

Flerlager- applikation



Upphovsrätt för detta verk

Detta verk är framtaget i anslutning till kursen ASP.NET Web Forms vid Linnéuniversitetet.

Du får använda detta verk så här:

Allt innehåll i detta verk av Mats Looch, förutom Linnéuniversitetets logotyp och symbol samt ikoner, bilder och fotografier, är licensierad under:



Creative Commons Erkännande 4.0 Internationell licens.

<http://creativecommons.org/licenses/by/4.0>

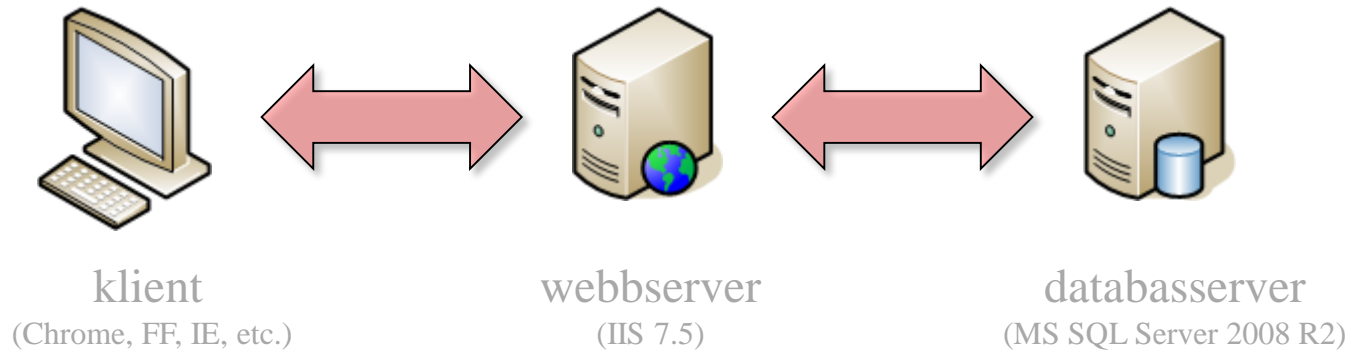
Det betyder att du i icke-kommersiella syften får:

- kopiera hela eller delar av innehållet
- sprida hela eller delar av innehållet
- visa hela eller delar av innehållet offentligt och digitalt
- konvertera innehållet till annat format
- du får även göra om innehållet

Om du förändrar innehållet så ta inte med Linnéuniversitetets logotyp och symbol samt ikoner och fotografier i din nya version!

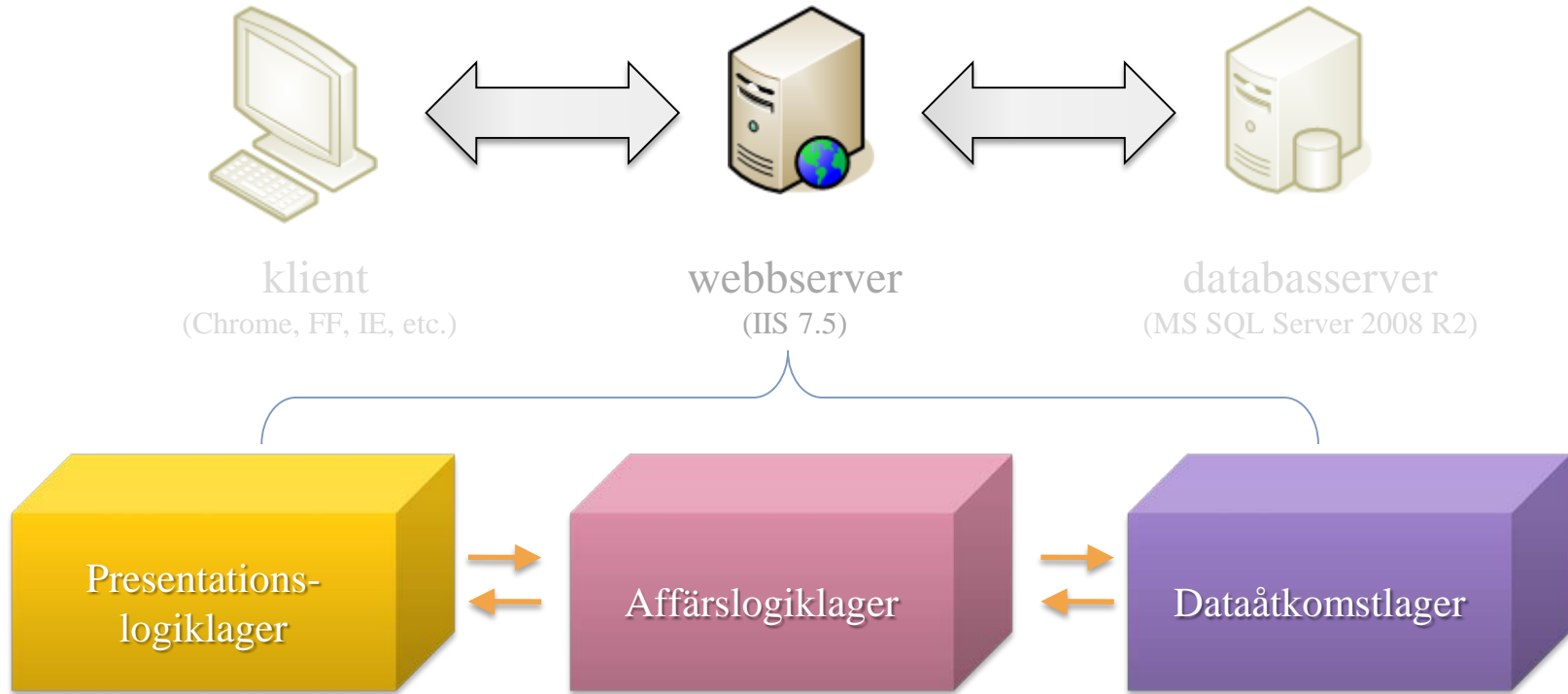
Vid all användning måste du ange källan: "Linnéuniversitetet – ASP.NET Web Forms" och en länk till <https://coursepress.lnu.se/kurs/aspnet-web-forms> och till Creative Common-licensen här ovan.

Datadriven webbapplikation – en distribuerad arkitektur



- ✓ Huvuduppgiften är att hämta, visa och modifiera data.
- ✓ Applikationen finns "utspridd" på flera olika fysiska datorer – fysiska lager.

Logiska lager

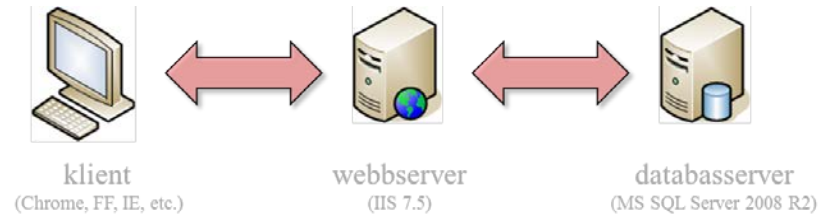


- ✓ En distribuerad applikation kan diskuteras i form av logiska lager.
- ✓ Logiska lager kan finnas på en enskild dator eller uppdelad på flera datorer – den logiska arkitekturen definierar inte hur.

Fördelar med flera lager

✓ En bra fysisk arkitektur ger:

- Prestanda
- Skalbarhet
- Feltolerans
- Säkerhet

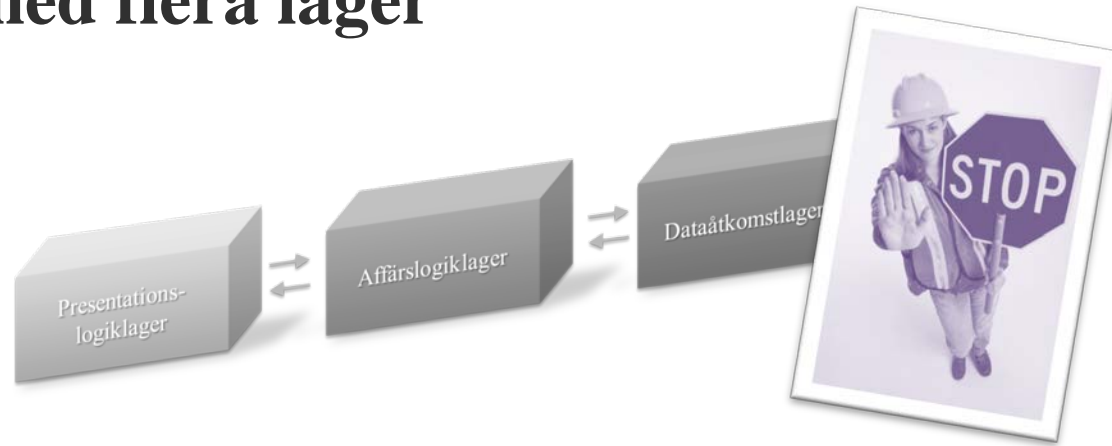


✓ En bra logisk arkitektur ger:

- Logiskt organiserad kod
- Enklare underhåll
- Bättre återanvändning av kod
- Enklare att utveckla i grupp
- Tydligare kod



Nackdelar med flera lager



- ✓ En flerlagerarkitektur är ett sätt att förenkla en stor och omfattande applikation och minska dess komplexitet. Men det behöver inte gälla alla typer av applikationer!
- ✓ Om det t.ex. är fråga om en liten applikation är det kanske enklare att hålla sig till så få lager som möjligt eftersom ju fler lager som införs desto mer komplex applikation blir det.

En logisk arkitektur med fem lager



Användargränssnitt

✓ **Användargränssnitt** – lagret presenterar information och samlar indata från användaren. Utgörs av webbläsare.

Presentationslogik

✓ **Presentationslogik** – lagret bestämmer vad som ska visas, navigeringsalternativ och hur indata ska tolkas.

Affärslogik

✓ **Affärslogik** – lagret innehåller applikationens kod för "affärsregler", validering av data, manipulering av data, beräkningar och säkerhet. Lagret måste vara separerat från presentationslogiklagret. Kod, t.ex. validering, kan finnas duplicerat i presentationslogiklagret för att ge ett rikare gränssnitt, men affärslogiklagret måste innehålla all kod för affärslogiken.

Dataåtkomst

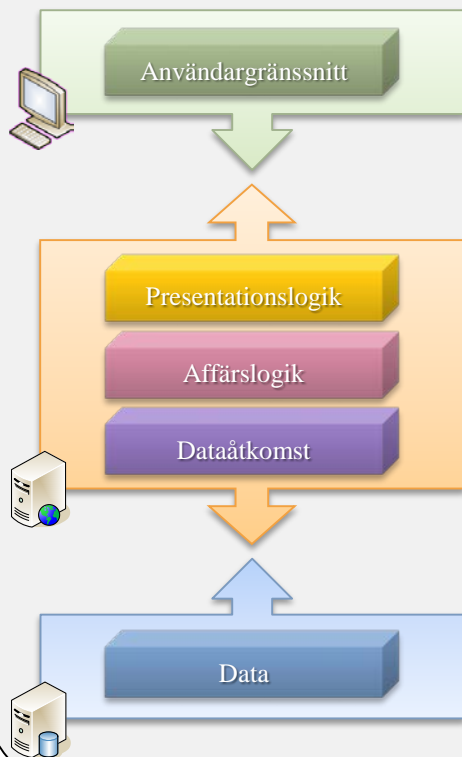
✓ **Dataåtkomst** – lagret kommunicerar med datalagret för att hämta, lägga till, uppdatera och ta bort information. Lagret hanterar eller lagrar inte någon information; dess enda uppgift är att vara en länk mellan affärslogiklagret och datalagret.

Data

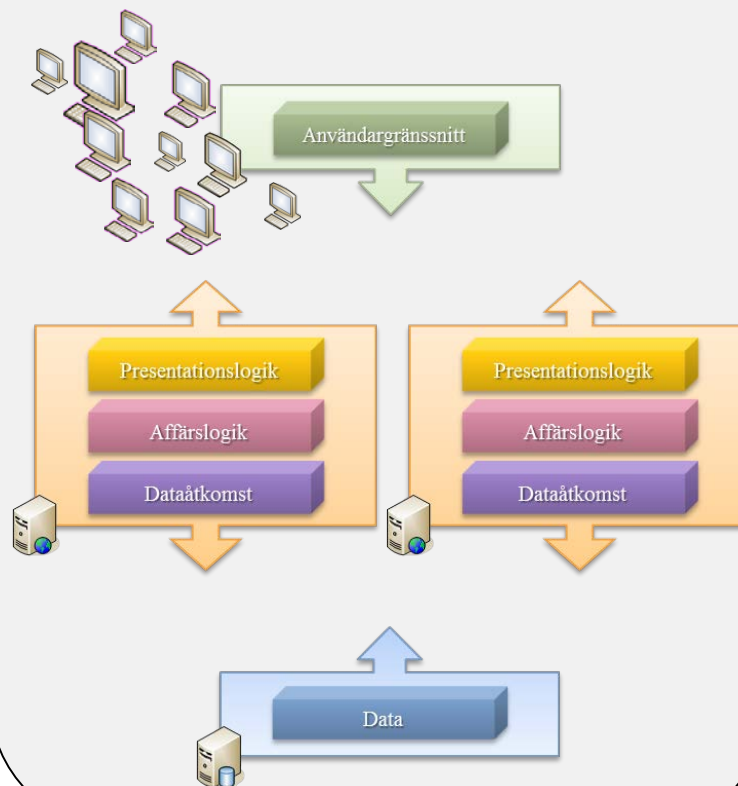
✓ **Data** – Skapar, hämtar, uppdaterar och tar bort data fysiskt. Implementeras i regel av en databasserver.

Skalbar arkitektur

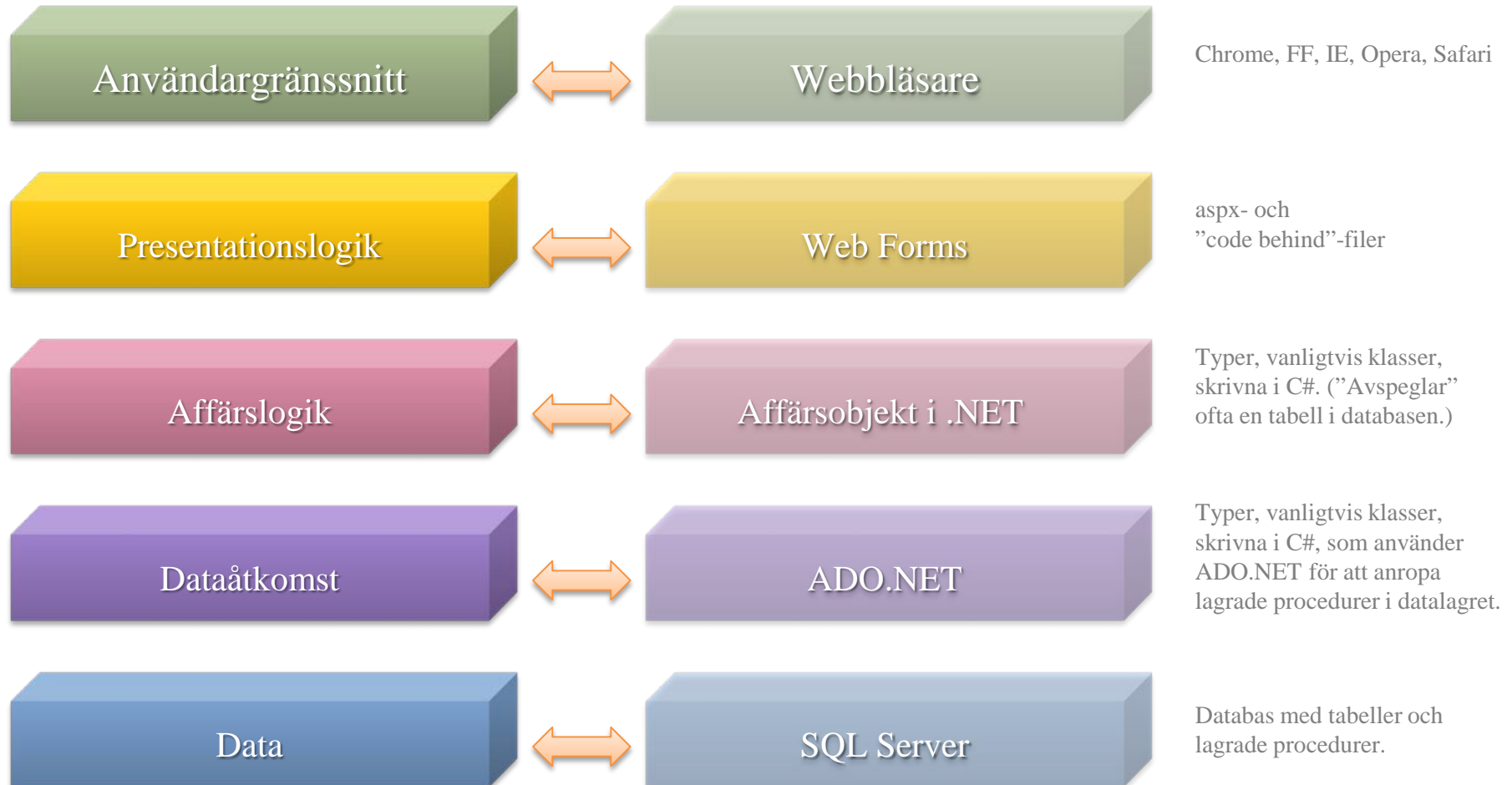
Fem logiska lager som de kan användas i samband med webbapplikationer.



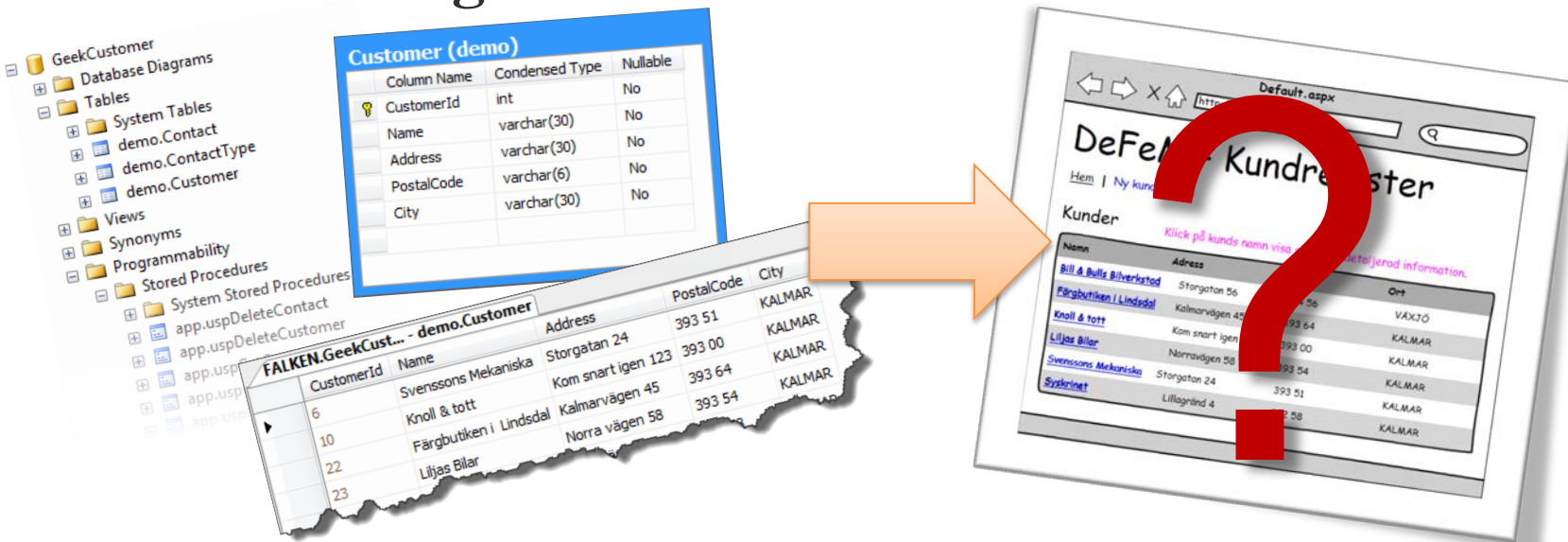
Fem logiska lager som de kan användas i samband med en lastbalanserande "web farm".



Logiska lager i praktiken



Från datalagret till klienten



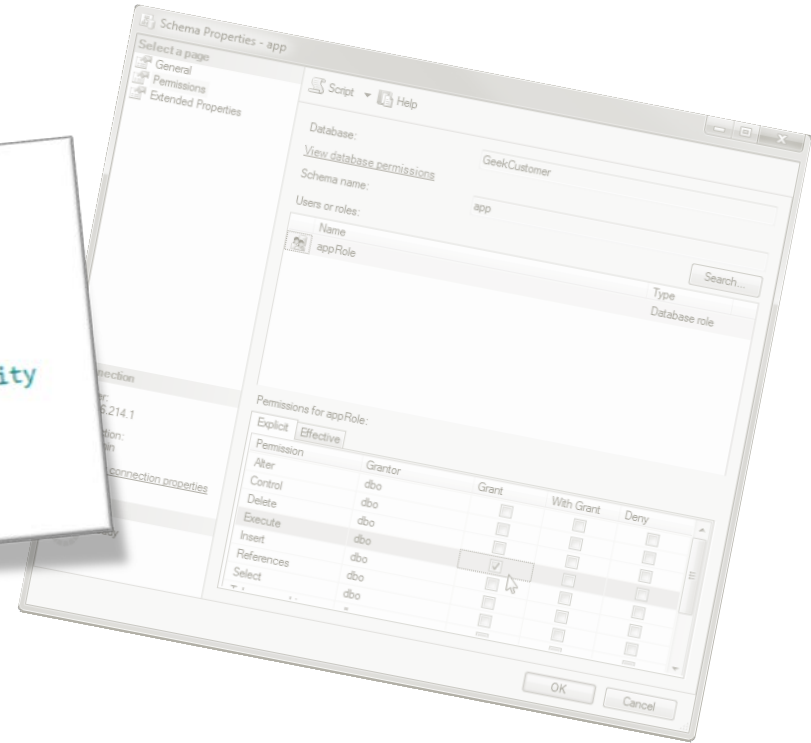
✓ Vad behöver jag göra för kunna visa innehållet i tabellen **Customer** som finns i databasen **GeekCustomer**?

- Användargränssnittlagret får ombesörja med hjälp av en **ListView** och en **ObjectDataSource** att kunderna renderas och skickas till klienten i form av (X)HTML.
- En affärslogikklass måste skapas med motsvarande egenskaper/fält som finns i tabellen **Customer**.
- I dataåtkomstlagret måste en klass med en metod skapas som hämtar data från tabellen **Customer** via en lagrad procedur.
- Datalagret måste kompletteras med en lagrad procedur som ger alla poster tabellen **Customer** innehåller.

Den lagrade proceduren

```
ALTER PROCEDURE [app].[uspGetCustomers]
AS
BEGIN
    SET NOCOUNT ON;

    SELECT      CustomerId, Name, [Address], PostalCode, City
    FROM        demo.Customer
    ORDER BY Name, City, [Address], PostalCode
END
```



- ✓ Den lagrade proceduren metoden **app.uspGetCustomers** returnerar helt enkelt alla poster i tabellen **Customer**.
- ✓ Användaren **appUser** har rätt att exekvera den lagrade proceduren, via rollen **appRole** och schemat **app**, men saknar direkta rättigheter till tabellen **Customer**.

...och några SPROCs till

```
ALTER PROCEDURE [app].[uspInsertCustomer]
    @CustomerId int OUTPUT,
    @Name varchar(30),
    @Address varchar(30),
    @PostalCode varchar(6),
    @City varchar(30)
AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO demo.Customer
        (Name, Address, PostalCode, City)
    VALUES
        (@Name, @Address, @PostalCode, @City)

    SET @CustomerId = SCOPE_IDENTITY()
END
```

```
ALTER PROCEDURE [app].[uspUpdateCustomer]
    @CustomerId int,
    @Name varchar(30),
    @Address varchar(30),
    @PostalCode varchar(6),
    @City varchar(30)
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS(SELECT (0) FROM demo.Customer WHERE CustomerId = @CustomerId)
    BEGIN
        UPDATE demo.Customer
        SET
            Name = @Name, [Address] = @Address, PostalCode = @PostalCode, City = @City
        WHERE
            (CustomerId = @CustomerId)
    END

    IF @@ROWCOUNT = 0
    BEGIN
        RAISERROR('The customer was not updated.', 16, 1)
    END
END
```

```
ALTER PROCEDURE [app].[uspDeleteCustomer]
    @CustomerId int
AS
BEGIN
    SET NOCOUNT ON;

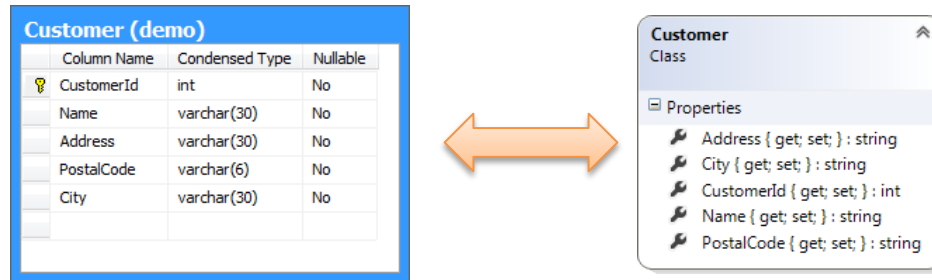
    IF EXISTS(SELECT (0) FROM demo.Customer WHERE CustomerId = @CustomerId)
    BEGIN
        DELETE FROM demo.Customer
        WHERE (CustomerId = @CustomerId)
    END

    IF @@ROWCOUNT = 0
    BEGIN
        RAISERROR('The customer was not deleted.', 16, 1)
        RETURN (1)
    END

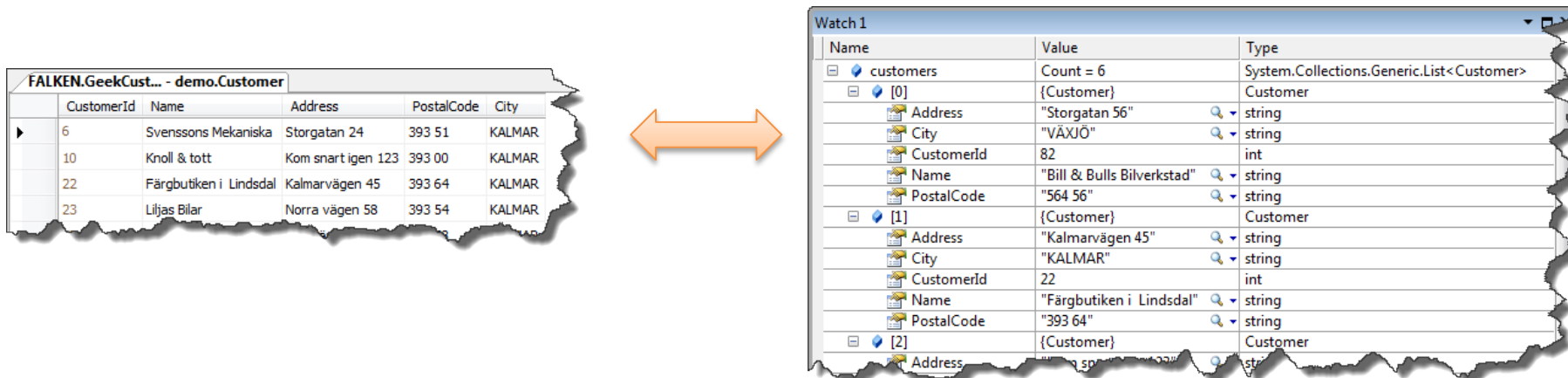
    RETURN (0)
END
```

Affärslagerklassen (1 av 2)

- ✓ Tabellen **Customer** i datalagret motsvaras av klassen **Customer** i affärslogiklaget. Tabellen definierar vilka fält och typer som **Customer** måste implementera.



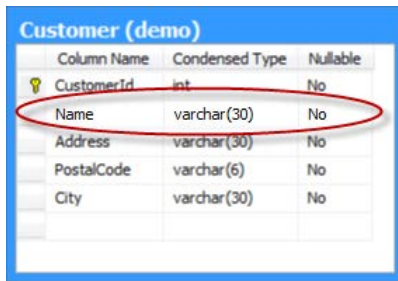
- ✓ Varje post (rad) i tabellen **Customer** motsvaras av objekt av typen **Customer**.



Affärslagerklassen (2 av 2)

- ✓ Genom att använda egenskaper kan du säkerställa att ett validerat objekt av affärslogiklagerklassen aldrig kan innehålla data som inte kan representeras i databasen.
- ✓ Fältet **Name** i tabellen **Customer** kan som mest innehålla 30 tecken och får inte vara NULL.
- ✓ Affärslogiklagerklassen **Customer** implementerar tabellen **Customer** och fältet **Name** med hjälp av den autoimplementerade egenskapen **Name**.

(OBS! Det saknas för tillfället attribut i klassen som undersöker om Name tilldelas null eller en tom sträng(?), och om strängen har fler än 30 tecken.)



Column Name	Condensed Type	Nullable
CustomerId	int	No
Name	varchar(30)	No
Address	varchar(30)	No
PostalCode	varchar(6)	No
City	varchar(30)	No

```
/// <summary>
/// Klass för hantering av kunduppgifter.
/// </summary>
public class Customer
{
    // Egenskapernas namn och typ ges av tabellen
    // Customer i databasen.
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string PostalCode { get; set; }
    public string City { get; set; }
}
```

Dataåtkomstklassen

- ✓ Affärslogiklagerklassen **Customer** kommunicerar inte direkt med tabellen **Customer** i databasen. All kommunikation går genom en klass i dataåtkomstlagret, vars en uppgift är att slussa information mellan affärslogiklagret och datalagret.
- ✓ Dataåtkomstklassen **CustomerDAL** (DAL = Data Access Layer) implementerar metoden **GetCustomers** som returnerar en lista med referenser till affärslogiklagerobjekt av typen **Customer** som skapas av metoden, ett objekt för varje post.

```
public IEnumerable<Customer> GetCustomers()
{
    string connectionString =
        WebConfigurationManager.ConnectionStrings["GeekCustomerConnectionString"].ConnectionString;
    using (var conn = new SqlConnection(connectionString))
    {
        try
        {
            var customers = new List<Customer>(100);

            var cmd = new SqlCommand("app.uspGetCustomers", conn);
            cmd.CommandType = CommandType.StoredProcedure;
            conn.Open();

            using (var reader = cmd.ExecuteReader())
            {
                var customerIdIndex = reader.GetOrdinal("CustomerId");
                var nameIndex = reader.GetOrdinal("Name");
                var addressIndex = reader.GetOrdinal("Address");
                var postalCodeIndex = reader.GetOrdinal("PostalCode");
                var cityIndex = reader.GetOrdinal("City");

                while (reader.Read())
                {
                    customers.Add(new Customer
                    {
                        CustomerId = reader.GetInt32(customerIdIndex),
                        Name = reader.GetString(nameIndex),
                        Address = reader.GetString(addressIndex),
                        PostalCode = reader.GetString(postalCodeIndex),
                        City = reader.GetString(cityIndex)
                    });
                }
            }

            customers.TrimExcess();

            return customers;
        }
        catch
        {
            throw new ApplicationException("An error occurred while getting customers from the database.");
        }
    }
}
```


Service använder CustomerDAL

- ✓ En tumregel är att ett lager får aldrig ”hoppas över”, d.v.s. presentationslogiklagret måste alltid gå genom affärslogiklagret och får inte använda typer i dataåtkomstlagret direkt.
- ✓ Det är affärslogiklagerklassen **Service** som använder CustomerDAL och innehåller därför bland annat en metod som returnerar alla kunder.

```
public class Service
{
    private CustomerDAL _customerDAL;

    private CustomerDAL CustomerDAL
    {
        // Ett CustomerDAL-objekt skapas först då det behövs för första
        // gången (lazy initialization, http://en.wikipedia.org/wiki/Lazy\_initialization).
        get { return _customerDAL ?? (_customerDAL = new CustomerDAL()); }
    }

    public void DeleteCustomer(int customerId) {...}

    public Customer GetCustomer(int customerId) {...}

    public IEnumerable<Customer> GetCustomers()
    {
        return CustomerDAL.GetCustomers();
    }

    public void SaveCustomer(Customer customer) {...}
}
```


Presentationslogiklagret

- ✓ Det "enklaste" återstår – presentationen av data med hjälp av ett ListView-objekt.

```
<%--  
Visar alla kunder. Innehåller även kommandoknappar för att lägga till, uppdatera och ta bort kunder.  
Hämtar alla kunduppgifter som finns i tabellen Customer i databasen via affärslogikklassen Service och  
metoden GetCustomers, som i sin tur använder klassen CustomerDAL och metoden GetCustomers, som skapar en  
lista med referenser till Customer-objekt; ett Customer-objekt för varje post i tabellen.  
--%>  
<%  
<asp:ListView ID="CustomerListView" runat="server" ItemType="GeekCustomer.Model.Customer"  
SelectMethod="CustomerListView_GetData" InsertMethod="CustomerListView_InsertItem"  
UpdateMethod="CustomerListView_UpdateItem" DeleteMethod="CustomerListView_DeleteItem"  
DataKeyNames="CustomerId" InsertItemPosition="FirstItem">  
<LayoutTemplate>  
<table class="grid">  
<tr>  
<th>  
<th> Namn  
</th>  
<th>  
<th> Adress  
</th>  
<th>  
<th> Postnummer  
</th>  
<th>  
<th> Ort  
</th>  
</tr>  
<tr>  
<td colspan="5">Platshållare för nya rader --%>  
<asp:Placeholder ID="itemPlaceholder" runat="server" />  
</tr>  
</table>  
</LayoutTemplate>  
<ItemTemplate>  
<%-- Mall för nya rader. --%>  
<tr>  
<td>  
<%# Item.Name %>  
</td>  
<td>  
<%# Item.Address %>  
</td>  
<td>  
<%# Item.PostalCode %>  
</td>  
<td>  
<%# Item.City %>  
</td>  
<td class="command">  
<!-- "Kommandknappar" för att ta bort och redigera kunduppgifter. Kommandonamnen är VIKTIGA! --%>  
<%-- "Kommandknappar" för att ta bort och redigera kunduppgifter. Kommandonamnen är VIKTIGA! --%>  
<asp:LinkButton runat="server" CommandName="Delete" Text="Ta bort" CausesValidation="false" />  
<asp:LinkButton runat="server" CommandName="Edit" Text="Redigera" CausesValidation="false" />  
</td>  
</tr>  
</ItemTemplate>  
</LayoutTemplate>  
</asp:ListView>  
<%-- Kunduppgifter saknas i databasen. --%>
```

ASP.NET Web Forms (1DV406) → Föreläsning → Exempel

Kundkontakter (version 2, CRUD)

Namn	Adress	Postnummer	Ort	
Bill & Bulls Bilverkstad	Storgatan 56	564 56	VÄXJÖ	Lägg till Rensa
Färgbunken i Lindsdal	Kalmarvägen 45	393 64	KALMAR	Ta bort Redigera
Knoll & tott	Kom snart igen 123	393 00	KALMAR	Ta bort Redigera
Liljas Bilar	Norra vägen 58	393 54	KALMAR	Ta bort Redigera
Svenssons Mekaniska	Storgatan 24	393 51	KALMAR	Ta bort Redigera
Syskrinet	Lillagränd 4	392 58	KALMAR	Ta bort Redigera

© BY-NC-SA

Linnéuniversitetet | Fakulteten för teknik | Institutionen för datavetenskap

...och mycket mer finns att läsa...

- ✓ ...om "Data Components and the DataSet" i kapitel 8 på sidorna 321-333.

