Umeå universitet
Institutionen för datavetenskap
Examensarbete D, 20p

Arkitekturförslag för webapplikation

Robert Ögren
c00ron@cs.umu.se
robert.ogren@prodacapo.com

31 januari 2005

Intern handledare:
Oscar Appelgren

Extern handledare:
Lars Näslund

Sammanfattning

En webapplikation är en mjukvaruapplikation som används via en webläsare och där programvaran är installerad centralt på en eller flera webservrar. Till skillnad från statiska websidor är en webapplikation interaktiv.

I denna rapport redovisas ett examensarbete i datavetenskap utfört vid företaget Prodacapo AB. Rapporten beskriver ett arkitekturförslag för en webapplikation avsedd att utvecklas för plattformen Microsoft .NET och med användande av utvecklingsverktyget Borland Delphi for .NET. Designen av arkitekturen har gjorts för att uppnå god skalbarhet och prestanda, och dessutom för att kunna användas även med Windows-baserade klienter och gränssnitt till andra system. En annan viktig faktor var systemets säkerhet.

En slutsats som kunde dras från arbetet var att det borde vara möjligt att implementera den föreslagna designen men att den flexibilitet som lösningen ger kommer till priset av ökad komplexitet jämfört med en enkel webapplikation som endast tillåter webklienter och är implementerad i ren ASP.NET.

Architecture suggestion for web application

Abstract

A web application is a software application that is used through a web browser and where the software is installed centrally on one or more web servers. As opposed to static web pages, a web application is interactive.

This report describes a master's thesis project in computing science performed at the company Prodacapo AB. The goal of the project was to create a suggestion for a new system architecture for a web application using the Microsoft .NET platform and the Borland Delphi for .NET development tool.

The web application was a part of the Prodacapo software product, which contains functionality such as activity based costing and management, and Balanced Scorecard.

Since the system can be deployed in large organizations with many users scalability was an important concern. Security was also important due to the sensitive nature of the information managed by the system such as financial figures. An additional design consideration was to enable sharing of program code with other parts of the system, such as a Windows based application.

To increase maintainability and code reuse it is common to partition a system into several logical layers, for example the user interface layer, the application logic layer and the data layer. The only layer that has to be replaced to create a Windows-based application instead of a web application is the user interface layer. When the actual system is built some of these logical layers can be part of the same physical layer.

To achieve good scalability it was desirable to keep the number of physical layers low so that the system capacity could be increased by adding more web servers. At the same time it was desirable to access the application logic in other ways, for example for the Windows-based client.

The final design suggestion was centered around an application logic layer containing a set of application services that handled the application logic. These services could either be loaded directly into the same process as the one containing the user interface, or be accessed remotely for example from a Windows-based client. The first option is recommended for a web interface yielding good performance and good scalability since multiple web servers could easily be set up with a load balancer that balances the requests from the users.

Some of the conclusions that could be drawn from the project was that it should be possible to implement the system using the suggested design, but that the flexibility of the chosen solution comes at the cost of added complexity compared to a plain web application that only allows web clients and that is implemented in plain ASP.NET. It was also obvious that a lot of alternate solutions existed so there is a lot of room for future work.

Innehåll

Kapitel 1

Inledning

I denna rapport redovisas ett 20p examensarbete i datavetenskap. Examensarbetet gick ut på att ge ett förslag till en ny systemdesign för en webapplikation. Detta kapitel ger en översikt av uppgiften samt beskriver hur resten av rapporten är uppbyggd.

1.1 Bakgrund

Företaget Prodacapo® AB tillverkar en mjukvaruprodukt för verksamhetsstyrning och beslutsstöd. Produkten, som också heter Prodacapo, kan användas både via ett Windows®-baserat program samt via en weblösning som används genom en vanlig webläsare.

Borland® Delphi™, det utvecklingsverktyg som Prodacapo använder för hela produkten, har kommit ut i en ny version för plattformen Microsoft® .NET. Det är önskvärt att kunna använda denna version i första hand för weblösningen för att kunna dra nytta av det kraftfulla stödet för att bygga webapplikationer i ASP.NET, vilket är en del i plattformen .NET. Den äldre versionen av utvecklingsverktyget saknar stöd för .NET och bytet av utvecklingsverktyg kommer därför att kräva en hel del förändringar även om den existerande designen skulle bibehållas. Dock är det önskvärt att titta på alternativa designer både för att fullt utnyttja styrkan i det nya verktyget och för att lättare uppfylla de allt större önskemålen att göra större delar av applikationen tillgänglig via ett webgränssnitt.

1.2 Uppgift

Uppgiften som skulle utföras var att ge ett förslag till en ny systemdesign för weblösningen.

Då examensarbetet utfördes åt ett kommersiellt företag kan inte alla detaljer i den existerande lösningen redovisas i denna rapport. Av samma skäl är den prototypkod som utvecklats inte allmänt tillgänglig.

1.3 Rapportens uppbyggnad

Återstoden av rapporten är uppbyggd på följande sätt:

Kapitel 2 ger en detaljerad beskrivning av uppgiften och syftet med arbetet.

Kapitel 3 beskriver tillvägagångssättet som användes för att utföra uppgiften.

Kapitel 4 ger en introduktion till olika tekniker och verktyg som ligger till grund för examensarbetet.

Kapitel 5 ger en översikt av olika tekniker som kan användas för att kommunicera data mellan en databas och en webapplikation.

Kapitel 6 förklarar de val som gjorts och som ligger till grunden för den slutliga designen.

Kapitel 7 beskriver den slutliga designen som tagits fram samt den prototypimplementation som gjorts.

Kapitel 8 förklarar vilka slutsatser som kan dras av arbetet samt ger förslag till vidareutveckling.

Kapitel 9 tackar de personer som på olika sätt underlättat arbetets utförande.

I slutet av rapporten finns också ett sakregister samt en källförteckning.

1.4 Varumärken

Kapitel 2

Problembeskrivning

Detta kapitel ger en närmare beskrivning av det sammanhang examensarbetet ingår i, uppgiften som skulle utföras och vilka avgränsningar som gjorts.

2.1 Sammanhang

Detta avsnitt ger en översikt av det system som examensarbetet ingår som en del i.

Mjukvaruprodukten Prodacapo innehåller ett stort antal funktioner. En viktig del är ABC/ABM vilket står för aktivitetsbaserad kalkylering och styrning, där till exempel kund- och produktlönsamhet samt organisationers kostnadseffektivitet kan beräknas. Kostnader i en organisation spåras från konton via resurser, de aktiviteter de utför och slutligen till produkter och kunder. En annan viktig del är *Balanced Scorecard*, vilket är ett hjälpmedel för att sätta upp en vision och strategi för en organisation och mäta hur väl det följs. Det som mäts är inte bara ekonomiska aspekter utan också saker som kundnöjdhet och personalens utveckling och välmående.

Programvaran lagrar data i en relationsdatabas. Några viktiga typer av information som lagras i databasen är information om basobjekt, till exempel avdelningar och resurser, beräknade kostnader för olika objekt, användarbehörigheter och systeminställningar.

Det finns ett antal huvudsakliga sätt att presentera informationen. Basobjekt kan visas antingen enskilt eller i tabellform, till exempel alla resurser tillhörande en avdelning. Ekonomiska resultat för till exempel en avdelning visas ofta i tabellform eller som grafer. Ett mer avancerat sätt att presentera information är grafiska kartor, till exempel över en organisationsstruktur eller innehållet i ett styrkort. En sådan karta sätts samman av information rörande många olika objekt. Dessutom kan navigationshjälpmedel krävas, till exempel en trädstruktur med alla avdelningar i en organisation.

Gemensamt för den mesta information som ska presenteras är att den erhålls genom att kombinera information från flera olika tabeller i databasen och filtrera, gruppera och summera. Informationsmängden kan vara rätt stor, till exempel är det inte orimligt att en stor organisation har 100 000 produkter. Detta gör det nödvändigt att tänka på prestandan och låta databasservern göra så mycket jobb som möjligt och inte hämta onödig information.

2.2 Uppgift

Uppgiften som skulle utföras i examensarbetet var att ge ett förslag till systemdesign för weblösningen där plattformen .NET och utvecklingsverktyget *Borland Delphi for the Microsoft .NET Framework* användes. Systemdesignen skulle beskriva vilka komponenter lösningen består av och hur de interagerar med varandra.

För att testa olika tekniker ingick det att skriva lite prototypprogramkod, men inte att utveckla någon produktionsfärdig kod.

Examensarbetet skulle vara av en undersökande natur och användas som underlag eller förstudie för den faktiska designen och implementationen. Av denna anledning kan vissa av kraven som redovisas härnäst komma att förändras om det visar sig nödvändigt.

2.3 Krav

I detta avsnitt redovisas några av de krav som ställdes på webapplikationen. Av konkurrensskäl kan inte alla krav redovisas i detalj i denna rapport.

2.3.1 Datalagring

Data skulle lagras i en relationsdatabas och i huvudsak använda existerande databasschema för att weblösningen skulle fungera tillsammans med den existerande Windows-baserade applikationen. Dock kunde tillägg till och vissa förändringar av databasschemat göras.

Webapplikationen skulle stödja åtkomst till flera databasinstanser samtidigt, och kunna hantera flera olika databasserverprodukter, däribland Oracle® Database och Microsoft SQL Server.

2.3.2 Säkerhet

Säkerhet var en viktig aspekt av designen då applikationen behandlar känslig information. På säkerhetssystemet ställdes kravet att det skulle hantera användare med olika rättigheter enligt befintliga behörighetstabeller. Det var också önskvärt att integrera med Microsoft Active Directory®, vilket är en katalogtjänst som bland annat kan hålla reda på användare och användargrupper i ett nätverk av datorer, samt att tillhandahålla gemensam inloggning (*single sign on*), vilket innebär att en användare inte ska behöva logga in flera gånger om weblösningen används som en del i en portallösning där flera applikationer är tillgängliga. Avsnitt 4.7 beskriver detta närmare.

2.3.3 Prestanda och skalbarhet

Webapplikationen ska användas i företagsnätverk med många samtidiga användare. Som tidigare beskrivits kan också datamängderna vara relativt stora. Av denna anledning var det nödvändigt att tänka på prestanda och skalbarhet vid utformningen av designen.

2.3.4 Övriga krav

Det var önskvärt att kunna återanvända programkod mellan den Windows-baserade applikationen och webapplikationen.

2.4 Avgränsning

För att examensarbetet skulle ge en rimlig arbetsbörda var det nödvändigt med ett flertal avgränsningar.

Det primära utvecklingsverktyg som användes var *Borland Delphi 8 for the Microsoft .NET Framework*, eftersom det fanns önskemål från arbetsgivaren att använda det. En anledning till detta var att det fanns existerande programkod skriven med tidigare versioner av detta verktyg. Detta innebar att en del potentiellt användbara tekniker och verktyg, till exempel Enterprise JavaBeans™, inte användes i examensarbetet, även om de bidrog med viss inspiration.

De tidigare nämnda kraven begränsade också lösningen. Till exempel uteslöt kravet på att en relationsdatabas med existerande databasschema skulle användas objektdatabaser som en möjlig komponent i lösningen.

I de relationsdatabaser som användes i lösningen så användes endast de vanliga primitiva datatyperna såsom heltal och strängar samt binärdata (BLOB). Vissa databasserverprodukter har utökad funktionalitet för mer komplexa datatyper, men dessa användes inte i denna lösning, bland annat av kompatibilitetsskäl.

Kapitel 3

Metodbeskrivning

Detta kapitel beskriver vilket tillvägagångssätt som använts under arbetet, bland annat hur källor valts ut och hur designen skapats.

3.1 Tillvägagångssätt

Arbetet inleddes med att fastställa syftet med examensarbetet, upprätta en preliminär tidsplan, samt fastställa de krav och önskvärda egenskaper lösningen skulle uppfylla. Därefter utfördes en teoretisk fördjupningsstudie om datahantering i webapplikationer, eftersom detta var i högsta grad relevant för utformningen av designen.

Efter detta identifierades några huvudsakliga val som måste göras för designen, och designarbetet inleddes. Det blev också uppenbart att ytterligare avgränsningar behövdes, och det beslöts att tonvikten skulle läggas på datahantering och struktur.

Vid arbetet med designen användes en *top-down*-metodik, vilket innebar att hela systemet först designades i grova drag och därefter förfinades delarna successivt. Parallellt med designarbetet utvecklades en enkel prototyp för att testa hur tekniker och struktur fungerade i verkligheten. Erfarenheterna från prototyparbetet användes för att justera designen eftersom.

3.2 Litteratursökning

Vetenskapliga artiklar har främst sökts i *The ACM Digital Library* och *CiteSeer*. Böcker har sökts bland annat på bokhandeln Amazons webplats och Umeå Universitetsbibliotek. Websidor har sökts med sökmotorn Google, och viss information har inhämtats från diskussionsgrupper (*newsgroups*). Några exempel på sökfraser som använts är “web 3-tier”, “web architecture”, “web middleware”, “ASP.NET” och “ADO.NET”. De tre förstnämnda sökfraserna valdes för att hitta information om weblösningar som inte pratar direkt med databasen utan är uppdelade i flera lager (*tiers*), vilket är intressant för fördjupningsstudien som utfördes. De två sistnämnda är namn på tekniker som är användbara vid utveckling av weblösningar i plattformen Microsoft .NET, och beskrivs kortfattat i avsnitt 4.9.

3.3 Val av källor

Artiklar publicerade i vetenskapliga tidskrifter har föredragits där så var möjligt, eftersom de kan antas ha en hög grad av tillförlitlighet eftersom de granskats innan publicering. Tryckta böcker och artiklar som ej granskats kan ha något mindre tillförlitlighet. Vanliga websidor har mest använts för att få en översikt av ett ämne och som vägledning för vidare litteratursökning.

Eftersom webapplikationer är ett relativt nytt fenomen och stora framsteg har gjorts vad gäller utvecklingsverktygens och plattformarnas användbarhet under de senaste åren så har källor skrivna efter år 2000 föredragits. Ett undantag är information rörande relationsdatabaser, då dessa har funnits betydligt längre.

När det gäller information från tillverkare av produkter så förutsätts det tekniska innehållet ha hög tillförlitlighet medan diskussioner angående produkternas fördelar samt jämförelser med andra produkter antas vara otillförlitliga då det ligger i tillverkarens intresse att låta sina produkter framstå som bättre än konkurrenternas.

Kapitel 4

Översikt av tekniker

Detta kapitel ger en kortfattad introduktion till tekniker och verktyg som används i den existerande lösningen samt den tänkta, för att läsare som ej är bekanta med dessa ska kunna förstå sammanhanget. För mer ingående beskrivningar, se litteraturreferenserna i varje avsnitt.

4.1 HTML och HTTP

HyperText Markup Language (HTML) och *HyperText Transfer Protocol* (HTTP) är fundamentala begrepp för websidor.

HTML är ett sidbeskrivningsspråk som beskriver innehåll och layout i en websida. En HTML-fil är en textfil som innehåller formateringsdirektiv, så kallade taggar (*tags*). (Raggett et al., 1999, Pfaffenberger och Gutzman, 1998)

HTTP är det protokoll som används för kommunikation mellan en webserver och en webläsare. HTTP-protokollet är tillståndslöst (*stateless*), vilket innebär att webservern inte sparar någon information från ett klientanrop till ett annat och att varje anrop är fristående från de andra. När en webläsare vill ha en fil, till exempel en HTML-fil eller en bildfil, från en webserver, skickas en förfrågan, och servern skickar tillbaka begärd fil eller en felkod. (Fielding et al., 1999, Coulouris et al., 2001)

I webapplikationer är det ofta nödvändigt att behålla information mellan anrop, till exempel för att hålla reda på om en användare är inloggad. Detta kan göras genom så kallade *cookies*, vilket är en mindre mängd information, till exempel ett unikt identifikationsnummer, som lagras av användarens webläsare. Då webservern skickar tillbaka en websida kan den också skicka med en begäran att lagra en cookie. Om webläsaren godtar denna begäran kommer cookien att skickas med i alla efterföljande anrop till webservern som på så sätt kan identifiera användaren. Cookies kan också användas till exempel för att spara personliga inställningar för en användare mellan olika sessioner. (Kristol och Montulli, 2000)

4.2 CSS

Cascading Style Sheets (CSS) används för att separera information rörande utseende från innehållet i en websida. Med hjälp av CSS kan till exempel typsnitt,

marginaler och färger specificeras i en separat fil som ett flertal HTML-sidor refererar till. Detta gör det enkelt att ha ett konsekvent utseende på alla sidor och det blir enkelt att byta utseende. (Pfaffenberger och Gutzman, 1998)

4.3 JavaScript

JavaScript är ett programmeringsspråk som kan användas i websidor. Det kan hantera händelser, till exempel då användaren trycker på en knapp, och manipulera innehållet i websidan. Detta gör det möjligt att till exempel kontrollera att användaren har fyllt i ett formulär korrekt utan att något anrop till webservern behöver göras. Av säkerhetsskäl finns det ett flertal begränsningar av vad som får göras, till exempel är det inte tillåtet att läsa filer på användarens hårddisk. (Pfaffenberger och Gutzman, 1998)

4.4 XML

Extensible Markup Language (XML) är ett standardiserat textformat för utväxling av strukturerad information. En stor fördel med XML är att det är oberoende av plattform och utvecklingsverktyg vilket gör det enkelt att utbyta information mellan olika system. (Bos, 2001)

4.5 XML-webtjänster

XML-webtjänster (*XML Web Services*) utnyttjar XML för att tillhandahålla fjärranrop i distribuerade system oberoende av vilken plattform och vilket utvecklingsverktyg som använts för att bygga de kommunicerande parterna. XML-webtjänster består av ett flertal specifikationer för meddelandeformat, beskrivning av tjänster och säkerhet. (Haas, 2004)

4.6 Dynamiska websidor och CGI

För att kunna skapa websidor som genereras dynamiskt, det vill säga genereras då en webläsare efterfrågar dem och där innehållet vanligen anpassas baserat på användarens fråga, krävs att någon form av programkod skapar sidan. *Common Gateway Interface* (CGI) är en standard för hur en webserver kan anropa ett externt program för att skapa en websida. En stor nackdel med CGI är att en ny process skapas och avslutas för varje sida som ska genereras, vilket förbrukar datorkraft i onödan. Det finns ett flertal olika tekniker som åtgärdar detta. (Lu och Feng, 1998)

En populär teknik är så kallade scriptspråk, till exempel *PHP: Hypertext Preprocessor* (PHP) och Microsoft *Active Server Pages* (ASP). Dessa tillåter att programkod läggs in med speciella direktiv i en HTML-sida och körs av ett tillägg till webservern då en webläsare hämtar sidan. (Grimes, 2002)

4.7 Single sign on och integrerad Windows-inloggning

Single sign on innebär att användaren inte ska behöva ange lösenord flera gånger då olika program eller webapplikationer körs. Det är till exempel vanligt att ha så kallade portaler som användare kan anpassa efter sina behov och länka in de webapplikationer som behövs och som efter att användaren loggat in sköter inloggningen till de andra webapplikationerna.

Integrerad Windows-inloggning är ett sätt att uppnå *single sign on* i Windows-baserade nätverk. Det är en funktion som finns i webservern Microsoft Internet Information Server och stöds av webläsarna Microsoft Internet Explorer och Mozilla Firefox. Denna funktion tillåter webläsaren att automatiskt logga in på en webplats med den användare som är inloggad på den dator som webläsaren körs på.

4.8 Relationsdatabaser

En databas är enligt Elmasri och Navathe (2004) en samling av relaterat data. Vanligen representerar en databas en del av verkligheten, och skapas och fylls med data för ett visst syfte.

Relationsdatabaser är en beprövad teknologi för att hantera stora datamängder där flera användare samtidigt kan läsa och skriva data. De är baserade på Codds relationsmodell, där en databas består av en samling av relationer. En relation kan ses som en tabell av värden, där alla element i en kolumn i tabellen har samma datatyp och där en rad i tabellen är data som hör ihop.

Några exempel på viktig funktionalitet som tillhandahålls av de flesta system för att hantera databaser är att möjliggöra effektiv åtkomst och uppdatering av stora datamängder, skydda data både från obehörig åtkomst och från systemkrascher, samt att se till att data uppfyller begränsningar som satts upp av den som skapade databasen, till exempel att dubbletter inte får förekomma i en relation.

Det finns ett flertal produkter, däribland Oracle Database och Microsoft SQL Server, för att hantera relationsdatabaser. Dessa är uppdelade i en klientdel och en serverdel, där serverdelen utför allt arbete och klientdelen används av andra processer för att kommunicera med servern.

Applikationen hämtar och uppdaterar data genom att skicka en förfrågan i form av en textsträng i *Structured Query Language* (SQL). Data från flera tabeller kan kombineras och resultatet kan filtreras, sorteras, grupperas och summeras.

Informationen i detta avsnitt är hämtad från Elmasri och Navathe (2004).

4.9 Microsoft .NET

Microsoft .NET är en relativt ny plattform för att utveckla Windows-baserade applikationer, webapplikationer samt XML-webtjänster. Tack vare ett gemensamt typsystem (*Common Type System, CTS*), och ett mellanspråk (*Intermediate Language, IL*), är .NET oberoende av vilket programmeringsspråk som

används. Källkod för ett visst programmeringsspråk översätts först till mellanspråk av en språkspecifik kompilator, och därefter till maskinkod då en metod körs för första gången. En del i .NET sköter om skräpsamling (*garbage collection*) och skyddar .NET-program som körs, så kallade *managed applications*.

Det finns ett omfattande klassbibliotek som bland annat hanterar grundläggande datatyper, samlingstyper, grafik, kommunikation, säkerhet, samt användargränssnitt både i form av Windows-baserade applikationer och webapplikationer.

Några klasser som var av speciellt intresse för detta projekt är *Active Data Objects .NET* (ADO.NET) som hanterar databasåtkomst samt *Active Server Pages .NET* (ASP.NET) som används för att bygga webapplikationer. ASP.NET bygger på tidigare versioner av ASP, men ger tillgång till all funktionalitet i .NET och innehåller många fördelar jämfört med ASP.

En annan nämnvärd del av .NET är *remoting*, vilken möjliggör anrop av objekt som ligger i andra processer eller över ett nätverk på ett sätt som är transparent för programmeraren. Det går att välja mellan olika transportkanaler och sätt att formatera nätverksmeddelandena, och dessutom går det att skapa egna kanaler och tjänster för att till exempel hantera kryptering.

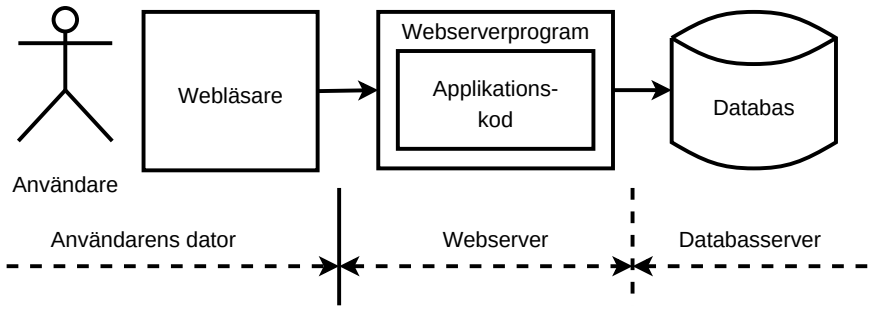
Informationen i detta avsnitt har hämtats från Fergal Grimes (2002) bok som ger en översikt av hela .NET.

Kapitel 5

Datahantering i webapplikation

En av grundfrågorna i en webapplikation som kommunicerar med en relationsdatabas är hur data hämtas och representeras internt samt hur uppdateringar sker. I detta kapitel redovisas den teoretiska fördjupningsstudie som gjorts inom området. Det finns flera olika tekniker med varierande egenskaper, och valet av metod påverkar i hög grad applikationens struktur.

5.1 Problemet



Figur 5.1: De viktiga delarna i en databasdriven webapplikation

Figur 5.1 visar de viktigaste komponenterna i en enkel databasdriven webapplikation. Användaren skickar via sin webläsare en begäran om att hämta en websida, som i detta fall genereras dynamiskt. Förfrågan tas emot av webservern, som vidarebefordrar den till den applikationsspecifika koden som kan vara i form av ett CGI-program, en ASP-sida eller liknande. Om den applikationsspecifika koden behöver data från databasen för att bygga sidan öppnar den först en koppling mot databasen. Därefter skickas en begäran att hämta data i form av en SQL-fråga och databasservern svarar med att skicka tillbaka data. Ändringar av data sker på liknande sätt genom att SQL-kommandon för uppdatering

av data skickas till databasen. Kommunikationen med databasservern sköts via ett klientbibliotek som tar hand om alla detaljer i kommunikationen. Vanligen används ett standardiserat gränssnitt, till exempel *Open DataBase Connectivity* (ODBC) eller ADO.NET, för att kommunicera med klientbiblioteket.

I enkla databasdrivna websidor är det vanligt att koden som genererar användargränssnittet direkt kommunicerar med databasen. Fördelarna med denna lösning är att det är enkelt att utveckla och nackdelarna är främst att det blir svårt att underhålla systemet samt att kod gärna dupliceras flera gånger i stället för att återanvändas. Gränssnittskoden är också starkt knuten till databaskoden, och det blir svårt att t.ex. byta databasserver. För att råda bot på nackdelarna kan olika former av flerlagerslösningar tillämpas.

5.2 Flerlagerslösningar

För att göra program lättare att underhålla och underlätta återanvändning av programkod är det vanligt att dela in programmet i flera logiska lager eller skikt, med väldefinierade gränssnitt mellan lagren. Ett lager på en viss nivå ska endast bero av lager på lägre nivå. Fowler (2003) använder följande lager:

Presentationslagret sköter kommunikationen med användaren. I en webapplikation kan detta sägas utgöras dels av användarens webläsare och dels av den del av koden på webservern som genererar HTML för användargränssnittet och hanterar kommandon från användaren.

Applikationslogiklagret , även kallat affärslogiklagret, hanterar regler för uppdatering av data.

Datalagret hanterar att lagra och hämta data i ett permanent lagringsutrymme. Detta lager kan till exempel utgöras av en relationsdatabas.

De logiska lagren kan sedan på olika sätt fördelas på fysiska lager, som anger hur funktionaliteten delats upp på olika datorer. Lhotka (2004) anser att antalet fysiska lager aldrig kan överstiga antalet logiska lager, och delar därför upp presentationslagret i två delar, presentation och användargränssnitt, eftersom användarens webläsare samt koden i webservern som genererar användargränssnittet normalt körs på olika datorer. Lhotka delar också upp datalagret i två lager, dataåtkomst och datalagring, bland annat eftersom detta möjliggör att datakällorna byts ut utan att logiklagret påverkas.

Skillnaden mellan logisk och fysisk lagerindelning är viktig. Den logiska uppdelningen påverkar hur lätt det är att underhålla systemet och återanvända programkod, medan den fysiska uppdelningen påverkar prestanda, skalbarhet, feltolerans och säkerhet. (Lhotka, 2004)

Om lagren läggs på olika datorer blir det viktigt att hålla nere antalet metodanrop. Detta beror på att ett anrop till en metod inom samma process är mycket snabbt, medan ett anrop mellan processer är långsammare och ett anrop mellan olika datorer är ännu långsammare. (Fowler, 2003)

Fowler rekommenderar att minimera antalet fysiska lager och i stället ha flera identiska system för att öka kapaciteten (klustring). Han anför följande:

> *First Law of Distributed Object Design: Don't distribute your objects!*
> (Fowler, 2003, p. 89)

Av säkerhetsskäl kan en fysisk uppdelning vara att föredra. En webserver som är ansluten till Internet utsätts ofta för intrångsförsök. Om applikationslogiken är åtskild från webservern och endast webservern är åtkomlig från Internet betyder det att även om någon lyckas ta sig in på webservern så kan denna bara göra vad applikationslogiklagret tillåter, vilket kan vara en avsevärd skillnad jämfört med att kunna skicka godtyckliga SQL-kommandon till databasen. Samma sak gäller i princip för webapplikationer som endast är tillgängliga i ett internt företagsnätverk, men Lhotka (2004) hävdar att det är mindre vanligt att det finns brandväggar som skyddar de interna systemen från attacker inifrån företaget. Om det inte finns någon brandvägg eller liknande som begränsar åtkomsten till till exempel databasservern så ger en indelning i fler lager inte någon ökad säkerhet.

5.3 Dataformat

Då data inte längre hämtas direkt från databasens klientgränssnitt utan ska gå via ett eller flera mellanlager blir det intressant att titta på i vilket format data ska representeras. Data från en relationsdatabasserver anländer vanligen rad för rad, där varje rad består av ett antal fält med primitiva datatyper. Varje rad har samma antal fält och samma typ på varje fält.

Det är inte nödvändigt att den interna representationen direkt svarar mot en tabell i databasen. Ett internt objekt kan bestå av data från flera tabeller i databasen och omvänt kan en tabell i databasen användas i flera olika interna objekt.

5.3.1 Primitiva datatyper

Med primitiva datatyper eller skalärer avses enstaka dataelement som skickas som parametrar eller returvärde till / från en metod eller funktion. Ett enkelt exempel är en funktion *skapa_resurs* som tar parametrarna *resursnamn* och *avdelning*, båda strängar, och returnerar *resursid* (ett heltal).

Några av fördelarna med detta alternativ är enligt Crocker et al. (2002) att det ger god prestanda och effektiv minnesanvändning då inget onödigt data behöver skickas, samt att det är enkelt att serialisera datat då det ska skickas över en fysisk lagergräns. Några nackdelar enligt dem är att det blir en stark koppling mellan anroparen och den anropade koden så om strukturen på datat som ska skickas ändras så måste funktionens parametrar ändras, samt att om flera poster av samma typ ska förändras måste ett anrop göras för varje post som ska förändras.

5.3.2 Dataset

Ett *dataset* är ett objekt som innehåller en eller flera tabeller med data och som har metoder för att stega igenom och manipulera datat. I .NET är detta klassen *DataSet* och tillhörande klasser för att representera enskilda tabeller, rader och kolumner. Flera utvecklingsmiljöer, bland annat den som används i detta projekt, har bra stöd för att arbeta med och visa data som lagras i dataset. Det finns vanligen också stöd för att plocka ut endast de rader som är förändrade då ett dataset som modifierats ska sparas till databasen.

5.3.3 Objekt

I objektorienterade språk är det möjligt att lagra information som objekt. Till exempel kan varje tabellrad som returneras från databasen lagras i ett eget objekt. En fördel med att använda objekt är att det går att kapsla in information samt ha metoder som arbetar på datat. Det är viktigt att skilja på objekt som skickas över i sin helhet, ibland kallade värdeobjekt (*value object*), och fjärrobjekt som ligger på ett ställe men vars metoder kan anropas från andra datorer.

5.3.4 XML

Extensible Markup Language (XML) är som tidigare nämnts ett standardiserat sätt att representera strukturerad information. Enligt Crocker et al. (2002) är fördelarna med att använda XML för att representera data att det är en öppen och flexibel standard som gör det lätt att utväxla information med andra system, det är lätt att representera en samling av poster, samt att det ger en lösare koppling mellan anroparen och den anropade funktionen. Nackdelarna är att XML normalt är en textsträng som måste analyseras av mottagaren, vilket tar tid, och att det inte är så minneseffektivt, vilket kan bli ett problem vid stora datamängder.

5.3.5 Övrigt

I ADO.NET, databasgränssnittet i .NET, finns ett koncept som kallas *data reader*. I korthet går detta ut på att då en SQL-fråga skickats till databasen för att hämta data så returneras ett *data reader*-objekt som används för att stega igenom de rader som returnerades och läsa ut värdena i kolumnerna i varje rad.

Crocker et al. (2002) nämner detta som ett möjligt alternativ med god prestanda för att returnera data från applikationslogiklagret då applikationslogiken ligger i samma fysiska lager som användargränssnittet, men lösningen är inte lämplig att använda då lagren är fysiskt åtskilda.

Alternativet ger god prestanda eftersom data inte behöver kopieras i onödan, men då lagren är fysiskt åtskilda blir prestandan sämre eftersom det krävs många metodanrop för att läsa ut allt data och metodanrop tar betydligt längre tid då anrop ska ske över en fysisk lagergräns.

5.4 Applikationslogik

Förutom datarepresentationen finns en annan central fråga - hur och var applikationslogiken ska implementeras. Fowler (2003) identifierar tre principiella mönster för detta:

Transaction script vilket i princip innebär att en procedur skapas för varje handling som användaren kan utföra. Proceduren tar indata från användargränssnittslagret, kontrollerar datat, manipulerar databasen och skickar ett svar till användargränssnittslagret. Funktionalitet som är gemensam för flera *transaction scripts* kan brytas ut till hjälpprocedurer. De huvudsakliga fördelarna enligt Fowler är att det är en enkel modell som är lätt att förstå och fungerar bra ihop med enkla datakällor, samt att det är

tydligt var transaktioner börjar och slutar. Nackdelarna är att det är svårt att undvika att duplicera programkod mellan olika *transaction scripts*.

Table Module är i princip ett objekt som kapslar in ett dataset (se avsnitt 5.3.2) och har metoder för att manipulera innehållet på olika sätt. Fördelarna med denna metod är att det ger mer struktur än *transaction scripts* samt att många utvecklingsmiljöer har bra stöd för att arbeta med dataset och bygga användargränssnitt kring dem.

Domain Model är en helt objektorienterad lösning som innebär att en modell av de dataobjekt som förekommer i applikationen skapas. Klasserna i modellen behöver inte alls överensstämma direkt mot tabeller i databasen. Klasserna har metoder för att manipulera objekten på olika sätt. Fördelarna med denna lösning är att även komplex applikationslogik kan hanteras på ett strukturerat sätt. Nackdelarna är att inlärningströskeln är hög samt att ett relativt komplicerat lager med programkod krävs för att mappa objekten i modellen mot databasen.

Lhotka (2004) beskriver en objektorienterad modell lämpad för distribuerade system där data och applikationslogik kombineras i ett objekt som kan skickas mellan lager. En kopia av hela objektet skickas vilket gör att användargränssnittslagret kan göra många små anrop utan att prestanda försämras. Lhotka kapslar också in större delen av dataåtkomstlagret i objekten. En möjlig nackdel med Lhotkas lösning är att den är hårt knuten till specifik funktionalitet i Microsoft .NET.

Ett annat sätt att implementera applikationslogik är att göra det direkt i databasen med hjälp av vyer och lagrade procedurer (*stored procedures*) genom vilka all läsning från och skrivning till databasen måste gå. Fördelarna med detta är att alla system som är kopplade till samma databas utnyttjar samma applikationslogik. En nackdel med denna metod är att lagrade procedurer ofta inte är kompatibla mellan produkter från olika tillverkare av databasservrar, så dessa kan behöva skrivas om för varje databasserverprodukt som ska stödjas.

5.5 Hantering av samtidig åtkomst

Ett problem som alltid uppkommer då flera användare samtidigt kan modifiera data är vad som ska hända då en användare hämtar data, bearbetar det och sparar det senare och en annan användare under tiden hunnit ändra datat.

Då data hämtas, bearbetas och modifieras inom samma begäran från en användare kan problemet enkelt lösas med hjälp av vanliga transaktioner i relationsdatabasen.

Om datat däremot har hämtats i en begäran och därefter ska modifieras i en annan begäran blir det svårare, eftersom det ej är lämpligt att hålla en koppling till databasservern öppen för varje användare. Det finns minst tre olika sätt att hantera problemet:

1. Strunta i problemet och låta den senast skrivna versionen av datat vara den som gäller. Den tidigare uppdateringen går förlorad utan att någon användare får veta det. Fördelen med denna lösning är att den är enkel att implementera och nackdelen är att ändringar går förlorade utan att användarna blir medvetna om det.

2. Använda pessimistisk låsning. Detta innebär att då en användare hämtar data markeras det som låst av den användaren. Andra användare som vill modifiera samma post får vänta. Fördelen är att den som låst data vet att ingen annan kan ha ändrat det under tiden, så det kommer alltid att gå att spara. Nackdelen är att ingen annan användare under tiden kan ändra datat även om den första användaren sedan väljer att inte spara sina ändringar. I weblösningar går det dessutom inte att veta om en användare stängt ner sin webläsare utan att logga ut, vilket gör att data kan hållas låst och otillgängligt onödigt länge.

3. Använda optimistisk låsning, vilket innebär att då data hämtas så sparas originalversionen, ett versionsnummer, eller en tidsstämpel, och då det modifierade datat ska sparas så kontrolleras att data i databasen inte ändrats under tiden. Fördelarna är att data inte hålls låst under längre tidsperioder vilket ger större möjligheter för andra användare att arbeta med systemet under tiden. Nackdelen med denna lösning är att användaren inte kan vara säker på att hennes ändringar av datat går att spara, så om många användare kan tänkas vilja ändra samma data samtidigt kan pessimistisk låsning vara att föredra. (Fowler, 2003)

Applikationen som beskrivs i denna rapport skulle stödja samtidig åtkomst till flera databaser, men en begäran från en enskild användare involverar endast en databas vilket gjorde att distribuerade transaktioner mot flera relationsdatabaser ej var nödvändiga. Dessutom bedömdes det att det första alternativet för att hantera samtidiga uppdateringar, det vill säga att helt enkelt låta den senast skrivna versionen vara den som gäller, var tillräckligt bra för den första versionen av designen.

5.6 Övrigt

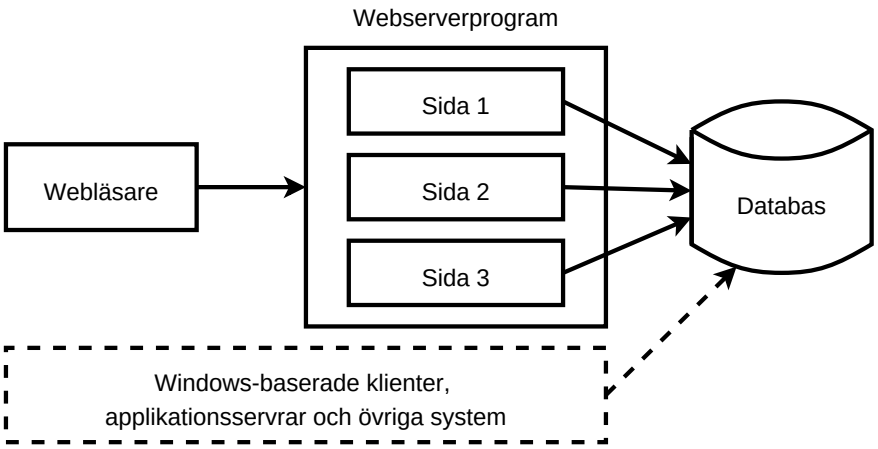
Det finns ett klart problem med att upprätta och stänga en koppling till databasen för varje anrop till webservern, och det är att det tar tid. För att öka prestandan är det vanligt att ha en “pool” – ett förråd – med databaskopplingar som är aktiva under en längre tid och används då de behövs. (Ritter, 1998)

Kapitel 6

Designval

Detta kapitel beskriver några viktiga val som låg till grund för den slutliga designen som beskrivs i nästa kapitel.

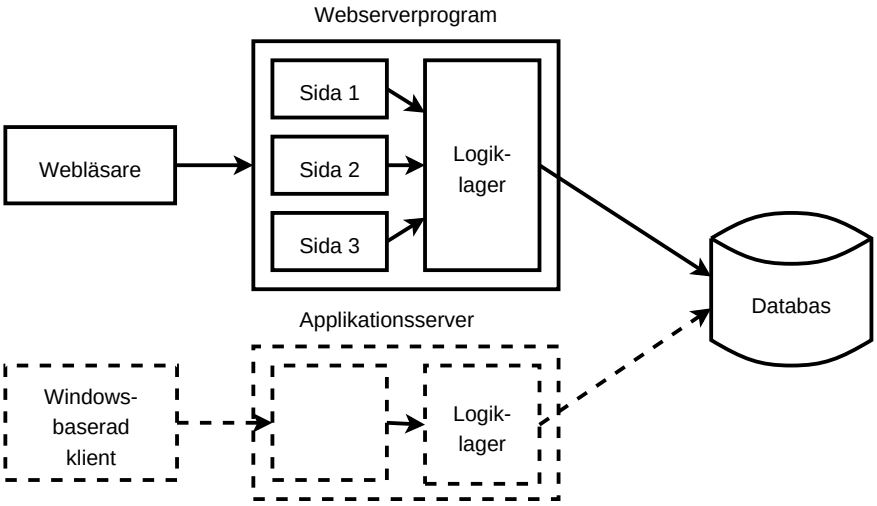
6.1 Lagerindelning



Figur 6.1: Webapplikation med direkt åtkomst till databasen utan separat logiklager

Ett av de viktigaste valen var vilka fysiska lagerindelningar som skulle tillåtas. Det finns en fysisk lagergräns mellan presentationslagret (användarens webläsare) och användargränssnittslagret som körs på webservern. Dessutom var det med de databasserverprodukter som skulle användas för weblösningen inga problem att tillåta att datalagringslagret var fysiskt åtskilt från applikationslogiken. Det viktiga valet var om det skulle vara möjligt att separera användargränssnittslagret fysiskt från applikationslogiklagret.

Den stora fördelen med att tillåta en separation är att det möjliggör att applikationslogiken återanvänds även för den Windows-baserade applikationen.



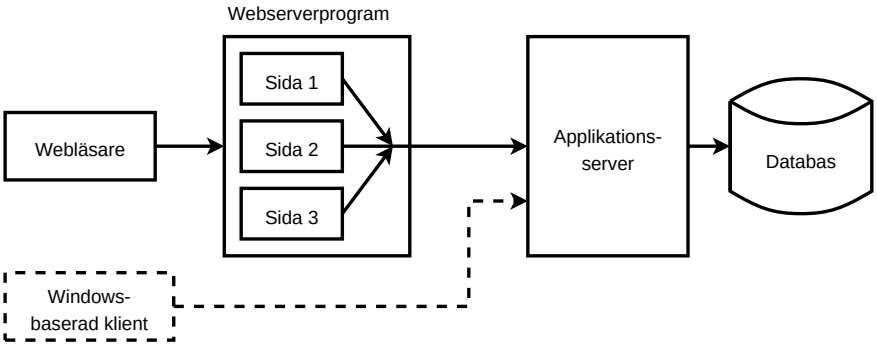
Figur 6.2: Webapplikation med separat logiklager som används i webserver och eventuella andra servrar

För själva webapplikationen finns också en liten fördel då säkerheten kan höjas något om en brandvägg sätts mellan webservern och applikationsservern.

Nackdelarna med att tillåta en separation är att lösningen blir mer komplex. En del av datarepresentationerna som beskrevs i kapitel 5 blir mindre lämpliga, och gränssnittet till applikationlogiklagret måste anpassas så det lämpar sig för fjärranrop, genom att se till att så få anrop som möjligt behövs. Detta kan ge sämre prestanda då de uteslutna datarepresentationerna kan ge bättre prestanda. Dessutom kan hanteringen av inloggning och säkerhet bli mer komplicerad.

Figur 6.1 visar den allra enklaste lösningen, där varje websida direkt kommunicerar med databasen utan något egentligt logiklager. Denna lösning ger ingen delning av kod alls med till exempel en Windows-baserad klient.

Figur 6.2 visar en lösning med ett applikationslogiklager som inte är fysiskt åtskilt från användargränssnittslagret, men som utgör en separat modul som



Figur 6.3: Webapplikation med separat applikationsserver

kan återanvändas i en applikationsserver för en Windows-baserad klient.

Figur 6.3 visar en lösning där det finns en applikationsserver som används både av webapplikationen och den Windows-baserade klienten. I denna lösning är presentationslogiken fysiskt åtskild från applikationslogiken.

I alla tre figurer indikerar streckade linjer funktionalitet som inte tillhör själva webapplikationen utan istället tillhör andra system.

Den lösning som slutligen valdes är ett mellanting mellan det andra och det tredje alternativet. Gränssnittet till applikationslogiken anpassades för att kunna användas via fjärranrop, men samtidigt möjliggjordes att koden som implementerar applikationslogiken laddas in i samma process som webgränssnittet för att slippa kostnaden för anrop mellan flera processer och för att förenkla att flera webservrar används. Denna design beskrivs utförligare i kapitel 7.

6.2 Datarepresentation

Som föregående kapitel visade så finns det flera alternativ för hur data kan representeras. Den lösning som valdes för lagerindelningen gjorde det nödvändigt att välja en datarepresentation som lämpade sig för gränssnittet till applikationslogiklagret. För det allmänna fallet då en eller flera tabeller med data ska skickas över valdes det så kallade Dataset-objektet i .NET eftersom det finns bra stöd för det. För vissa fall, till exempel de grafiska kartorna, valdes istället att skapa egna objekt som skickas över med hjälp av serialisering.

Valet som gjorts här hindrar dock inte att en annan datarepresentation används internt i applikationslogiklagret.

6.3 Säkerhet

Säkerhet är mycket viktig i denna typ av applikation då den kan behandla mycket känslig information såsom ekonomiska resultat. I begreppet säkerhet ingår både skydd mot avlyssning och förvanskning av datatrafik samt kontroll av användarens identitet och rättigheter.

6.3.1 Skydd av HTTP-trafik

Trafiken mellan användarens webläsare och webservern använder HTTP-protokollet. I sitt grundutförande skickas all information helt oskyddad, vilket till exempel kan inkludera användarnamn och lösenord som skickas från ett inloggningsformulär. En person som sitter på samma nätverk kan avlyssna denna information. Genom att använda HTTP över *Secure Sockets Layer* (SSL) förhindras avlyssning, till priset av högre belastning på webservern då kryptering tar en del tid. En mellanlösning är att enbart använda SSL för att skydda inloggningssidor vilket förhindrar att användarnamn och lösenord stjäls men inte hindrar att till exempel ekonomiska resultat kommer i orätta händer.

Det viktigaste att ta hänsyn till under designarbetet var att undvika lösningar som förhindrade att SSL används. Att använda SSL för all trafik är enkelt, och kan göras enbart genom att ändra webserverns konfiguration. Att tillåta att endast inloggningssidan skyddas skulle kräva logik i webapplikationen. Med tanke på den känsliga information som webapplikationen skulle behandla valdes att använda SSL för all trafik mellan webläsaren och webservern.

6.3.2 Skydd av trafik mellan servrar

Om webservern och databasservern inte körs på samma dator kan det vara önskvärt att skydda datatrafiken mellan dessa. Om det dessutom finns en separat applikationsserver blir det ytterligare en kommunikationslänk att skydda. Ett sätt att lösa detta är att se till att servrarna enbart kommunicerar över ett skyddat nätverk som inte är åtkomligt för vanliga användare och där till exempel direkt åtkomst till databasservern skyddas av en brandvägg. Ett annat sätt är att skydda all trafik mellan datorerna med hjälp av *Internet Protocol Security* (IPSec), vilket ger möjlighet till att kontrollera identiteten hos den dator som sänt data samt att skydda data med hjälp av kryptering. Ytterligare ett sätt är att kryptera varje förbindelse, till exempel med SSL. (Meier et al., 2002)

6.3.3 Intern säkerhetsmodell

Meier et al. (2002) identifierar två vanliga modeller för säkerheten i ett distribuerat system. Den första är *trusted subsystem*, vilken innebär att då ett lager ska kommunicera med ett fysiskt åtskilt lager på lägre nivå, till exempel då en applikationsserver ska kommunicera med databasservern, så används en fast identitet. Detta innebär att lagret på lägre nivå litar på att det högre lagret kontrollerar användarens rättigheter. Den andra modellen är *impersonation/delegation*, vilken innebär att användarens identitet används vid kommunikation med lagret på underliggande nivå. Meier et al. (2002) förklarar detta som om det alltid är operativsystemet (Windows) som sköter detta, men jag tycker att det bör gå bra att göra detta på applikationsnivå också.

En viktig fördel med *trusted subsystem* är att skalbarheten blir bättre då uppkopplingar kan poolas, det vill säga återanvändas för olika klienter. En fördel med *impersonation/delegation* är att säkerheten blir bättre.

För designen valdes *trusted subsystem* för kommunikationen mellan applikationslogiken och databasservrarna.

För kommunikationen mellan användargränssnittet och applikationslogiklagret behöver inget val göras om dessa lager inte är fysiskt åtskilda, vilket var normalfallet för webapplikationen i den slutliga designen. Även i det fall de är åtskilda är *trusted subsystem* möjligt för webapplikationen, om åtkomsten till applikationslogiklagret begränsas så bara webservrarna kan kommunicera med det. Däremot krävs ett val då användargränssnittet körs på en användares dator, till exempel i form av en Windows-baserad klient. I detta fall är *trusted subsystem* olämpligt som säkerhetsmodell, då ett program som användaren har kontroll över normalt inte kan ses som pålitligt. I detta fall är *impersonation/delegation* bättre för säkerheten mellan användargränssnittet och applikationslogiken.

Frågeställningen kompliceras ytterligare om användaren också ska kunna komma åt till exempel en rapportserver, eller använda en aktiv webklient (se avsnitt 7.4, utan att behöva logga in igen.

6.3.4 Övrigt

För ett säkert system krävs givetvis också alla normala säkerhetsåtgärder som att hålla operativsystem, webserver etc. uppdaterade med alla tillgängliga säkerhetsuppdateringar, stänga av tjänster som inte behövs, begränsa åtkomst genom t.ex. brandväggar osv.

Vid implementationsarbetet finns det också saker att tänka på, till exempel att aldrig lita på data som kommer från klienten, för att förhindra att användare kan kringgå säkerhetssystemet.

6.4 Skalbarhet

Ett skalbart system bibehåller effektiviteten då antalet användare och resurser ökar kraftigt. (Coulouris et al., 2001) Om antalet samtidiga användare till webapplikationen blir tillräckligt stort kommer en enda webserver till sist inte att ha tillräcklig prestanda för att klara av dem. Det är då viktigt att lösningen är skalbar så det går att lägga till fler webservrar utan att alltför stora flaskhalsar uppstår.

En sak som kan ställa till med problem med skalbarheten är tillståndsinformation (*state*) som behöver sparas i applikationen. Ett exempel på sådan information är information om vilka användare som är inloggade. Ett annat exempel är själva datat som lagras i databasen. Med tillräckligt stort antal användare kommer en ensam databasserver inte att orka med. I den lösning som presenteras i denna rapport förutsätts att detta kan lösas med hjälp av en klusterlösning för databasen som hanterar detta utan att webapplikationen behöver veta att databasen körs på mer än en server.

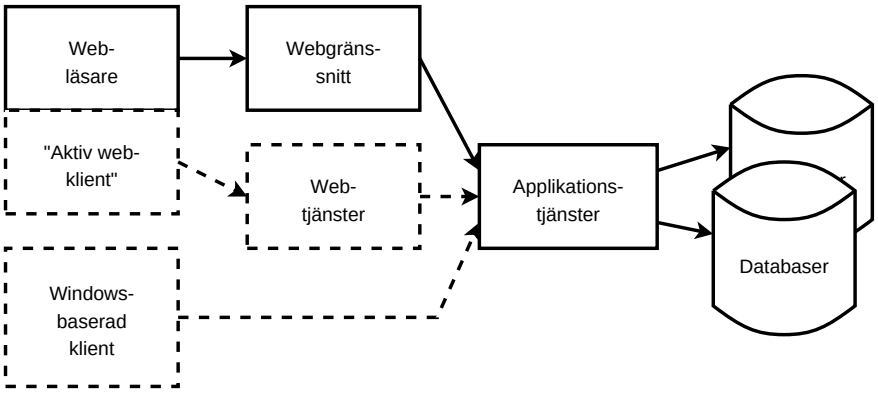
Det som främst var nödvändigt vid designarbetet var att undvika lösningar som förhindrade att flera webservrar användes.

Kapitel 7

Resultat

I detta kapitel redovisas den slutliga designen samt den prototypimplementation som gjorts.

7.1 Översikt



Figur 7.1: Huvuddelarna i den slutliga designen

Figur 7.1 visar huvuddelarna i systemet som designats. Detta är bara en principskiss som visar logiska programkomponenter och innebär inte att funktionaliteten behöver vara fysiskt uppdelad på detta vis. Delarna med streckad ram har behandlats mindre ingående i examensarbetet.

Delarna har följande funktion:

Webläsaren är en vanlig webläsare som körs på användarens dator.

Webgränssnittet körs på webservern och är ansvarigt för att generera HTML-sidor som svar på anrop från webläsaren.

Applikationstjänsterna ansvarar för applikationslogiken.

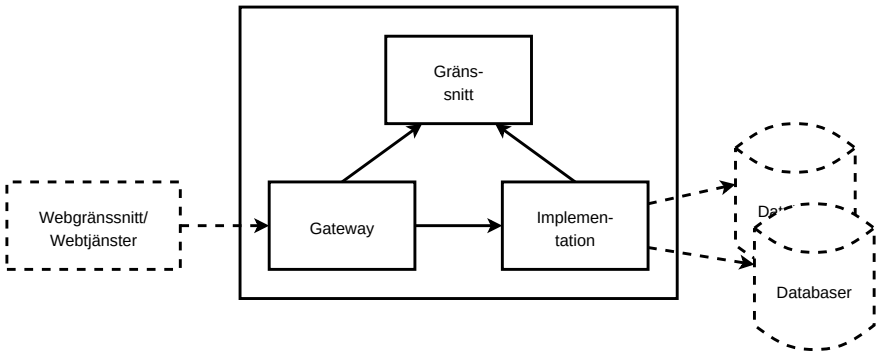
Databaserna lagrar datat.

Aktiva webklienter är programkod som körs av användarens webläsare för eventuell funktionalitet som inte kan implementeras med HTML och JavaScript, till exempel för att manipulera grafiska kartor.

Webtjänsterna är publika gränssnitt avsedda att användas från andra system för integration.

Den Windows-baserade klienten är ett vanligt program som körs på användarens dator.

7.2 Applikationstjänster



Figur 7.2: Beståndsdelarna i applikationstjänsterna

Applikationstjänsterna implementerar applikationslogiken och databasåtkomsten. Uppbyggnaden visas i figur 7.2. De streckade delarna tillhör inte själva applikationstjänsterna. Delarna har följande funktion:

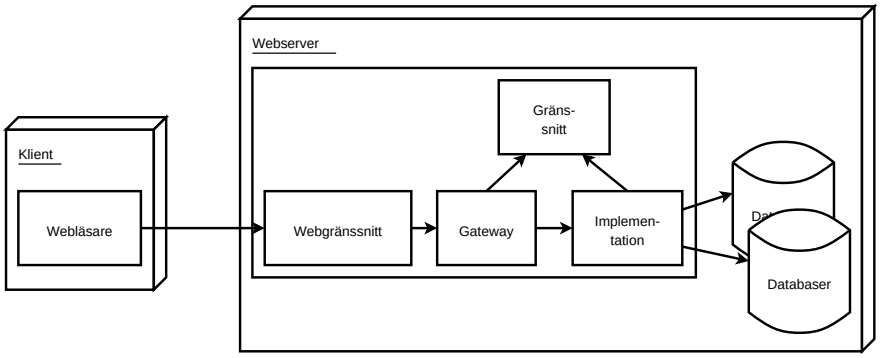
Gränssnittsdelen definierar gränssnitten till alla tjänster.

Implementationsdelen implementerar tjänsterna.

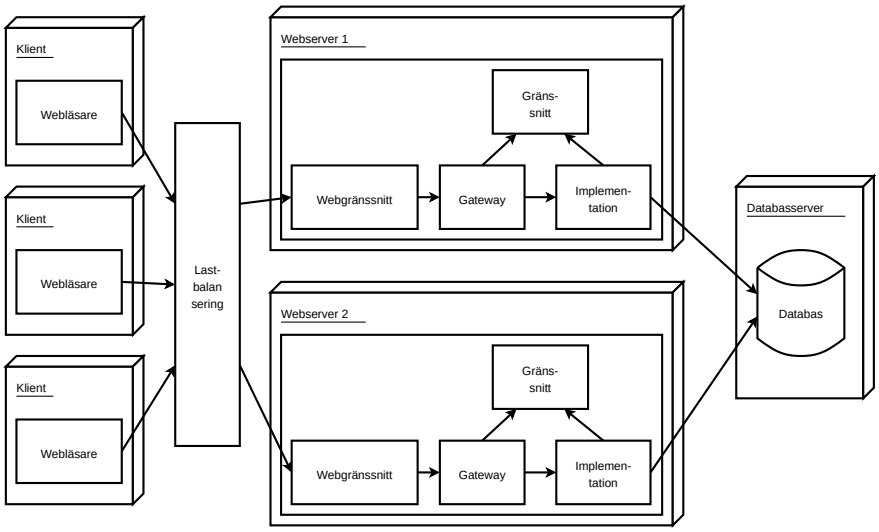
Gatewaydelen används för att få en instans av en tjänst. Beroende på hur systemet konfigurerats kan implementationsdelen antingen laddas in direkt eller anropas via *remoting*, vilket möjliggör att den ligger på en annan dator.

Denna uppdelning möjliggör dels att alla tre delar körs i samma process, och dels att de kan delas upp. För webapplikationen är det lämpligt att köra allt i samma process för bra prestanda och skalbarhet, vilket visas i figur 7.3. Figur 7.4 visar hur ännu en webserver kan läggas till.

För en Windows-baserad klient är det däremot lämpligt att dela upp det så implementationen av tjänsterna körs på en applikationsserver, vilket visas i figur 7.5. Några av anledningarna är att det tillåter att applikationsservern har ett mindre antal uppkopplingar till databasen än antalet klienter (poolning), samt att säkerheten blir bättre jämfört med om varje klientdator tillåts koppla upp sig direkt mot databasen.



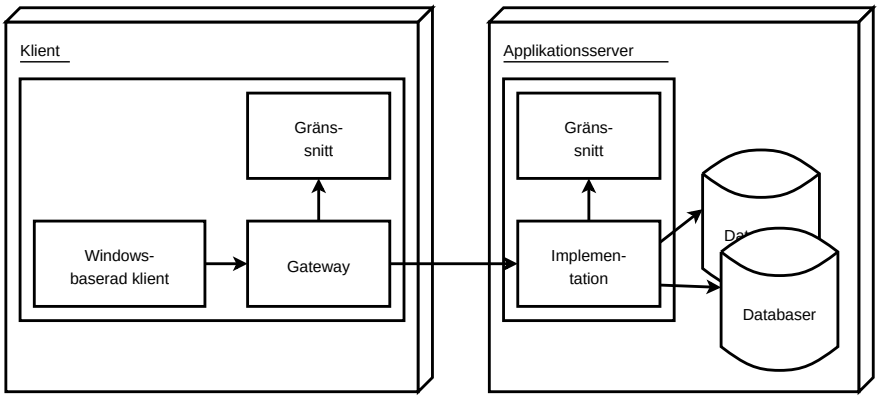
Figur 7.3: Applikationstjänster som körs i samma process som webgränssnittet



Figur 7.4: Lastbalansering mellan flera webservrar

De tjänster som finns har ett väldefinierat gränssnitt. Då webgränssnittet behöver använda en tjänst anropar den gatewayen som returnerar en referens till ett objekt som implementerar det gränssnittet. Beroende på hur systemet konfigurerats så skapar gatewayen antingen en instans av objektet som implementerar gränssnittet, eller så används ett fjärrobjekt via remoting. Tack vare detta blir det transparent för webgränssnittet var implementationen finns.

Ett alternativ till denna design hade varit att implementera applikationstjänsterna som XML-webtjänster. Alternativet att alltid köra applikationstjänsterna som XML-webtjänster valdes bort eftersom det skulle innebära att de alltid skulle köras i en separat process, vilket skulle leda till sämre prestanda. Av skalbarhetsskäl skulle det inte gå att köra dessa webtjänster på en enda server, utan om flera webservrar används i en klusterlösning så måste webtjänsterna sannolikt också klustras, enklast genom att lägga dem på samma maskiner som webservrarna. Dessutom bedömdes det göra installationen mer komplicerad.



Figur 7.5: Windows-baserad klient och applikationsserver

Däremot skulle XML-webtjänster kunna vara ett alternativ istället för remoting om applikationstjänsternas gränssnitt anpassas så de kan användas både som XML-webtjänster och laddas in direkt. Fördelen skulle då vara att andra system direkt skulle kunna ansluta sig till applikationstjänsterna istället för att gå via en separat webtjänst vilket skulle ge en komponent mindre i systemet. Nackdelarna är att prestandan kan bli sämre om tekniker som lämpar sig för remoting men inte för XML-webtjänster används. Dessutom är det av kompatibilitetsskäl svårare att ändra ett publikt gränssnitt som används av andra system.

Det finns även flera andra produkter som kan användas istället för remoting, till exempel RemObjects SDK.

7.2.1 Gränssnitt

Gränssnittet till tjänsterna definierar vilka operationer de kan utföra. Gränssnitten kan delas in i dels ett eller flera generella som hanterar saker gemensamma för hela systemet, till exempel inloggning och övrig säkerhet, samt ett antal som är specifika för olika delar av systemet.

Gemensamt för alla metoder i gränssnitten är att de ska lämpa sig för fjärranrop, det vill säga få metodanrop ska krävas för att kunna hantera en begäran från användaren.

För att uppnå bra skalbarhet är det också viktigt att inte spara en massa information för varje användare i applikationslogiklagret. Däremot går det bra att information som ofta behövs men sällan förändras mellanlagras i applikationslogiklagret i stället för att hämtas från databasen varje gång.

7.2.2 Tjänsteimplementation

För att implementera tjänsterna behövs ett antal stödjande funktioner. Några sådana beskrivs översiktligt här.

En viktig del är databasåtkomsten. Det behövs funktionalitet för att upprätta en koppling till en databas, hämta data, samt utföra uppdateringar. Microsoft .NET har klassbiblioteket ADO.NET, och Borland Delphi har ett bibliotek som

heter Borland Data Provider. För att undvika att låsa fast sig vid ett specifikt sådant bibliotek kan det vara lämpligt att skapa ett eget databaslager som i grunden använder något av dessa bibliotek men som har ett väldefinierat gränssitt som kan tillhandahålla utökad funktionalitet, såsom förbättrad hantering av parametrar i SQL-frågor.

En annan del är hantering av systemkonfiguration, till exempel information om databaskopplingar.

7.2.3 Säkerhet

Säkerhetssystemet har inte designats i detalj av tidsskäl, men i grova drag planerades en lösning med krypterade biljetter som kan begäras från en säkerhetstjänst i applikationslogiklagret. Det är viktigt att tänka på att lösningen ska vara skalbar och klara av att flera webservrar används. Dessutom får inte en biljett vara giltig hur länge som helst. Det bästa skulle vara om det gick att använda ett färdigt säkerhetsprotokoll som klarar detta.

7.3 Webgränssnitt

För webgränssnittet valdes vanlig ASP.NET, för att enkelt kunna använda designstödet i utvecklingsverktyget och för att kunna använda det stora utbudet av tredjepartskomponenter.

Det behövs även här några stödjande funktioner, bland annat för hantering av konfiguration.

7.4 Aktiva webklienter

Microsoft .NET innehåller funktionalitet som möjliggör att grafiska komponenter skrivna i .NET bäddas in i en websida och körs på klienten, på liknande sätt som Java-applets. Detta gör det möjligt att göra mer interaktiva lösningar än vad som kan åstadkommas med ren HTML, CSS och JavaScript, till exempel ritfunktioner. En klar begränsning med denna typ av lösning är att den kräver att Microsoft .NET Framework är installerat på klientdatorerna samt att den i dagsläget bara verkar fungera med webläsaren Microsoft Internet Explorer.

Normalt körs .NET-kod som används på detta sätt med kraftigt begränsade rättigheter. Detta visade sig dock otillräckligt i den enkla prototyp som gjordes för att testa detta, bland annat eftersom den använde .NET Remoting för att hämta data. Detta innebar att ett tillägg till säkerhetsinställningarna på klientdatorerna var nödvändigt för att ge prototypen mer rättigheter. Inställningsmöjligheterna för säkerheten är dock mycket flexibla, och gör det möjligt att bara ge extra rättigheter till kod som hämtas från en specifik server eller som har en specifik digital signatur. Behovet av inställningar på klientdatorerna var ändå en klar nackdel för denna typ av lösning.

Sammantaget gjorde dessa begränsningar att denna teknik bedömdes som användbar för begränsade delar av lösningen där mycket interaktion krävdes, men att större delen av lösningen skulle göras utan denna teknik.

7.5 Prototyp

För att se hur designen med tjänster och dataåtkomst fungerade i praktiken utvecklades en mycket enkel prototyp i Delphi 8. Då det handlade om en prototyp togs ett flertal genvägar, bland annat utelämnades i princip all felhantering och säkerhetshantering. Mycket enkla klientprogram, både en Windows-baserad klient och en ASP.NET-webapplikation gjordes också, som i princip bara hämtade lite data för att visa att det fungerade. Dessutom testades aktiva webklienter med .NET.

Prototypen visade att den tänkta lösningen rent principiellt fungerade, men prototypen var inte på något sätt en användbar produkt utan bara ett stöd för designarbetet. För att verkligen kunna utvärdera lösningen med applikationstjänster skulle prestandatester och jämförelser med andra lösningar behövas. Dessutom visade prototyparbetet att det finns ett stort antal praktiska detaljer att tänka på, till exempel säkerhetsinställningarna för aktiva webklienter om den tekniken ska användas.

Kapitel 8

Slutsatser

Detta kapitel sammanfattar arbetet och redovisar vilka slutsatser som kan dras, vilka begränsningar som finns i arbetet som gjorts samt vilka möjligheter till vidare arbete som finns.

8.1 Sammanfattning och slutsatser

Rapporten har beskrivit arbetet med att ta fram ett arkitekturförslag för en webapplikation där höga krav på säkerhet och skalbarhet ställs.

En slutsats som kan dras från arbetet är att det bör vara möjligt att implementera den föreslagna designen men att den flexibilitet som designen ger också gör lösningen betydligt mer komplex jämfört med en enkel webapplikation i ASP.NET som kommunicerar direkt med databasservern, använder ASP.NETs hantering av inloggning och så vidare. Denna komplexitet innebär sannolikt en längre utvecklingstid för första versionen men kan underlätta framtida underhåll om den innebär att en betydande mängd programkod kan återanvändas för både webgränssnittet och den Windows-baserade klienten.

En annan slutsats är att det finns otroligt mycket att tänka på vid designen av ett stort system. Många av designvalen påverkar varandra, och det finns konflikter till exempel mellan krav på skalbarhet och säkerhet. Till exempel kan ett system med fler fysiska lager göras säkrare mot intrång men det blir mer komplicerat att skala upp systemet och den interna säkerhetsmodellen blir mer komplicerad. Det blev också uppenbart att det fanns ett stort antal alternativa lösningar för många delar av designen.

8.2 Begränsningar

För att arbetsmängden skulle bli rimlig var det nödvändigt med ett flertal avgränsningar, vilket givetvis innebär att alla möjligheter inte har utretts. En allvarlig begränsning är att säkerhetsdelen i designen inte blev klar, då det är en viktigt del.

8.3 Vidare arbete

Den viktigaste vidareutvecklingen av arbetet är att fortsätta med säkerhetsdelen. Designen kan också i övrigt vidareutvecklas, dels på bredden genom att fler alternativ för till exempel datahantering utreds, och dels på djupet, genom att designen och prototypen förfinas till en mer detaljerad nivå. Några av de saker som bör utredas mer är till exempel en mer detaljerad design för användargränssnittslagret samt för hur applikationslogiken ska byggas upp. Dessutom skulle en mer omfattande prototyp vara önskvärd att ha för att bättre se hur designen fungerar i praktiken.

Kapitel 9

Tack

Det har varit mycket intressant och lärorikt att utföra detta examensarbete. Följande personer har varit till stor hjälp vid arbetets utförande.

Först vill jag tacka mina handledare, Oscar Appelgren och Lars Näslund. Oscar var min interna handledare vid Umeå Universitet och har varit till stor hjälp framför allt med rapporten. Lars Näslund var min externa handledare vid Prodacapo AB och gav mig möjligheten att utföra examensarbetet och ett flertal värdefulla diskussioner och synpunkter på arbetet.

Därefter vill jag tacka alla andra medarbetare vid Prodacapo AB, och ge ett speciellt tack till Mikael Olofsson, Niklas Hörnfeldt, Fredrik Vestin och Lars Österlund för värdefulla tips och diskussioner rörande designen och utvecklingsmiljön.

Slutligen vill jag tacka Dan Ögren för att ha korrekturläst rapporten.

Sakregister

Litteraturförteckning

Bos, B. (2001). XML in 10 points. Websida, besökt 2004-12-06. <http://www.w3.org/XML/1999/XML-in-10-points>.

Coulouris, G., Dollimore, J., och Kindberg, T. (2001). *Distributed Systems: Concepts and Design*. Addison-Wesley, tredje utgåvan.

Crocker, A., Olsen, A., och Jezierski, E. (2002). Designing Data Tier Components and Passing Data Through Tiers. Websida, besökt 2004-09-28. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp>.

Elmasri, R. och Navathe, S. B. (2004). *Fundamentals of Database Systems*. Addison-Wesley, fjärde utgåvan.

Fielding, R., J.Gettys, Mogul, J., Frystyk, H., Masinter, L., Leach, P., och Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. Internet Request for Comments 2616. <ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt>, besökt 2004-09-20.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley, första utgåvan.

Grimes, F. (2002). *Microsoft .NET for Programmers*. Manning Publications Co., Greenwich, CT, USA.

Haas, H. (2004). Web Services Activity Statement. Websida, besökt 2004-12-06. <http://www.w3.org/2002/ws/Activity>.

Kristol, D. och Montulli, L. (2000). HTTP State Management Mechanism. Internet Request for Comments 2965. <ftp://ftp.rfc-editor.org/in-notes/rfc2965.txt>, besökt 2004-09-20.

Lhotka, R. (2004). *Expert C# Business Objects*. Apress, första utgåvan.

Lu, H. och Feng, L. (1998). Integrating database and world wide web technologies. *World Wide Web*, 1(2):73–86.

Meier, J., Mackman, A., Dunner, M., och Vasireddy, S. (2002). *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication*. Microsoft Corporation. Tillgänglig på Internet, <http://msdn.microsoft.com/library/en-us/dnnetsec/html/-secnetlpmsdn.asp>.

Pfaffenberger, B. och Gutzman, A. D. (1998). *HTML 4 Bible*. IDG Books Worldwide.

Raggett, D., Hors, A. L., och Jacobs, I. (1999). HTML 4.01 Specification. Teknisk rapport, World Wide Web Consortium. Tillgänglig på Internet, <http://www.w3.org/TR/html401/>.

Ritter, D. (1998). The middleware muddle. *SIGMOD Rec.*, 27(4):86–93.