**Part 2 - Design**

We'd like the Shipping App to use the Unit of Work pattern. For the purposes of this assignment, the Unit of Work pattern means:

*The framework takes care of persisting all data at the end of a defined "unit of work".*

In concrete terms, as the Shipping App is a web app, a "unit of work" will be a web request. This means that the application code should not be explicitly saving the basket after modification, but instead the framework should somehow know that it has changed and save it at the end of the web request.

Please produce a document explaining how you might do this, including any restrictions your solution has or considerations as the app continued to be developed. Your document can include diagrams, pseudo-code or just consist of words.

Note that the important thing is your ideas and thinking and how you get them across, not the presentation / formatting of the document, so please don't worry unduly about this.

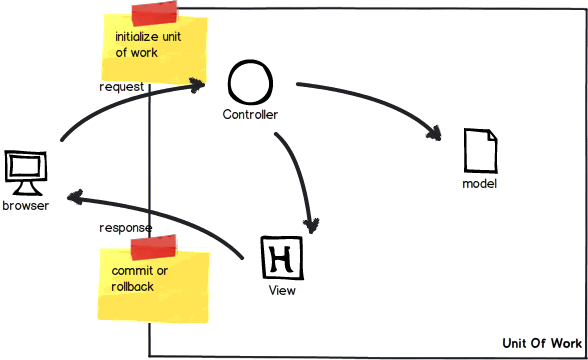
Any questions about either part of this assignment, please don't hesitate to ask. Good luck!



There are different way to implement Unit of Work pattern (UoW), in a web application.

When we want to implement it in a web application we must consider the web application life cycle, with additional focus on life cycle of Request / Response.

The developer in this way has just to worry to execute the statements, all its work will be under a unique transactionscope that automatically will be committed or roll backed at the end of the request.



The first approach is using the event BeginRequest / EndRequest native in every Web application.

It can be done in global.asax

protected void Application\_BeginRequest(object sender, EventArgs e)

or using httpModule.

We are going to open a connection + transaction or dbcontext during the BeginRequest and commit or rollback (if application raise an error) during the even EndRequest.

Everything happen in terms of SQL operation during the BeginRequest – EndRequest will be under the same transaction scope, and will be committed or rollback at the end of the request, before the server return the response to the client.

This approach present many cons:

This approach is making unit test complicated

It always open a connection for each incoming request even if the application doesn’t need to access to database.

It brakes the rule “open the connection later as possible and close as soon as possible” (even if it is possible to use the pattern lazy loading Lazy<T>).

The second approach (that I personally prefer) is using ActionFilterAttributes

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited = true, AllowMultiple = false)]

public class UnitOfWorkAttribute : ActionFilterAttribut

This technique allows to decorate the controller or better the single controller action with a custom attribute (e.g. [UnitOfWork].

Behind the scenes it happens that when the action with the attribute receives a request the filter on the event:

override void OnActionExecuting(ActionExecutingContext filterContext)

a transaction / connection will be opened and when the controller action is over an event:

override void OnActionExecuted(ActionExecutedContext filterContext)

will execute a commit / rollback base on the controller action result (OK or Errror).

Very important:

We must remember that we want UnitOfWork working for WebRequest, so it is important that we remember to configure correctly the lifestyle of the IUnitOfWork interface in our container being sure that is set to: .LifestylePerWebRequest()

The second approach allow to be more selective where set our Unit Of Work don’t opening indiscriminately for every request/response.

This approach suit very well even with the pattern CQRS.

Important to say that often UoW comes together with another design pattern Repository, and ORM like EF or Nhibernate..

There are a lot of discussion behind using ORM or not-ORM approach for our DAL.

I personally prefer the approach using standard ADO.Net or some “light-Orm” like dapper, for obvious reasons of performances, especially when are involved an high number of query / second on database.

Another elegant way is using Attributes and Interceptor by castle of Windsor container that allow to have a clean design of what happen in UoW.

Below an example using NHibernate Orm

public void Intercept(IInvocation invocation)

{

*//If there is a running transaction, just run the method*

if (NhUnitOfWork.Current != null || !RequiresDbConnection(invocation.MethodInvocationTarget))

{

invocation.Proceed();

return;

}

try

{

NhUnitOfWork.Current = new NhUnitOfWork(\_sessionFactory);

NhUnitOfWork.Current.BeginTransaction();

try

{

invocation.Proceed();

NhUnitOfWork.Current.Commit();

}

catch

{

try

{

NhUnitOfWork.Current.Rollback();

}

catch

{

}

throw;

}

}

finally

{

NhUnitOfWork.Current = null;

}

}

Some additional consideration.

Personally I do not think is a good idea implement UoW in a web application.

For my personal experience, Web application doesn’t know anything about persistence / connections / transaction, just take care about presentation logic, invoking an external service that provide Get/Set.

I would implement all the business logic and data access logic outside the web application for allowing a better scalability for the future application that will be based on the same business logic.