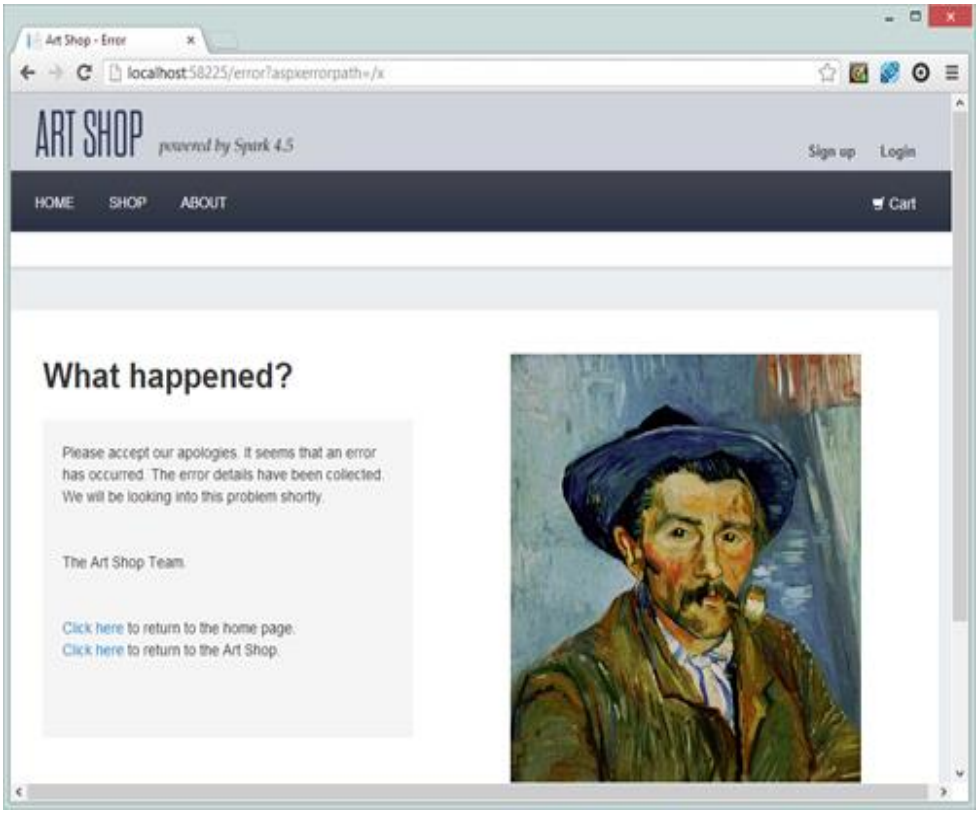
Let's push the envelope a little further. Suppose your app uses two databases, each represented by their own project: Art.Domain and Inv.Domain (Inv for inventory database). In that case you would transfer the Service.cs file to its own Art.Service project which would then be referenced by both the MVC project and the REST project. The Art.Service layer will reference both Art.Domain and Inv.Domain projects. The Spark 4.5 architecture can easily handle any of these scenarios.

Error logging

All unhandled errors in Spark 4.5 are captured by the *last resort* error handler which is the Application_Exception method in global.asax. It saves the details of the Exception as well as the Http Request details to the database in a table named Error. After that a redirect takes place to the url specified in the web.config. In our case the Error View in the root \Views folder. Here is the relevant web.config snippet where this is configured.

```
<!-- redirect to custom error page -->
<customErrors mode="On" defaultRedirect="~/error">
</customErrors>
```

Here is the error page for Art Shop:



Error logging is very helpful in determining the exact cause of the error. The IP address of the request is logged as well with some other request values. If it turns out that a hacker or a badly behaving crawler is at work, you can block requests coming from this IP address.

It is helpful to include an admin-only page to view the errors and their details. This page should also have a purge button which clears the error records from the database when there are too many.

In general, try/catch exception handling should be limited to situations where you know the application can handle the exception and restore itself to a stable situation. Most often this is not the case, so the next best thing is to log all the details of the error in the last resort error handler in the Global.asax file.

Sometimes you use a try/catch blocks to capture the context of the error condition, such as the values of local variables. This will be helpful in tracing down the cause of the exception. For an example of this see Db.cs in the Domain project. Here we want to capture the Sql and the parameters values before they get lost. Once captured, the exception gets re-thrown so that the details are logged in the Application_Error method.

JavaScript and jQuery

The importance of using JavaScript and jQuery to build a great user experience in modern-day web applications cannot be overstated. In the Art Shop, on the main shopping page, not a single page refresh occurs after the page is displayed; this can only be accomplished with the help of JavaScript and jQuery.

The JavaScript code follows modern JavaScript and jQuery script design patterns. JavaScript patterns is a separate topic altogether and is outside the scope of the Design Pattern Framework. To learn more about JavaScript patterns as well as SPAs (Single Page Applications) we suggest you check our *Pro JavaScript and jQuery Patterns* package which is available on our website.

Let's examine some of the JavaScript. A namespace called Patterns.Art.Products holds all the JavaScript code for this page. This is the JavaScript namespace pattern which helps avoid 'global namespace pollution'. In it two other namespaces Utils and History are referenced (both are defined in app.js). The browser history is initialized and pager, filter, and slider controls are activated.

The ajaxSubmit function is called when the form is submitted. It pushes the new state to the browser (this is called pushState in HTML5), shows the Reset All button, and uses jQuery to make an Ajax request with a properly serialized query string. When the results are coming back to the browser, the product list is replaced and the pager controls are re-activated. At the bottom of the JavaScript code you see that the Products JavaScript module is started when the page has completed loading.

If you are a casual user of JavaScript or jQuery, this code may seem lengthy and complex, but given the functionality that it offers, it is actually amazingly compact with just a little over 100 lines of code. Again our *Pro JavaScript and jQuery Patterns* package will get you started quickly and easily. This package is a great investment because your career is going to involve a lot more JavaScript as functionality moves from the server to the client. And HTML5 will just accelerate this process.

Building apps with Spark 4.5

This section describes how to get started with *Spark 4.5* and create your own MVC app similar to the Art Shop reference application. This step by step guide will get you up and running in no time.

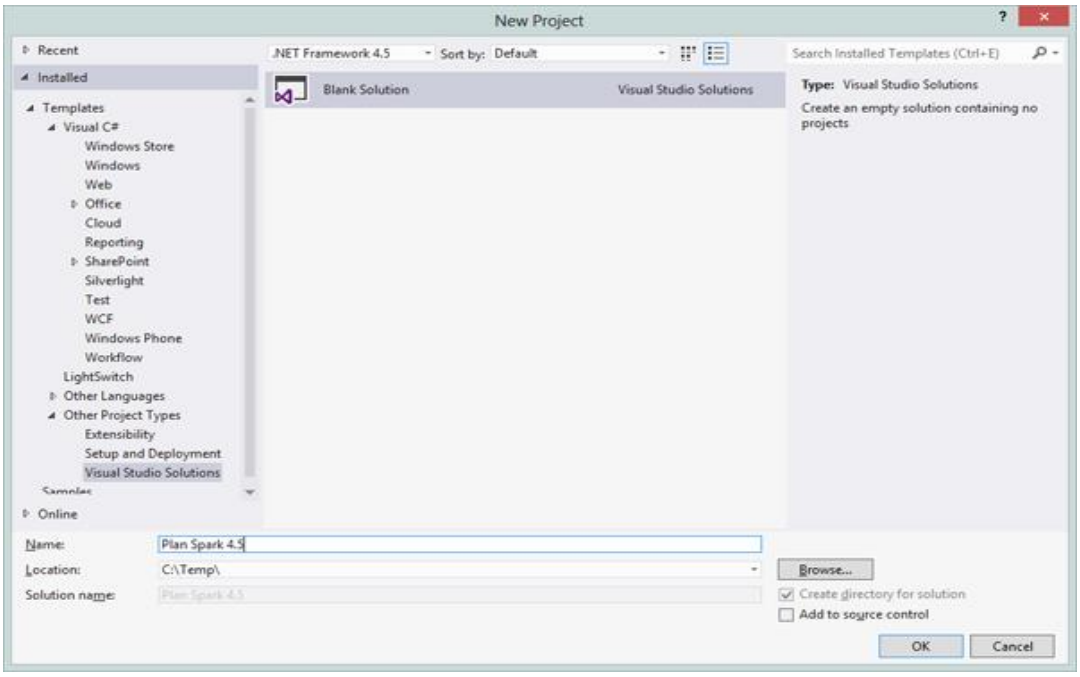
Let's assume the new application is a project planning application. A convention in Spark is to select a short application key name (2, 3, 4, or 5 letters at most) which is used throughout the app. You can use an acronym or abbreviation, such as, Trip, Go, Fin, Ins, Plus, Sales, etc. For this project we will use application key *Plan*.

The name *Plan* will occur in multiple places throughout the solution, including project names, namespace names, and some class names, such as PlanContext and PlanDb. The connectionstring name in web.config will be Plan as well and the database name will also be Plan -- although this can be changed if you like.

For your convenience the Plan Spark 4.5 solution is included. To limit the download size the 'packages' folder has been removed, which means that when opening Nuget you will not see any packages currently installed. For this reason it is best not to use this solution as your starter project. Simply create your own and follow along with the discussion below.

Solution and Projects

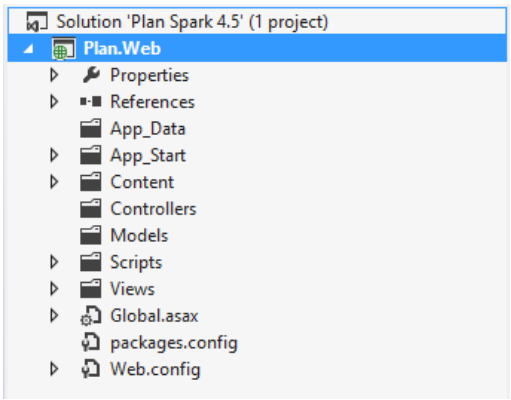
Step 1: Create a Blank Solution. Use any descriptive name. We call it Plan Spark 4.5.



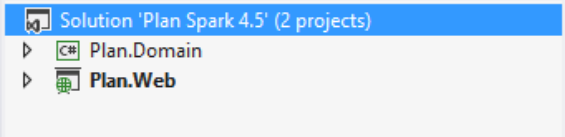
Note to VB users: For all projects that you will be creating within this solution please be sure that Option Strict is Off.

Step 2: Within this solution add an MVC 4 Web application project named *Plan.Web*. Use the Basic MVC template and the Razor view engine. Optionally include a Test project if you anticipate writing unit tests.

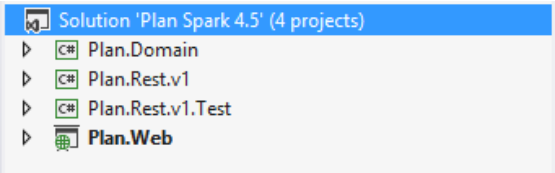
Note that the Basic template will not create an AuthConfig.cs file in \App_Start. You can add this manually (possibly copy it from Art Shop). Alternatively select the Internet MVC template and then remove the extraneous files. This is what the Web project looks like with the Basic MVC template.



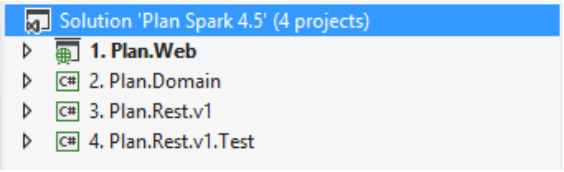
Step 3: Add a new class library project named *Plan.Domain*. This is where you will create (or code-generate) three layers: Repository, Domain, and Data.



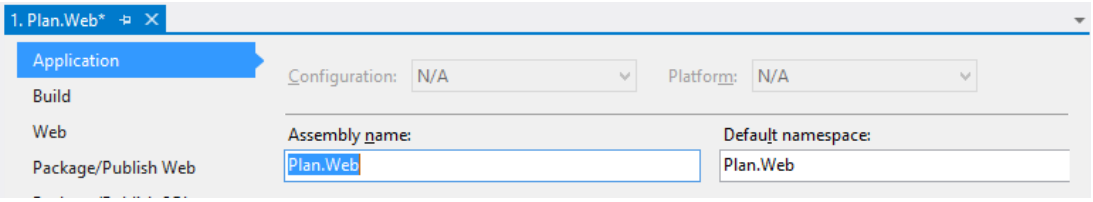
If your application is going to expose a REST interface add two more projects: a class library project named *Plan.Rest.v1* and a console application named *Plan.Rest.v1.Test* to test the REST interface Here are the results.



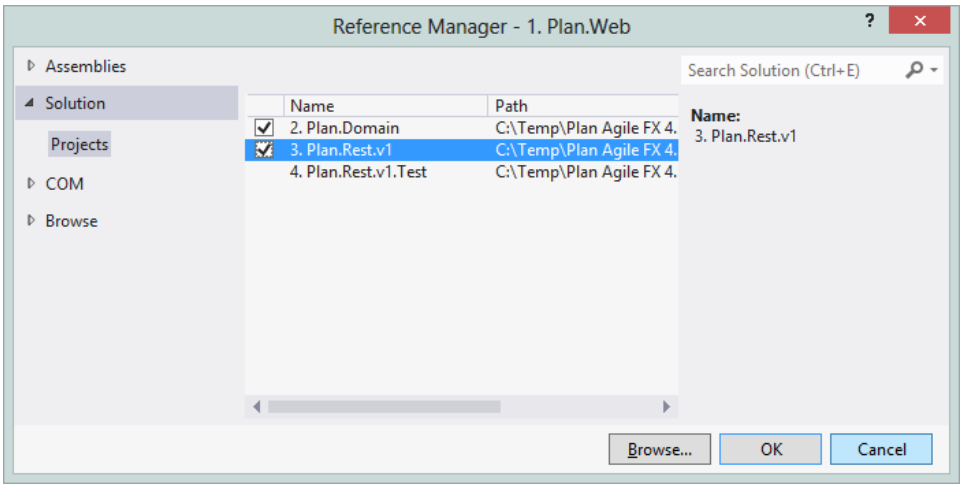
They are sorted alphabetically which is not quite the order in which we think of them. You can rename the projects by prefixing them with a number so that Visual Studio arranges them in the order that we desire. Here is what the results look like.



Please know that renaming the projects does not affect the default namespaces for the projects. For example, right click on the Plan.Web project and examine its Properties. Notice that neither the assembly name, nor the default namespace were changed.



Step 4: Now is a good time to add project references within the Solution. The *Plan.Web* project references *Plan.Domain* and *Plan.Rest.v1*. And *Plan.Rest.v1* references *Plan.Domain*. That is all you need.



Of course, other .NET assemblies will need to be referenced but they can be added over time.

Step 5: Next, add the external tools and libraries that are needed. Use Nuget to install Bootstrap in *Plan.Web* and Automapper in *Plan.Web* and *Plan.Rest.v1*. Also add Json.NET to *Plan.Web*, *Plan.Rest.v1* and *Plan.Rest.v1.Test*.

Your application determines what JavaScript tools and libraries are needed. Here is what we used in the Art Shop. First of all: jQuery and jQuery UI which are available by default in Rent.Web. You may want to update these with the latest versions from Nuget as well. Next, get jquery-history.js (needed for browser history support) and jquery-migrate.js (needed to make jQuery UI autocomplete work) from the Art Shop or retrieve these from github.com. If you need HTML5 charts in your app then also include jquery.flot.js and jquery.flot.pie.js. Get these from Art Shop or download from google.com. As an alternative to may want to consider Angular.js for supporting the browser history (using the $location service) – but this requires that you are familiar with Angular.

We now have a great starter solution in which Plan.Web project will host the Presentation layer and the Plan.Domain project is going to host the three remaining layers: Repository, Domain, and Data Access.

Creating the Database

Next we'll create a database. Here we will create a LocalDb database in Plan.Web, but it is more likely that you want to create a SQL Server database. Either one is fine. The database name is Plan. The connectionstring name in web.config is also Plan.

```
<connectionStrings>
  <add name="Plan"
       connectionString="Data Source=(LocalDb)\v11.0;AttachDBFilename=|DataDirectory|\plan.mdf;Integrated Security=SSPI;"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

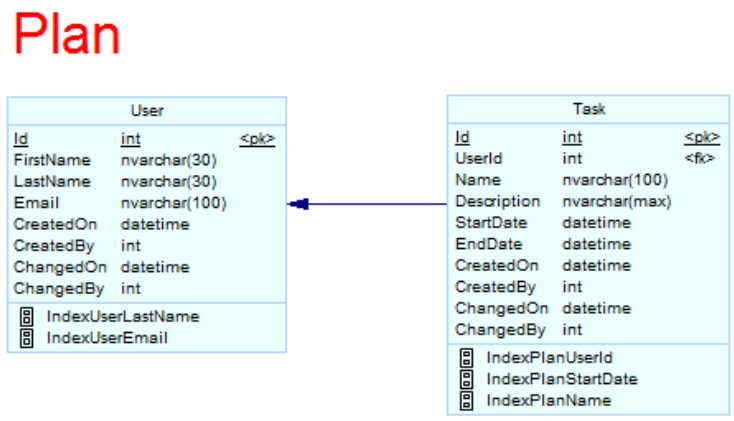
It is important that you remove the DefaultConnection entry in web.config as well as all <profile>, <membership> and <roleManager> tags.

At this time you need to have a data model that follows the simple conventions laid out earlier in this document. In summary, it should have only tables and indexes. Table names are short and singular.

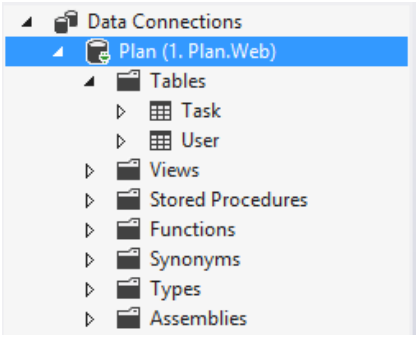
Column names are proper cased and have default values when appropriate. Each table has an Id identity field and 4 optional audit columns. Foreign keys are named 'parent table name' + 'Id', as in ProjectId, or UserId. It is preferred to use column constraints over lookup tables (to minimize JOINs).

Don't include any SimpleMembership tables as they will be added automatically.

We will use a very simple database with just 2 tables: User and Task. Below is the ERD (entity relationship diagram).



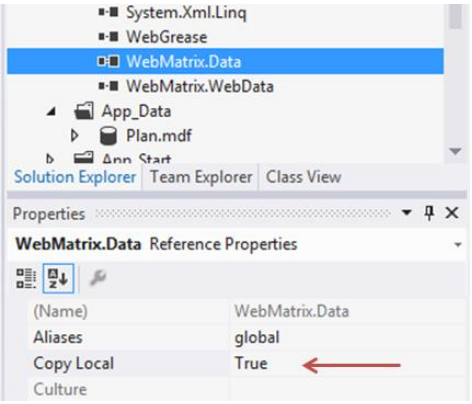
Once you have the DDL (data definition language), execute it against the database. Your database is now ready to go.



To add SimpleMembership to your solution requires two references: WebMatrix.Data and WebMatrix.WebData. Be sure to include Version 2 or better. You can either add this to web.config:

```
<compilation debug="true" targetFramework="4.5" >
  <assemblies>
    <add assembly="WebMatrix.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
    <add assembly="WebMatrix.WebData, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
  </assemblies>
</compilation>
```

Or you can reference these assemblies in your project *and* set their properties to: CopyLocal=True.



To instruct SimpleMembership to add its own tables to the database you include a few lines of code in Application_Start in global.asax. For the Plan app it will look something like this:

```
protected void Application_Start()
{
    WebSecurity.InitializeDatabaseConnection("Plan", "User", "Id", "Email", true);

    if (!Roles.RoleExists("Admin")) Roles.CreateRole("Admin");
    if (!Roles.RoleExists("Member")) Roles.CreateRole("Member");
```

The arguments in the InitializeDatabaseConnection will differ depending on your app's data model, as are the roles which may be different for your specific apps.

You're all set now with the database and SimpleMembership.

Building the Web project

First adjust some folders: remove the empty \Controllers and \Models folders. Then add \Code, \Areas, and \Images folders.

With the Art Shop reference application at hand you can follow the same structure. Of course, you will need to modify files in the \App_Start folder according to your needs. Similarly, \Areas is where your application specific Models, Views, and Controllers will reside.

Most of the folders and files in \Code can be taken verbatim from Art Shop with the exception of ArtCache, CurrentCart, and Service. In these situations follow the general pattern and implement your own application logic.

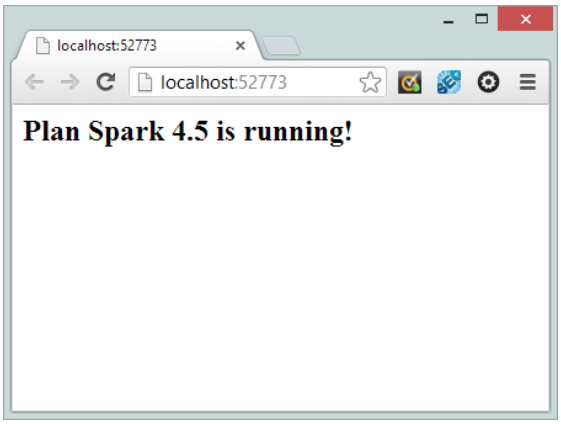
Add application specific app.css and app.js files to \Contents and \Scripts respectively.

Depending on your needs add jquery.history.js, jquery.flot.js, and jquery-migrate.js JavaScript files to \Scripts. The last one is a shim file for functionality that was removed from older versions of jQuery. In Art Shop it was necessary to get the jQuery UI controls to work.

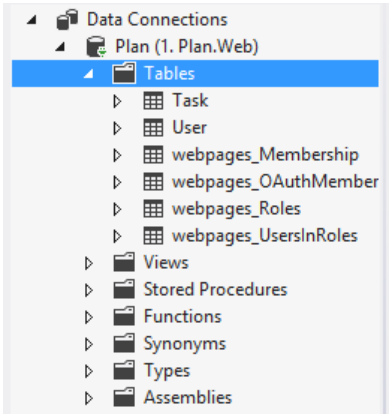
To get the first page rendered, create a Home Area with a HomeController. The Index Action method will render the Index View. Add this view and change the title (for example: "Plan Spark 4.5 is running"). Then make this the default page by adding the following route in the HomeAreaRegistration file.

```
context.MapRoute("", "", defaults: new { controller = "Home", action = "Index", area = "Home" });
```

Select run and you will see your first page:



When running for the first time, the 4 SimpleMembership tables will be added to the database.

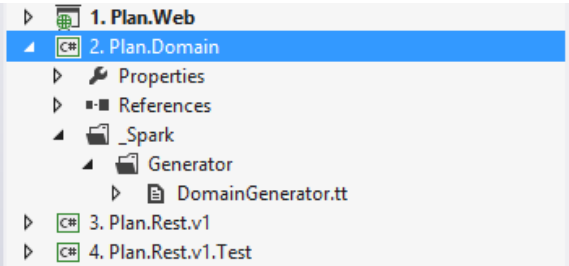


Building the Domain project

First, add two references to this project: System.Transactions and System.Configuration.

Initially we'll show you how the code generator in the PRO version does its work. After that we'll see how to create your Domain project without it.

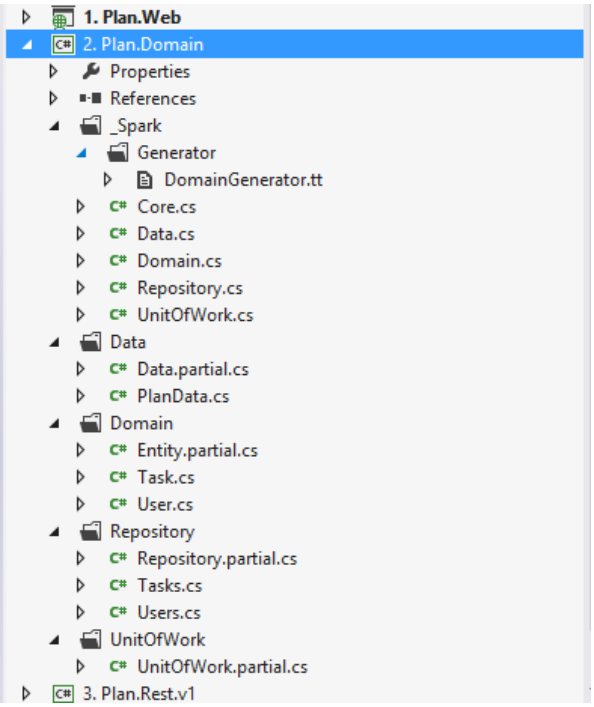
Create the _Spark\Generator folder and copy the DomainGenerator.tt T4 code generator file into this folder.



Next, change 3 custom settings inside this file: the connection string, the namespace, and the application key name, like so:

```
string connectionString = "Data Source=(LocalDb)\v11.0;AttachDbFile=etc.";
string domainNamespace = "Plan.Domain";
string appName = "Plan";
```

Save and within seconds the code generator will have created three complete layers: Repository, Domain and Data Access. Here you see the files and folders:



The Plan database has just two tables, so the number of domain and repository objects is small.

Now that the Domain project is complete you can start querying the database. There is no data in the database yet, so we'll perform a simple record count of the User table. We'll do this in the Index Action method (which renders the home page), like so:


```
public ActionResult Index()
{
    int count = PlanContext.Users.Count();
    return View();
}
```

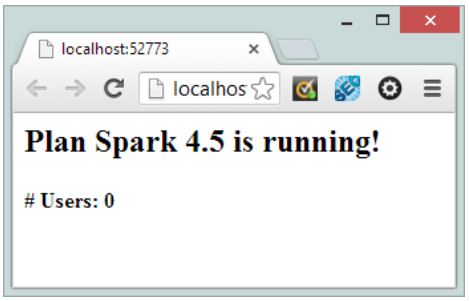
In fact, we like to shorten the data access code by aliasing PlanContext to *db* like so:

```
using db = Plan.Domain.PlanContext;
```

With this alias in place we can then write:

```
public ActionResult Index()
{
    int count = db.Users.Count();
    return View();
}
```

To see the results on the page we assign the count value to ViewBag and display the value on the main page. Here are the results.



As expected, this will return a value of zero.

You can see that in just a couple of minutes the code-generator has created three out of four layers. This allows you to focus on what makes your application unique (in Plan.Web) rather than on the more mundane task of getting data in and out the database and creating and saving domain objects.

Hand-coding the Domain project

Without a code generator you will first create four folders: \Data, \Domain, \Repository, and \UnitOfWork and then add the necessary files using the Art Shop application as a reference.

Actually, this process has already been discussed in detail in this document in the "Building Domain projects with Spark 4.5" section under the Code Review heading. In that section a Sales database example was used. Simply replace Sales with Plan and follow along.

Essentially, when building a Domain project by hand, you simply follow the same structure as in Art Shop and adjust the namespaces, some class names, and of course the Domain and Repository objects which should match the tables in the database. Most of the work will be in writing the Domain objects which should match the columns in the associated tables.

Hand-coding is entirely feasible although it will take time. Of course when the data model changes you will need to revisit and update the domain and repository objects again – but this is no different than what you normally do.

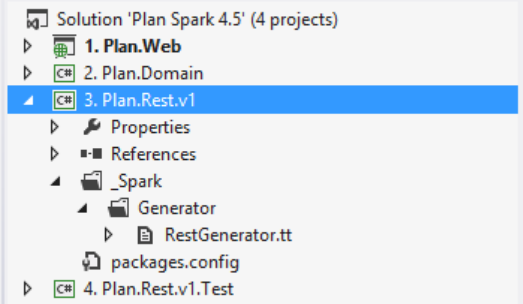
In summary, the advantages of using the PRO Spark 4.5 code-generator are 1) speed, 2) consistency, and 3) automatic code updates when data model changes occur. As usual, the PRO version of the Design Pattern Framework includes a 1-year subscription which covers future updates and enhancements.

Building the REST project

First, add a reference to System.Net.Http to this project.

Creating the REST projects follows the same patterns as the Domain project, that is, it can be created with the PRO Spark 4.5 code-generator or it can be coded by hand. First we'll see the code generator.

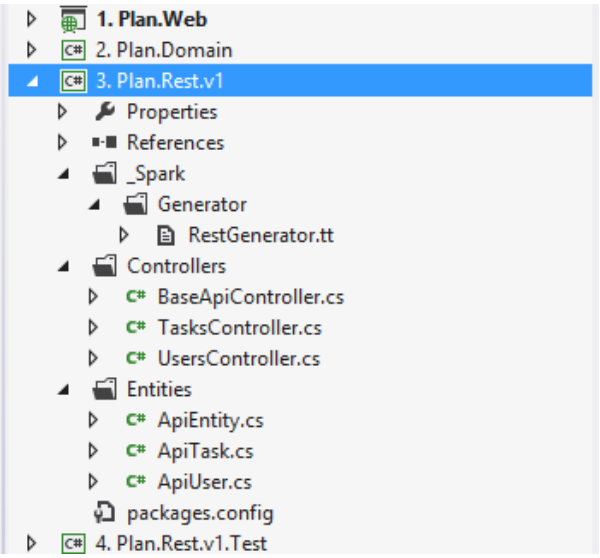
Create the _Spark\Generator folder and copy the RestGenerator.tt T4 code generator into this folder.



Next, change 4 custom settings inside this file: the connection string, two namespaces (the Domain and Rest projects), and the api root which is the root url of the REST services. Here's what this looks like:

```
string connectionString = "Data Source=(LocalDb)\v11.0;AttachDbFile=etc.";
string domainNamespace = "Plan.Domain";
string restNamespace = "Plan.Rest.v1";
string apiRoot = http://localhost:52773/api/v1/;
```

Save and within seconds the code generator will have created two folders with ApiController and ApiDomain objects. Here are the files and folders:



To be able to generate the proper domain objects (i.e. ApiObjects) the code generator needs to examine all the parent-child relationships in the database. Open the ApiUser and ApiTask classes and you'll see the result. A User can have many tasks and this is modeled as a list of Tasks inside the User.

```
public class ApiUser : ApiEntity
{
    public ApiUser()
    {
        Tasks = new List<ApiEntity>();
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public List<ApiEntity> Tasks { get; set; }
}
```

In the opposite direction each Task belongs to a single User, so the ApiTask class has a reference to a User.

```
public class ApiTask : ApiEntity
{
    public ApiEntity User { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public DateTime? StartDate { get; set; }
    public DateTime? EndDate { get; set; }
}
```

Using object references rather than foreign keys allow the REST clients to quickly ask for complete sets of data using the 'expand=' querystring variable which will greatly simplify and reduce the number of REST requests.

To support our first REST test (further down this document) we will add a little code to the UsersController in the GET method. It retrieves all users and maps these to ApiUser objects (note: db is an alias for PlanContext) which are then returned.


```
public class UsersController : BaseApiController
{

    // GET Collection

    [HttpGet]
    public IEnumerable<ApiUser> Get(string expand = "")
    {
        var users = db.Users.All();
        return Mapper.Map<IEnumerable<User>, IEnumerable<ApiUser>>(users);
    }
```

REST is gaining a lot of traction lately, but designing a REST API is more an art then a science. Other than HTTP itself there are no clear standards. When designing REST interfaces, you not only need to consider the urls, but also what data should be exposed, how should it be exposed (just retrieve or also update), and to whom.

The code generator cannot make these decisions for you and this is why it can only provide the skeleton controller classes for each table/domain object. In our Plan.Rest.v1 project two controllers are created: UsersController and TasksController. Both derive from BaseApiController. Each controller class contains scaffolding for the basic HTTP verbs: GET (single and list), POST, PUT, and DELETE. You will need to fill in these methods depending on what you choose to expose.

Plan.Rest.v1 is version 1 of the REST API for the application. If and when a second version is required you simply add a new project named Plan.Rest.v2.

Hand-coding the REST project

Without a code generator you will first create two folders: \Controllers and \Entities and then add the necessary files using the Art Shop application files as a reference.

Simply follow the same structure as in Art Shop and adjust the namespaces and of course the Controller and ApiEntity objects which should match the tables in the database. Most of the work will be the Controller objects although this depends on what you wish to expose to outside clients.

Hand-coding the basic structure is entirely feasible because it is pretty much cookie cutter code. Of course it takes time and when the data model changes you will need to revisit and update the code again.

As with the Domain project, the advantages of using the PRO Spark 4.5 code generator in the REST project are 1) speed, 2) consistency, and 3) automatic code updates for when data model changes are required.

Building the REST Test project

First, add a reference to System.Net.Http to this project.

Create a new \Entities folder and add the relevant entity objects (domain objects). With the Plan database you will need ClientUser and ClientTask. These are similar to the ApiObjects in Rent.Rest.v1,

but include a Href and the ApiEntity instances are replaced by their Client instances. Here is an example of ClientUser:

```
public class ClientUser
{
    public ClientUser()
    {
        Tasks = new List<ClientTask>();
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public List<ClientTask> Tasks { get; set; }
}
```

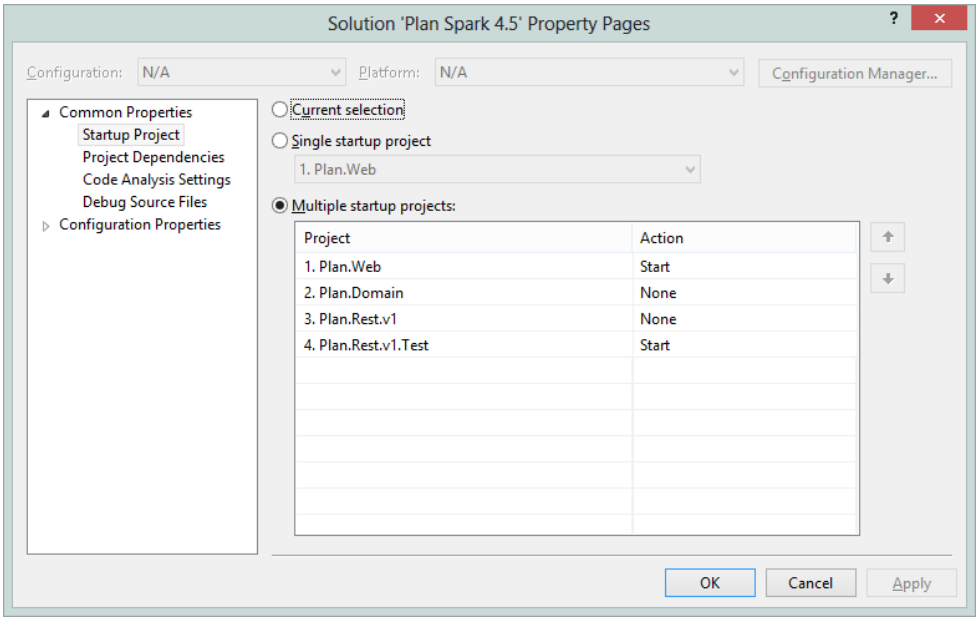
Inside the Program.cs file remember to change the endpoint to the base url of Plan.Web project, something like: http://localhost:52773/api/v1/. Of course, you need to write/adjust the code inside this file as well, depending on what you are going to test. In our case we retrieve all users and display their count.

```
// Get users with task information

var users = Get<List<ClientUser>>(endpoint + "?expand=task").Result;
Console.WriteLine("Read " + users.Count + " products with Task details");
```

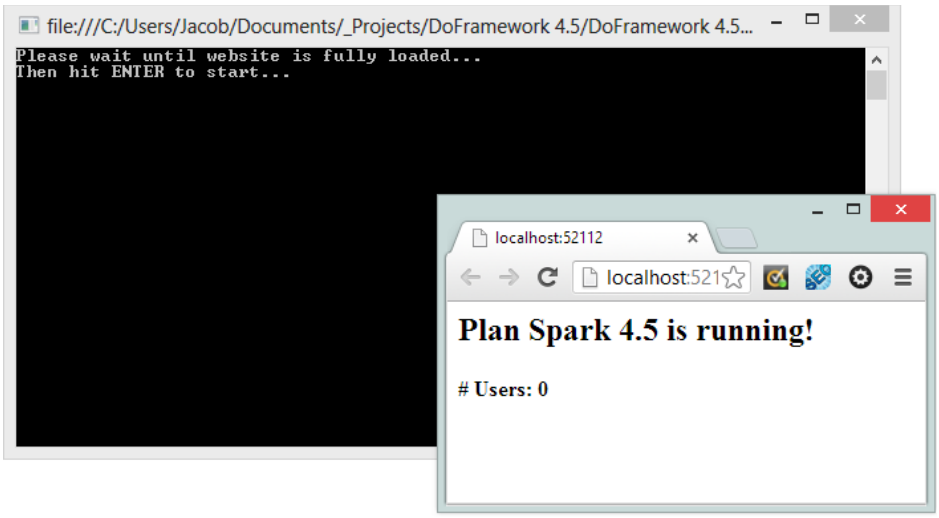
In project Plan.Web the WebApiConfig.cs file needs to define an HttpRoute before it can respond to REST requests. Here is what is needed: (note the inclusion of v1 for the first version)

```
public static void Register(HttpConfiguration config)
{
    // note the inclusion of the v1 folder

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/v1/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}
```

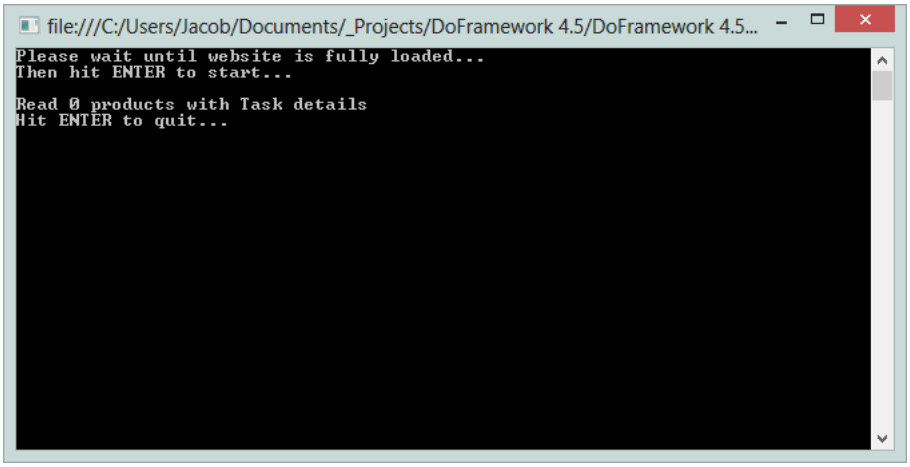
We can now run the REST test. Right click on the Solution select Properties. Inside the dialog box you can select two startup projects: Plan.Web and Plan.Rest.v1.Test.



Run and wait until both projects are fully loaded and ready to go.



Put focus on the Console window and hit enter. The first time it runs it may take a few moments but you will see the following results, which is correct because there are no users.



Summary

This completes our overview of Spark 4.5 and the Art Shop. Spark is a powerful pattern-based platform for rapid application development. The Art Shop clearly demonstrates how Spark 4.5 helps building great applications easily and quickly.

Now is the time to get started and apply the Spark techniques in your own projects. When doing this you may want to consider an upgrade to the *PRO Design Pattern Framework 4.5* which is available for purchase on our website. It includes the PRO Spark 4.5 code generator which will accelerate your projects by creating 3 of the 4 layers in your projects in a matter of seconds. Other advantages are that 1) the generated code is super consistent (requiring little or no testing), and 2) data model changes are quickly and easily incorporated into your code simply by rerunning the generator.

The cost of the PRO Design Pattern Framework 4.5 is less than having an experienced .NET consultant work for a day. Clearly, the Spark code-generator will save you much more than a day's worth of work. Just consider the time savings of having 3 out of 4 layers automatically created; not to mention the benefits of having data model changes automatically incorporated into your code. Don't be surprised to see you developer productivity go up by a factor 4! If you dread working under continuous deadlines then the PRO version is a very compelling product. Give it a try!

If you have any feedback or questions please feel free to contact us via the *contact us* page on our website or simply an email to support@dofactory.com.

Good luck with your design pattern and architecture endeavors.