|  |
| --- |
| Open Source |
| Enterprise level web-app Architecture and Design |
| A simple web application to demonstrate the architecture and implementation, commensurate to an enterprise project. |

|  |
| --- |
| Vaibhav Singh (.Net Architect, New Delhi, India)  4/9/2015 |

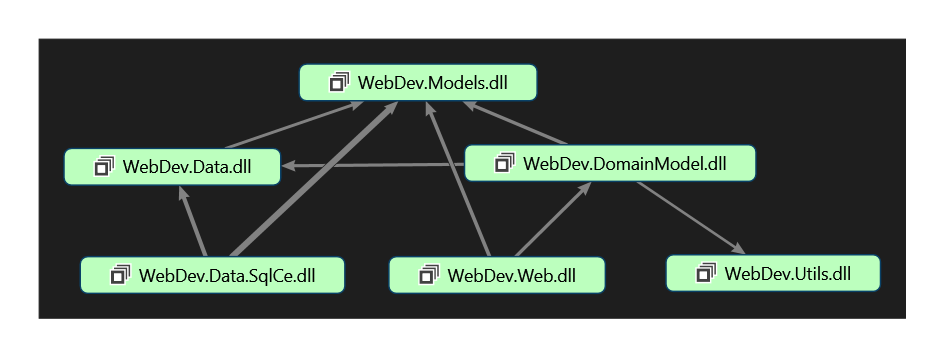
Contents

[Introduction 2](#_Toc416384141)

[Pre-requisites 2](#_Toc416384142)

[HLD – High Level Diagram: System Architecture 2](#_Toc416384143)

[Layers 3](#_Toc416384144)

[ 3](#_Toc416384145)

[Data Tier 3](#_Toc416384146)

[WebDev.Data 3](#_Toc416384147)

[WebDev.Data.SqlCe 4](#_Toc416384148)

[The basic *Context* accesses is encapsulated in the *BaseRepository*. 5](#_Toc416384149)

[Model Layer 5](#_Toc416384150)

[Domain Layer 6](#_Toc416384151)

[WebDev.DomainModel 6](#_Toc416384152)

[Library 7](#_Toc416384153)

[WebDev.Util 7](#_Toc416384154)

[The Web App Layer 7](#_Toc416384155)

[WebDev.Web 7](#_Toc416384156)

[Client-Server Optimizations 9](#_Toc416384157)

[Asynchronous XMLHttp 9](#_Toc416384158)

[The UI 10](#_Toc416384159)

[Link to the source code 10](#_Toc416384160)

# Introduction

The WebDev web application is a MVC4 based project developed using the latest designs, frameworks, libraries and tools. The application uses Entity Framework Code First with POCO and dynamic seeders.

The application exploits Microsoft’s Unity container for Inversion of Control and implements other proven design patterns like:

* Dependency Injection
* Repository Abstraction
* Strategy
* Service Locator
* SOLID principles

This document enunciates the projects and their intrinsic components in greater detail. It also serves as a technical reference, knowledge transfer document, HLD and presentation.

# Pre-requisites

The application is based on and needs the following technologies on the deployment server/system:

1. Microsoft .net 4.0 framework
2. ASP.net MVC4
3. SQL CE 4.0
4. Microsoft IIS 7.5+ or equivalent webserver
5. Any latest (javascript enabled) browser.

# HLD – High Level Diagram: System Architecture

Presentation Layer

Domain Model

Libraries

Data Tier

As depicted in the layer diagram, the system comprises of three discrete tiers stacked seamlessly in a loosely coupled fashion to facilitate scalability and flexibility of code.

The layers with the comprehensive class diagrams with the description of each will follow ahead in the document.

# Layers

## 

Here is a schematic of the projects involved in the corresponding layers as outlined by the abridged HLD above.

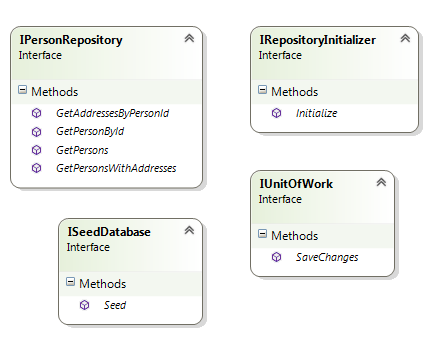
Following is a comprehensive description of each project per the layers.

## Data Tier

The Data Tier implements the Repository Pattern making it easier for the Domain Model (explained later) to be loosely coupled with the database implementation. The Domain layer works closely with the contracts defined by the Repository Pattern. This coherent layer of indirection makes the Domain Model agnostic of the databse implementations. This is congrous to the SOLID principles.

### WebDev.Data

Here is an exploded view of the project itself:



This is where the Repository Pattern is implemented.  
The base directory defines interfaces for the ‘Unit of work’ pattern, a Repository Initializer to initialize the database and a seeder. More details are mentioned in the commets against the usage of each in the source code.

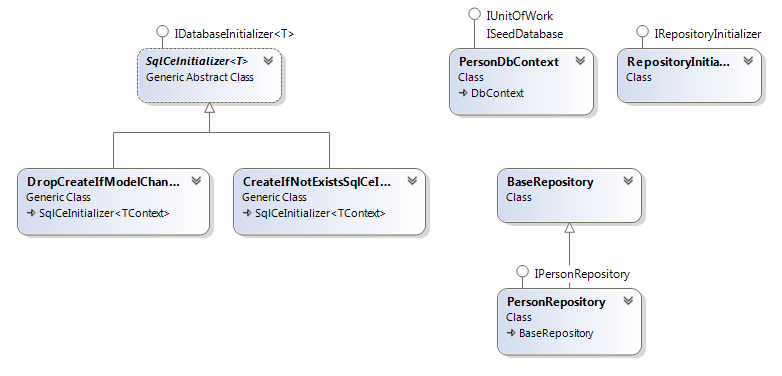
The IPersonRepository interface defines a basic contract for the Person entity.

### WebDev.Data.SqlCe

This is an implementation of the Repository layer abstraction. It can be noticed that the application is agnostic (loosely coupled) of this project. The database interaction logic can be later changed to any desired driver/provider and the application will not have to worry about the changes thereof.

This makes the application extremenly scalable, robust and flexible. This exhibits the SOLID principles.

Here is a pseudo UML of this project:



The **PersonDbContext** overrides the events from *DbContext* and uses *FluentAPI (EF)* to construct the database model – code first.

It is also responsible for *Seeding* the database if the DB is deleted or its schema modified.

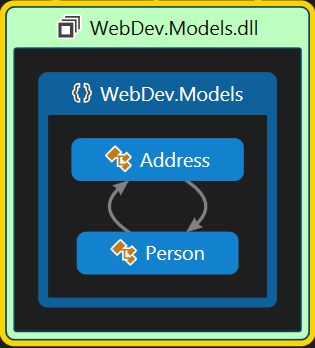
The **SqlCeInitializer** is the *Strategy Pattern* implementation using multiple *algorithms* or methods for constructing the database as depicted in the UML above.

### The basic *Context* accesses is encapsulated in the *BaseRepository*.

## Model Layer

This layer defines the POCO entities that the application will be using. The *Entity Framework*’s Code-First approach makes use of these models to populate the Database schema and relationships as defined by using either of the *Data Annotations* or *Fluent API.*

Here is a schematic outline of the project:

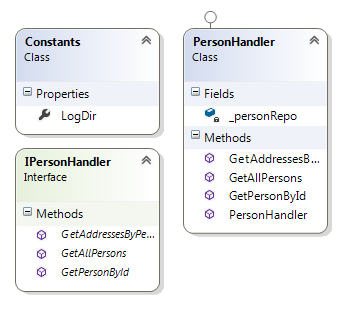


## Domain Layer

This is canonically known as the BAL or Business Access Layer. The project here comprises of abstract contracts (interfaces) and their respective implementations. The *Presentation Layer* is loosely coupled with the Business Layer in that, it makes use of only the contracts defined by the *Domain Model* to interact with the *Domain Layer.*

### WebDev.DomainModel

Here is a basic UML of the project.



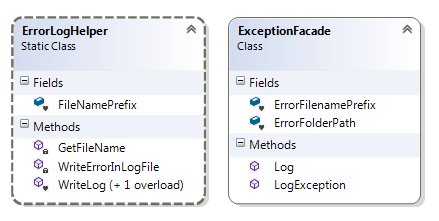
As depicted, it provides the essential methods to deal with the *Person Handler.* This is synonymous to the *Domain Driven Design.*

## Library

The Library consists of basic utilities used ubiquitously. Currenly, it hosts the Exception Handlers implemented using *Façade.*

Here is an abridged UML of the project:

### WebDev.Util



Façade provides a simplified encapsulation of the underlying Exception logging Helpers.

## The Web App Layer

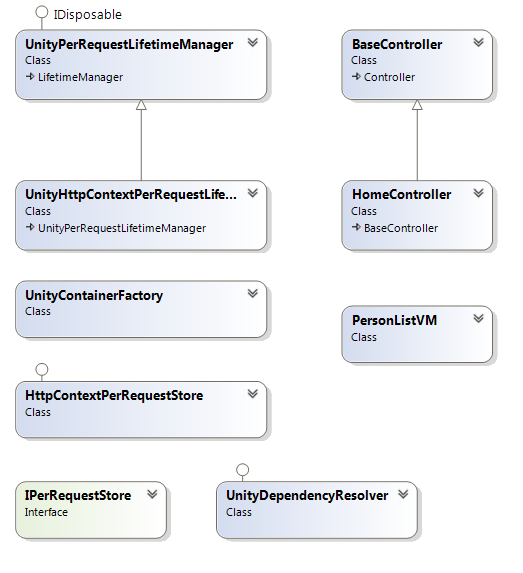
Last but not the least is the presentation layer itself that consumes all of the above architecture mentioned thus far.

### WebDev.Web

The presentation layer is ASP.net MVC4 project that makes use of the following patterns:

1. Dependency Injection:
   1. For this, it uses Microsoft’s *mature* IOC container called **Unity.**
   2. The **IDependencyResolver** is implemented by Unity and registered in *Global.asax.*
   3. The Unity registry is implemented as a config file so the application would have to be recompiled everytime a new dependency implementation is added (or modified) to the application.
   4. Unity also makes use of the *Service Locator* pattern available out-of-the-box with MVC4.
2. MVVM
   1. The project defines strongly typed **Views** and the backing **ViewModels.**
   2. The controllers play the role of negotiating between the Domain Layer and the View while constructing and supplying the *ViewModels.*
3. Factory
   1. The project implements the Unity Container Factory class that is responsible for returning an object of type IunityContainer.

Here is a schematic:



# Client-Server Optimizations

In order to optimize and best utilize the network bandwidth and server resources, the application uses several *strategies* to **cache** same data across multiple client requests.

The application uses [OutputCache] Action Filter attribute cache a *Person’s* details and varies is only by the action parameter. It must however be noted that the *SqlCacheDependency* may be configured if the cache needs to be invalidated in the event of a modification to the data in the SQL Server Database.

Here is the msdn article to perform the same:

*“Use SqlDependency indirectly by leveraging the ASP.NET SqlCacheDependency object. The SqlCacheDependency object uses a SqlDependency to listen for notifications and correctly update the cache.”*

<https://msdn.microsoft.com/en-us/library/9dz445ks(v=vs.110).aspx>

## Asynchronous XMLHttp

The client uses the **Web2.0**’s async **ajax** calls.

There are two ways demonstrated in the application of the usage of ajax as:

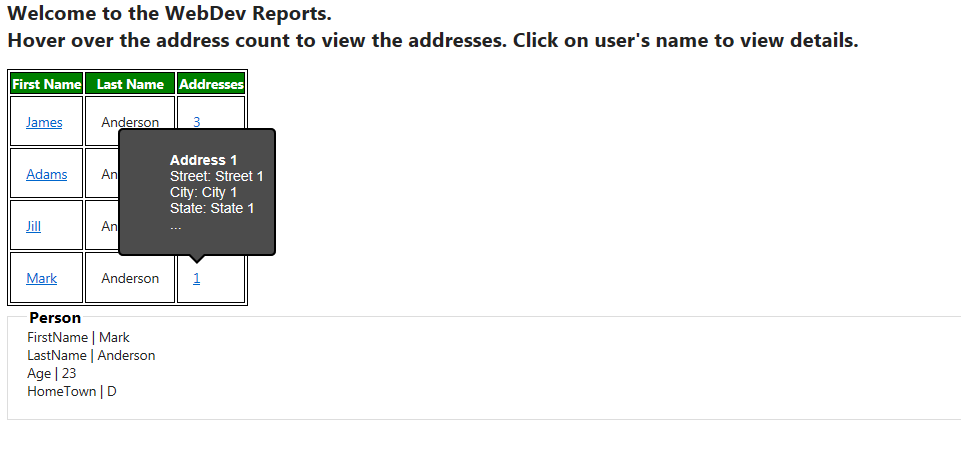
1. Using the MVC4 *Ajax* helper.
   1. The application makes use of *partial views* and MVC4’s Ajax helper that implicitly generates the required client side ajax plumbing before sending the HttpResponse to the client while processing the ViewEngine in the MVC4 request pipeline.
   2. The *Person’s Details* are fetched using this **REST** async ajax call using the *HTTP GET request* and a section on the same page is updated without refreshing the page.
2. Using *Jquery Ajax.*
   1. The application, as instructed in the task’s requirements to display address information when hovering the mouse pointer over the address count, makes Jquery **REST** ajax calls (POST) to the server, subsequently caching the response.
   2. Hence, if the user hovers the mouse over again on the same address count link, the browser returns the *tooltip* from the cache instead of making another call to the server.

PS: the above caching optimizations assume that the SQL Server data is static. As metioned above, in case the database is updated, other mechanisms like *SqlCacheDependency* may be considered to invalidate the cache.

Nevertheless, these caching optimizations increase the app’s response time prodigiously.

Please refer to the source code and the accompanying comments to get a deeper insight of the application and its architecture.

# The UI



## Link to the source code

The source code (zipped file) can be downloaded from the following dropbox location.

<https://www.dropbox.com/s/mk05s5cls5ia2f8/WebDev.Web.rar?dl=0>

(IDE: Visual Studio 2012)

Errata: Although, every effort has been made to keep the description presented in this document as accurate as possible, please feel free to report any error in the content.