Running Head: SIMPLESTORE ARCHITECTURE

SimpleStore Web Architecture Research

Patrick Brossard

Regis University

Abstract

There are a vast number of architectures for building web sites. This paper identifies some of the best practices for use in designing small, data-driven websites, focusing on the following design goals: 1. Separation of concerns, 2. Complete unit test coverage, and 3. Convention over configuration. Comparisons were made among leading reference architectures to extract the design elements which would best support these goals, while maintaining a relatively low level of complexity. The final recommendations propose a web architecture based on the Model-View-Controller pattern to provide separation of concerns, the use of dependency injection and mocking to support robust unit test development, and incorporating an Object-Relational Mapping (ORM) package to allow for a convention-based data access layer.

SimpleStore Web Architecture Research

Background

Web development is a crucible for many software development techniques and methodologies. Technology changes faster, deadlines are tighter, and the social and open nature of the Internet makes competitive advantage or disadvantage quickly obvious to users. Thankfully, web development also tends to be less mission critical than traditional software development. Though an Internet juggernaut such as Amazon or eBay may perhaps disagree, it is much more essential for flight navigation monitors or city traffic control systems to functional faultlessly than for consumers to be able to buy online.

Thus we see that web development, by its nature and medium, has certain biases: towards speed, impermanence, and a love of change over stability. In order to be successful in developing for the Internet, these biases must also inform our development practices. So-called “agile” methodologies, which have been rapidly gaining acceptance in the marketplace, are embracing these biases by abandoning the traditional linear software development lifecycle: long analysis, design, implementation, test and release phases – often taking years to complete. Already a significant source of project failure in traditional software projects, these long project cycles are anathema to web development. In place of this monolithic process, agile methods instead *expect* change and minimize its impact on the development cycle. Development cycles are kept short, design is focused on the problems at hand, and a constant stream of project iterations and prototyping keeps putting something tangible in front of the customers or decision makers, allowing them to grow their vision with the project instead of having to completely document their needs up front. (Fowler, 2005)

On the heels of these changes in the software development lifecycle have come changes in how these smaller, tighter iterations are designed. The greater focus on change and constant iterations has required that the code being written be much more robust and loosely coupled. Instead of writing a function once and not needing to revisit it for two years, programmers working on agile products find themselves reworking and “refactoring” code on a continual basis. If the code is poorly written, or the architecture poorly designed, this continual code churn becomes a negative, not a positive. Instead of iterating toward better functionality and a closer match to the customer’s goals, teams can find themselves introducing bugs, spending more time integrating, and having to re-architect other systems affected by changes. (Agile, 2010)

For this reason, a greater focus on design and architecture has emerged. This re-examination has resulted in many new methods and practices; so many in fact, that development teams must judiciously decide which they are going to attempt to apply. Even if each pattern or practice offers some good outcome, attempting to incorporate them all will result in a train wreck of complexity and overhead. For this reason, I have kept my scope deliberately small, and selected but three primary concerns to shoot for in defining a web architecture. Read on to see how they were arrived at…

Approach / Methodology

There is no shortage of guidance available on the Internet when looking at web architectures and development. To try to compare and contrast all that is available would be foolish – we need to provide constraints to limit the pool of information. By virtue of where the SimpleStore project had progressed by the time significant research into the architecture began, there were already some constraints:

* We are developing a small, data-driven website.
* We are using Microsoft technologies.

In addition to these constraints, I needed to place some design goals on the architecture. As you will recall, the biases of web development are to favor rapid change and time-to-market over other considerations such as performance or strict engineering control. With that in mind, I selected design goals which I felt would contribute to being able to accommodate rapid development and changing requirements.

**Design Goal #1: Single Responsibility Principle**

This principle is similar in goal to “Separation of Concerns” or “loose coupling/high cohesion”, but with a much easier rule of thumb to determine when and where to apply it. The principle states that a class should only have one reason to change. It has a single responsibility, and if that responsibility is not changing, you should have no reason to change the class. The classic example of this is database awareness in a business logic class. You now have two reasons to change the class – a change to the business logic, or a change to the data tier. That class should have a single responsibility, the business logic, and the data persistence functionality should be implemented elsewhere. (Hayden, 2005)

At a high level, Microsoft has two major web architectures: ASP.Net Web Forms, and ASP.Net MVC. With the single responsibility principle as my number one concern, I have pretty much barred Web Forms from consideration. In the Web Forms architecture, the page classes have more than one reason to change: changes to the UI, or changes to the form functionality. With the MVC architecture, these responsibilities are divided – the View takes care of all the UI considerations, and the Controller manages the functionality. While it is *possible* to create these delineations using Web Forms, the separation is enforced natively in MVC, making it a much better choice.

**Design Goal #2: Complete Unit Test Coverage**

Unit testing has had a well deserved resurgence under agile methods. In fact, a particular school of development as emerged, Test-Driven Development (TDD) which advocates that a unit test should be written *before* any development is undertaken. This not only implicitly requires any new functionality to have unit test, but also forces the developer to clearly state in code the goal of his work beforehand. (Ambler, 2010)

While I have not embraced TDD, I do see the value in a comprehensive suite of unit tests. Referring back to our initial biases towards speed and change, having a complete set of test cases gives an incredible amount of confidence to undertake large changes. You can refactor and make some wide-reaching changes, see your unit tests pass, and have a good degree of confidence that your application is still functional. Especially on smaller teams, or teams without dedicated testing staff, a good suite of unit tests is critical in catching bugs and keeping code quality up.

**Design Goal #3: Convention over Configuration**

Every developer dreads writing “plumbing code”, the rote cut/paste/modify chunks that everyone already knows and understands and just needs to be propagated all over the application to get something done. Database access code often used to fall into this category. Well, it turns out there’s a better way. Instead of explicitly telling the program what to do in each instance, we can make the program make educated guesses about what needs to be done, and only explicitly code the exceptions. Most MVC frameworks such as Ruby on Rails and ASP.Net MVC make extensive use of convention – there’s no need to explicitly associate a controller with its views, as they are associated by naming *convention*. (Miller, 2009)

One relatively new development in web architectures is the advent of Object Relation Mapping (ORM) software. ORM packages act as an encapsulated database access layer, translating between objects and database records. Many ORM packages are convention based, and by following the conventions you may never need to write a single line of database housekeeping code.

Results

Armed with my three design goals, I began to look at the reference architectures available for Microsoft’s ASP.Net MVC. I found three, two from Microsoft itself, and one from a group of third-party developers. No surprise as to which one proved most valuable…

**MVC Music Store (** [**http://www.asp.net/mvc/samples/mvc-music-store**](http://www.asp.net/mvc/samples/mvc-music-store)**)**

I believe the MVC Music Store is meant primarily as an educational tool. It displays the basics of MVC in an easy to follow manner, but as an architecture, it leaves much to be desired. No unit testing at all, rudimentary LINQ-to-Sql data access, limited error handling, and other deficiencies.

**Nerd Dinner (** [**http://nerddinner.codeplex.com/**](http://nerddinner.codeplex.com/) **)**

The Nerd Dinner application has some interesting design elements. It makes use of Ajax and highlights some of the interesting features and frills Microsoft has built into the ASP.Net MVC platform. It makes some use of unit testing, and tries to show how unit tests could be built to cover the controllers. However, it uses hand coded “fakes” as opposed to an actual Mocking framework, something which definitely contradicts the Convention over configuration mandate. I ended up using the Nerd Dinner application primarily as a syntactical tool to introduce myself to the new features of ASP.Net MVC 2.0.

**Sharp Architecture (**[**http://sharparchitecture.net/**](http://sharparchitecture.net/)**)**

This architecture astounded me. I think I learned more about MVC and web architecture design by thoroughly reviewing and understanding their sample application than I have in the last two years at my day job. The Sharp team takes a firm TDD approach, which I don’t necessarily embrace; but the fact that they believe in supporting unit testing so much means there is excellent support for testing at all tiers (well, except for the Views, though they do touch on that), including mocking and dependency injection. The architecture employs the NHibernate ORM, which I have used before, but manages to use it in a novel way, and combines it with a few custom libraries to bring its convention-based design to new heights. The architecture also specifies very strict lines of delineation between the different projects which make up the solution, creating a great single-responsibility starting point.

Despite being the clear winner among the architectures I was review, the Sharp Architecture is not perfect. In fact, it does too much. It is a full strength, best-possible, enterprise scale architecture; and overkill for my needs. An excellent starting point, but I have trimmed it down to just the features which I feel I really need… an architecture I have dubbed Sharp Architecture Lite.

Discussion

So how do the facilities provided by the Sharp Architecture actually play out in real code? What do “mocking”, or “dependency injection”, or “separation by interface” mean for an application built on the Sharp Architecture Lite? I will review each of my design goals, and show exactly how the architecture helps to realize that goal.

**Single Responsibility Rule**

Aside from the separation of concerns already provided natively by the MVC model, the Sharp Architecture also builds in separation between the different logical tiers of the application. Let us examine the situation brought up earlier, a violation of the single responsibility principle by having database access awareness built into the business logic components. The Sharp Architecture addresses this situation through a technique called a “Separated Interface”. A separated interface *defines* the functionality a component will provide in one class, but actually *implements* that functionality elsewhere. (Fowler, 2002)

As for how this works in practice, examine the following code snippet. A private member is set up which hold a sellerRepository interface. When the SellerController is instantiated, an object actually implementing the interface is passed in and assigned to that private variable. But this class can code against the sellerRepository interface as if the implementation were there all along, as it does in the SaveOrUpdate method used later in Create.

private readonly IRepository<Seller> sellerRepository;

public SellerController(IRepository<Seller> sellerRepository)

{

this.sellerRepository = sellerRepository;

}

public ActionResult Create(Seller newSeller)

{

try

{

newSeller.Created = DateTime.Now;

sellerRepository.SaveOrUpdate(newSeller);

return RedirectToAction("Index");

}

…

**Unit Test Coverage**

The above example leads naturally into a discussion on the many things the Sharp Architecture does to make unit testing easier, and to expand the coverage possible in your tests. As we saw in the above example, this SellerController accepts in a class implementing the sellerRepository interface at the time it is instantiated. This means that we can selectively decide which implementation we are going to pass in depending on how we are using the SellerController. For example, under normal use the SellerController would be getting instantiated as part of our website, and we would be passing in a sellerRepository implementation which would retrieve and store data to the database. BUT, if we decide to run some tests on the SellerController class, we can alternatively pass in a different sellerRepository implementation, which in this case does not hit a database; databases introduce a high level of lag, statefulness, and general fragility into your tests. This is known as Dependency Injection – the SellerController is depending on the sellerRepository, but instead of that dependency being abstracted and hidden from the caller, the caller is given the option to supply and “inject” that dependency. Here is the concept in action in a sample test case:

SiteController controller = new SiteController(CreateMockSiteRepository());

ViewResult result =

controller.MatchingFilter("Does not matter")

.AssertViewRendered()

.ForView("MatchingFilter");

Assert.IsNotNull(result.ViewData);

Assert.IsNotNull(result.ViewData.Model as List<Site>);

Assert.AreEqual((result.ViewData.Model as List<Site>).Count, 4);

This test cases also highlights another extremely useful facility – mocking. Since in the test above we do not want to hit the database, we must provide our own repository implementation. The call to CreateMockSiteRepository() creates a “mock” repository which has been stubbed out to offer just some basic methods. Here is the code:

public ISiteRepository CreateMockSiteRepository()

{

MockRepository mocks = new MockRepository();

ISiteRepository mockedRepository = mocks.StrictMock<ISiteRepository>();

Expect.Call(mockedRepository.FindAllMatching(null))

.IgnoreArguments()

.Return(CreateSites());

mocks.Replay(mockedRepository);

return mockedRepository;

}

private List<Site> CreateSites()

{

List<Site> Sites = new List<Site>();

Sites.Add(new Site(1));

Sites.Add(new Site(2));

Sites.Add(new Site(3));

Sites.Add(new Site(4));

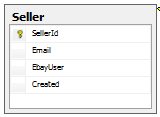
return Sites;

}

Without getting into too much detail, the CreateMockSiteRepository method returns a mock repository which expects to field a FindAllMatching method call. When it does, it returns the four Sites supplied by the CreateSites method. Mocking provides a quick and easy way to implement stubbed out implementations which can be injected into the main classes you wish to test. This strips out the latency, fragility, and constant test tweaking required when testing against a real database.

**Convention over Configuration**

Though MVC is chock full of the uses of convention over configuration, there is one example in the Sharp Architecture that surprised me by how much it made coding easier, and that was its use of the ORM NHibernate. When I went to wire up the database access, I had a Seller class that I wanted to represent the Seller entity, which was backed by the Seller table in my database as follows:



public class Seller : Entity

{

[DomainSignature]

public virtual int SellerId { get; protected set; }

public virtual string Email { get; set; }

public virtual string EbayUser { get; set; }

public virtual DateTime Created { get; set; }

public Seller() { } // NH Requirement

public Seller(int sellerId)

{

SellerId = sellerId;

}

}

Ordinarily, this would have required a whole set of Insert/Update/Select procedures in the database, as well as more than a few lines of plumbing code to wire together the class with the Table. But with the “Fluent” mapping interface in NHibernate, it required only this mapping class:

public class SellerMap : ClassMap<Seller>

{

public SellerMap()

{

Id(x => x.SellerId);

Map(x => x.Email);

Map(x => x.EbayUser);

Map(x => x.Created);

}

}

This ClassMap specifies that for the Seller entity, the SellerId member is its primary key, and the Email, EbayUser, and Created fields all get mapped to their Table counterparts of the same name. Not on the fields, but eve the class to table mapping is based on a naming convention. By convention, if there is an Entity-based class with a name of Seller, it implicitly connects that to the table with a name of Seller. Instead of having to wire everything up by hand, the developer needs only define the exceptions. This represents significant time savings, and reduced code bloat.(McCafferty, 2008)

Conclusion

In conclusion, I feel that the MVC model combined with the organization, testing enhancements and ORM integration provided by the Sharp Architecture offers an excellent starting point for virtually any web application. Given the small scope of the SimpleStore site, I chose to pare down the complete offering to focus more closely on my three design goals: single responsibility, unit test coverage, convention over configuration. These design goals were met, in fact I can honestly say that the level at which the architecture supports those three goals exceeds my initial expectations. The architecture is solid. The obvious next steps for this project are to begin implementation!

In personal reflection, I can say with complete certainty that I have learned more, and certainly been more interested in this class and project than any other during by work at Regis. This partly has to do with my life situation – I have been a software developer for ten years, so intro to Java courses do not hold much appeal for me. I really appreciate the latitude that was given for students to pursue their own interests in this Capstone. It takes the project beyond “graduation requirement” and into something we can become personally invested in. Thanks!

References

Ambler, Scott W. (2003) Introduction to Test Driven Design <http://www.agiledata.org/essays/tdd.html>

Fowler, Martin. (2002). Separated Interface

<http://www.martinfowler.com/eaaCatalog/separatedInterface.html>

Fowler, Martin. (2005). The New Methodology

<http://martinfowler.com/articles/newMethodology.html>

Hayden, David. (2005). Single-Responsibility Principle

<http://davidhayden.com/blog/dave/archive/2005/05/29/1066.aspx>

McCafferty, Bill. (2006). Dependency Injection for Loose Coupling  
<http://www.codeproject.com/KB/architecture/DependencyInjection.aspx>

McCafferty, Bill. (2008). NHibernate Best Practices with ASP.NET

<http://www.codeproject.com/KB/architecture/NHibernateBestPractices.aspx>

McCafferty, Bill. (2008). Refactoring Service Dependencies to Separated Interface  
<http://devlicio.us/blogs/billy_mccafferty/archive/2008/10/30/refactoring-service-dependencies-to-separated-interface.aspx>

Miller, Jeremy. (2009). Patterns in Practice: Convention Over Configuration  
<http://msdn.microsoft.com/en-us/magazine/dd419655.aspx>

Wikipedia, retrieved 6/21/2010. Agile Software Development  
<http://en.wikipedia.org/wiki/Agile_software_development>

Appendix A

Capstone Product

The Capstone Product is submitted as a separate document.

Appendix B

Capstone Product Presentation

The Capstone Product Presentation is submitted as a separate document.