**Transforming the Application**



[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

Contents

[Introduction 3](#_Toc265172582)

[Web Application 3](#_Toc265172583)

[Using a Relational Database 4](#_Toc265172584)

[Web Services 4](#_Toc265172585)

[Service Layer Pattern 4](#_Toc265172586)

[Proxy Pattern 6](#_Toc265172587)

[Summary 6](#_Toc265172588)

# Introduction

Now we will demonstrate the power of partitioning and layering as we transform the application. This unit will have three steps. First we change the controller and view in the user interface to a web application. Then we will start using SQL Server by simply substituting a new data mapper class. Finally, we will introduce web services by using a service pattern for the business logic, and a proxy pattern in the model of the user interface.

This can be done very easily because of the way we have partitioned and layered our application. This document outlines the highlights of the transformation. The Code Cast for this unit has the details.

# Web Application

Since we have real events in a web app, the ICustomerView interface can be reduced to its basic form:

public interface ICustomerView

{

void SetController(CustomerController controller);

void WriteCustomer(IList<ModelCustomer> customers);

}

The web page itself becomes the view:

public partial class \_Default : System.Web.UI.Page, ICustomerView

The controller itself is created in the PageLoad:

protected void Page\_Load(object sender, EventArgs e)

{

CustomerController controller = new CustomerController(this);

SetController(controller);

}

The web page delegates the appropriate actions to the controller. The controller has a similar set of methods to handle the user interface events:

AddNewCustomer

RetrieveById

RetrieveByName

RetrieveAllCustomers

AddOrderForCustomer

As before, it uses the view to update the user interface, and the model to work with the business logic. The model class is unchanged. The controller is only slightly different because we now have real events. The view, of course, is completely changed because we now have a web application. It goes without saying that the rest of the application is totally unchanged.

Because we had organized our user interface with the model-view-controller pattern, it was very straightforward to update it for a new user interface. We went very easily from a console application to a web application!

# Using a Relational Database

To modify the application to use a relational database is even easier.

We have a class SqlCustomerDataMapper which implements the ICustomerDataMapper interface. The web.config file has the following entry:

<add key="CustomerDataMapper" value="SqlCustomerDataMapper"/>.

Our inversion of control container will then use the SqlCustomerDataMapper.[[1]](#footnote-1) The Repository will automatically use this data mapper class because it works with the data mapper through the ICustomerDataMapper interface.

The implementation of the SqlCustomerDataMapper class uses the appropriate Sql.Data.SqlClient namespace classes.

Again no other part of the application had to change.

# Web Services

Here we have to make two modifications. We have to put a remote service pattern in front of our business logic layer. We then will use the proxy pattern inside our model. Since we are using MVC, our user interface is unchanged.

## Service Layer Pattern

The service layer represents the application boundary by defining an interface that represents the access points to the business logic. We have, in fact, already done this with the ICustomerManager interface. Each operation on the service represents a set of business logic objects tied together by a particular operation. Service Layers[[2]](#footnote-2) come in two flavors.

One approach is to use a facade. A facade is a pattern that provides a uniform higher level interface to a subsystem.[[3]](#footnote-3) Another approach is to put application logic in the service layer using reusable domain objects. For example, a workflow application would tie domain objects such as Customer, Order, and Products in various ways depending on the particular action required. We have chosen the facade approach.

As we can see from the way our application has been structured up to now, a service layer does not have to be a remote call.

However, if you are going to use a remoting technology such as web services or REST, a service layer is the right approach. Distributing objects, as experience has demonstrated with DCOM, CORBA, and RMI, has not been a successful approach.

Since we have already the constructed the appropriate layer, it very simple to use Windows Communication Foundation (WCF) to transform our ICustomerManger interface into an ICustomerManagerService interface, and our CustomerManger class into our CustomerManagerService class. Since stateless web services scale much better than state full ones, we have gotten rid of the CustomerManagerFactory and instantiated a CustomerManger instance within each service layer call.

Now, the web.config file for the web service layer must have the mapping for the dependency injection container:

<add key="CustomerDataMapper" value="SqlCustomerDataMapper" />

The user interface’s model will now make calls on our web service interface:

public interface ICustomerManagerService

{

[OperationContract]

AddCustomerResponse AddCustomer(AddCustomerRequest request);

[OperationContract]

AddOrderForCustomerResponse

AddOrderForCustomer(AddOrderForCustomerRequest request);

[OperationContract]

RemoveCustomerResponse RemoveCustomer(RemoveCustomerRequest request);

[OperationContract]

GetCustomerResponse GetCustomerByName(GetCustomerByNameRequest request);

[OperationContract]

GetCustomerResponse GetCustomerById(GetCustomerByIdRequest request);

[OperationContract]

GetCustomersResponse GetCustomers(GetCustomersRequest request);

}

## Proxy Pattern

Since the model in the MVC pattern represents the business logic, it is the appropriate layer to add the calls to the application’s web services. We use a proxy pattern to represent the CustomerManagerService object in the client’s address space. We use the normal WCF mechanism to generate the proxy. The model layer because a facade to hide the details of the web service plumbing from the user interface’s model. For example, the AddOrderForCustomer in the model becomes:

public static void AddOrderForCustomer(ModelCustomer c,

IList<string> items)

{

CustomerManagerServiceClient proxy = new

CustomerManagerServiceClient();

string[] list = new string[items.Count];

items.CopyTo(list, 0);

proxy.AddOrderForCustomer(c.Id, list);

}

CustomerManagerServiceClient is the WCF generated proxy.

# Summary

This unit demonstrated the power of good partitioning and layering. We used the existing partitioning developed in the previous units to easily upgrade the technologies used in our application. Interface based development, and proper physical design are the basic principles.

While much more could be said about partitioning and layering an application, we have introduced the key basic concepts. We have also demonstrated how good partitioning and layering can make it straight-forward to upgrade and evolve an application.

1. We did not have to use a dependency injection container. We could have just created an instance of the SqlCustomerDataMapper class in the CustomerManagerFactory. [↑](#footnote-ref-1)
2. Fowler, *Patterns of Enterprise Application Architecture*, Chapter 9. [↑](#footnote-ref-2)
3. Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Chapter 4 [↑](#footnote-ref-3)