Technology and Business
University College Nordjylland
Sofiendalsvej 60
9200 Aalborg SV
http://www.ucn.dk

Datamatiker - 3 Semester

Flight Routing

Gruppe 3
Kim Bach
Lasse Dahl Jensen
Jakob Korsgaard
Nick D Pedersen

Afleveringsdato: 04 Juni 2015

Titel:
3. Semesterprojekt
Flight Routing

Klasse:
dmaa0214

Fag:
CDS
SDP
Web

Semester:
3. Semester

Projektgruppe:
Gruppe 3

Medlemmer:
Kim Bach
Lasse Dahl Jensen
Jakob Korsgaard
Nick D Pedersen

Hovedvejleder:
Simon Kongshøj

Vejledere:
Simon Kongshøj
Gianna Bellè

Sidetal:
15 sider af 2400 tegn + bilag

Afsluttet den:
04-06-2015

Technology and Business
University College Nordjylland
Sofiendalsvej 60
9200 Aalborg SV
http://www.ucn.dk

Synopsis:

Denne rapport tager udgangspunkt i casen Flight Routing. Der er igennem designfasen udarbejdet relevante diagrammer, herunder domænemodel og SSD, ud fra den udleverede use case.

Endvidere er der identificerede teknologier og løsninger, heriblandt WCF og Entity Framework, samt Dijkstra's algoritme som hver især danner grundlag for implementering af et client / server system.

Forord

Denne rapport er udarbejdet af projektgruppe 3 på 3. semester på holdet dmaa0214 på datamatiker uddannelsen under University College Nordjylland. Dette foregik i perioden februar til juni 2015.

Rapporten danner baggrund for vores projektarbejde i forhold til client-server systemet, der blev udviklet i samme periode og rapporten giver også et overblik over hvilke design- og systemmæssige valg vi har traf undervejs.

Vores målgruppe er systemarkitekter, systemingeniør og systemudviklere. Rapporten vil fungerer som en skabelon for udviklingen af systemet.

Tak til Simon Kongshøj for at fungere som vores hovedvejleder igennem perioden samt også som underviser i faget CDS (Computer Network and Distributed Systems). Også en tak til undervisere Gianna Bellè i faget SDP (Software Architecture and Distributed Programming) og Michael Holm Andersen i faget Web (Web Development).

Gruppe 3, dmaa0214, University College Nordjylland, 2015

Indhold

Indledning

I dette projekt har vi valgt at arbejde med casen "Flight Routing". Fokus for vores projekt er derfor udviklingen af et distribueret system, som kan bruges til booking af flyrejser. IT-systemet vil være en klient / server løsning, det er i stand til ud fra en algoritme at beregne den billigst mulige rejse fra en lufthavn til en anden lufthavn. Fokus i projektet vil derfor være på den økonomiske side og ikke nødvendigvis den hurtigste mulige rejse. Under klient / server begrebet vil der blive designet en web-baseret klient, som giver mulighed for reservation gennem internettet. Designmæssigt tilstræbes systemet at ligne eksisterende løsninger inden for online bookingsystemer.

I først del af rapporten tager vi fat på designdelen, som danner baggrund for udviklingen af systemet. Der gøres her brug af UML (Unified Modeling Language), der hjælper os med at visualisere, dokumentere og specificere den dokumentation vi skal bruge i vores udviklingsproces. Anden del af rapporten omhandler softwarekonstruktion, hvor vi ser nærmere på teorien bag vores implementerede teknologier. Vi vil desuden argumentere for, hvorfor valget faldt på visse teknologier / algoritmer frem for andre og dele af systemet belyses i form af kodeeksempler. Vi vil afsluttende opsummere og konkludere.

Problemformulering

Vi har gennem casen Flight Routing fået stillet til opgave at designe et n-tier system, der skal omfatte følgende dele:

- Klient / server system
- Rejseplanlægger ud fra en valgt algoritme
- Web-baseret klient

På baggrund af casen vil vi derfor undersøge, hvordan vi kan designe et brugervenligt n-tier system, der kan håndtere mulige concurrency problemer samt tilgodese kundes behov for at finde den billigst mulige flyrejse mellem flere lufthavne.

Systemet skal som udgangspunkt kunne håndtere:

- Administration af CRUD funktionaliteterne
- Finde den billigste rute fra afgang til ankomst med mulige stop på vejen
- Reservation/booking af flyafgange for kunder

Læringsmål

Der er i forbindelse med dette projekt opstillet nogle læringsmål. Disse er som følgende:

C#:

- Entity Framework
- WCF Distributed programming
- Parallel programming and concurrency
- Windows Forms
- The programming language C# and the .NET platform
- Language and parsing (HTML, CSS and XML)
- ASP.NET programming with C#
- Test of Server / Client

Project:

- Be able to identify and construct algorithm
- Distributed Architecture

Software Design

Use Case

Vi bruger use cases til at beskrive de funktionelle krav. Vi har her fokus på hvem der bruger systemet, samt deres mål og perspektiv. Vi har fundet vores use cases ved at identificere de primære brugere af systemet samt deres mål.
De fleste af vores use cases omhandler ren CRUD funktionalitet og er derfor kun beskrevet brief, disse kan ses i vores bilag #5. De mest komplekse use cases er beskrevet Fully Dressed.

Use Case Fully Dressed

Som en del af opgavebeskrivelsen var "planning a route" lagt ud til os som fully dressed use case. Denne ligger under bilag #2. Vi har efterfølgende udarbejdet vores udgave af denne use case, som vi har kaldt "Flight Reservation". Denne behandler hændelsesforløbet for planlægning af en rejse.

Use-case	**Flight Reservation**
Scope	Flight booking applikation
Niveau:	Brugerdefineret mål
Primær aktør	Kunde
Interessenter	Firma, kunde, medarbejdere

Preconditions	Kunde er oprettet og logget ind i systemet. Fly, lufthavne, ruter samt afgange er oprettet	
Succeskriterier	En billet med tilhørende sædereservationer er oprettet	
Basic Flow	1. Brugeren vælger rejsedestination, afrejsedato, antal passager, billettype samt rejsetype	2. System viser afgange, rejsetid og priser
	3. Brugeren vælger en afgang	4. System viser detaljeret rejsedata samt pris
	5. Brugeren trykker vælg sæder i fly	6. Systemet viser ledige sæder i flyet
	7. Brugeren vælger et sæde og bekræfter valg	8. Systemet bekræfter og viser persondata
	9. Brugen bekræfter oplysningerne	10. System viser betalingsmuligheder
	11. Brugeren betaler	12. Systemet bekræfter og kvittering sendes
Alternative Flows	1. Brugeren vælger at få vist -/+3 dage -> Systemet viser en liste med datoer samt tilbud på afgange 5. Brugeren vælger at se hjemrejse muligheder 6. Systemet viser mulige hjemrejse datoer og priser 7. Brugeren vælger en hjemrejse 10. System viser oversigt over rejsen samt mulighed for tilvalg	

Use Case Fully Dressed - "Flight Reservation"

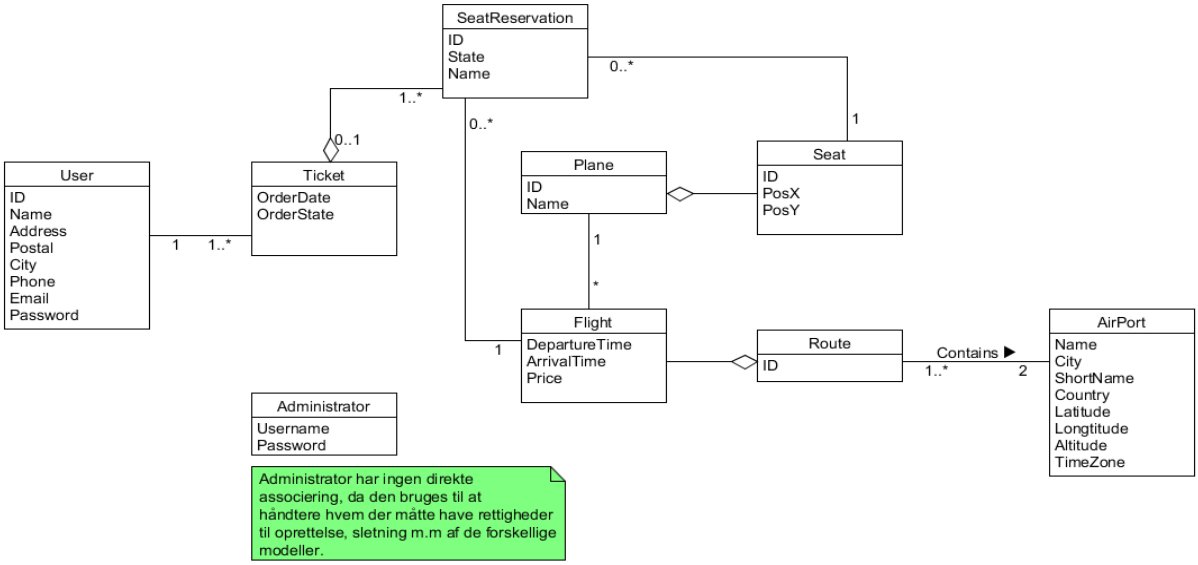
Opsummering

Vi har redegjort for systemets funktionelle krav samt fundet systemets use cases og har beskrevet den mest komplekse use case i Fully Dressed format. Det næste vi vil se nærmere på er systemets business logic. Det gør vi i en mere objekt orienteret sammenhæng ved at lave en domænemodel. Emnerne til klasserne er fundet på baggrund af en udarbejdet kandidatliste, som kan ses i bilag #4.

Domænemodel

I vores domænemodel vil vi give et billede af de objekter som programmet kommer til at indeholde. Disse er identificeret ud fra kandidatlisten, som kan ses på bilag #4. Domænemodellen anvendes desuden også til at identificere sammenhængen mellem de forskellige objekter i form af associeringen og aggregeringer.

Domænemodellen på figuren er bygget op omkring det klassiske aftalemønster (Kunde -> Aftale -> Ydelse). I dette tilfælde User som Kunde, Ticket som Aftalen, der indeholder en eller flere SeatReservations. Denne indeholder Ydelsen i form af Flights og Seats.



Domænemodel

- Fly (Plane) har en aggregering i form af et antal sæder (Seat) på flyet
- Lufthavne (Airport) har associeringen en til mange ruter (Route)
- Afgang (Flight) som har en aggregering til en bestemt rute (Route), da der er et antal afgange til en rute (Route)
- En afgang (Flight) har associering til et fly og til et antal sædereservationer (SeatReservation)
- Sædereservation (SeatReservation) har så associering til seat, da navnet antyder er en reservation af et sæde. Derudover har den en aggregering til en billet (ticket)
- Kunden har associering til billet (Ticket), da kunden kan have en til mange billetter
- Vi har valgt at opstille klassen Administrator alene, da denne er en enkeltstående klasse til brug i administratordelen af systemet og derfor vil denne klasse ikke have noget forhold til de andre klasser

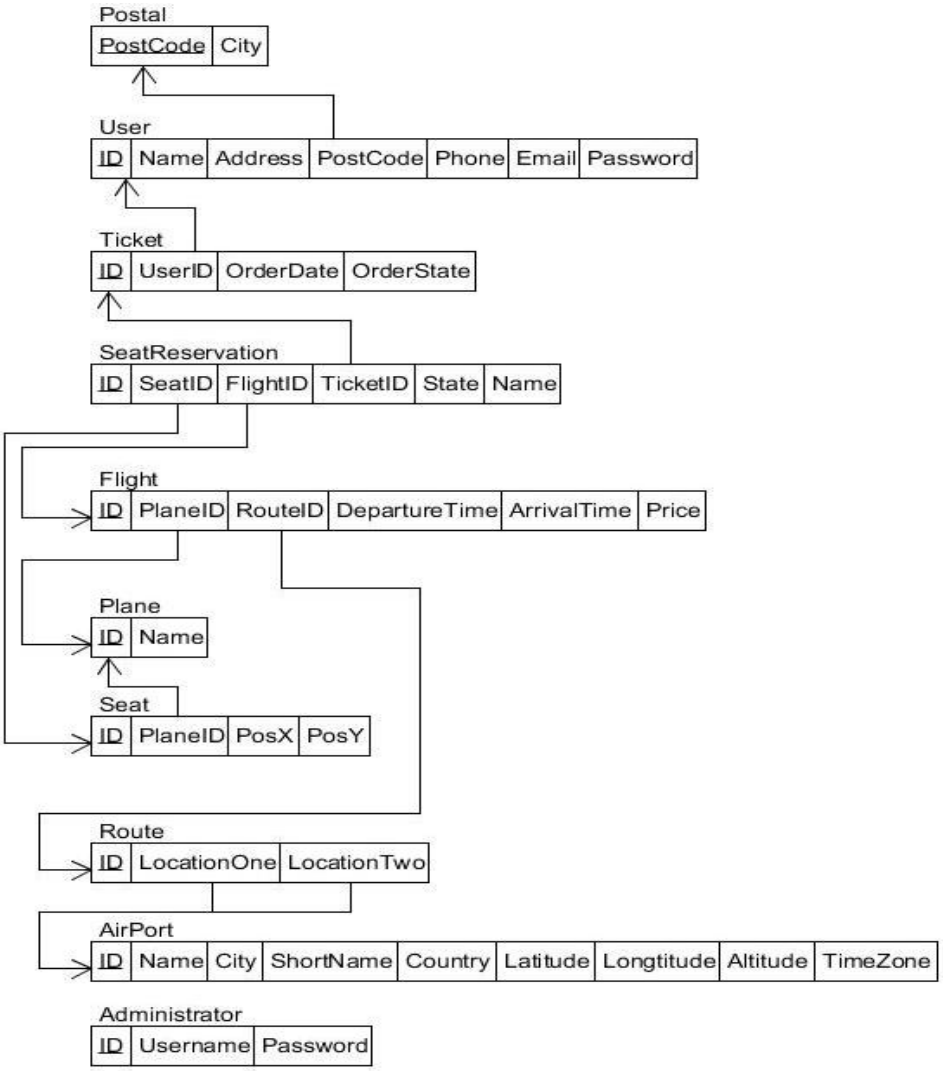
Relationel Database Model

Vores database design er konstrueret på baggrund af vores domænemodel samt associationerne mellem klasserne. For hver klasse er der blevet lavet en tabel, hvor kolonnenavne er oprettet på baggrund af attributterne på klasserne. Ved en til en relationer mellem klasserne har vi taget primærnøglen fra den ene tabel og placeret som fremmednøgle på den anden tabel. Ved en til mange relationer er fremmednøglen placeret på mange siden.

En map/junction tabel med composite keys vil være at foretrække ved mange til mange relationer, men vi har ingen mange til mange relationer i vores model, og har derfor ikke gjort brug af dette design.

Normaliseret Databasemodel

Vores database design har vi valgt at betragte ud fra Boyce- Codd normalform. Når en tabel er på Boyce-Codd overholder den samtidigt 1.-, 2.- og 3. Normal Form, som kan ses i bilag 8.

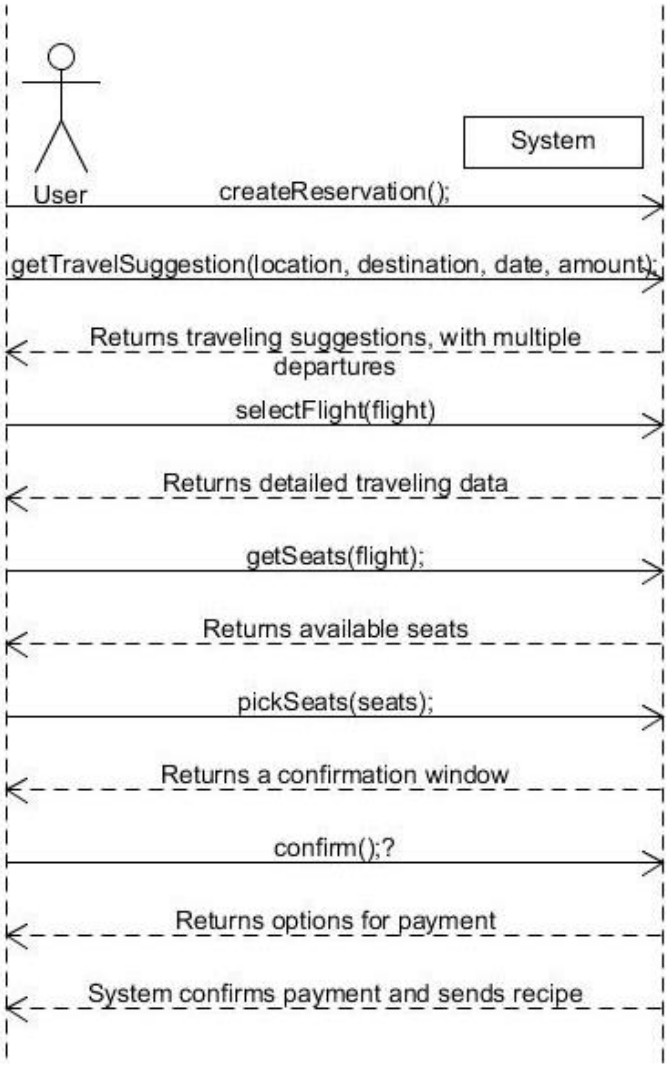


Normaliseret Databasemodel

På figuren ses den normaliserede databasemodel. For at undgå redundans og overholde NF. 3 har Postnummer og By har fået deres egen tabel.

SSD - Flight Reservation

Dette systemsekvensdiagram viser input og output handlinger relateret til use casen "Flight Reservation". Diagrammet viser ikke hvad der foregår i systemet, men kun det som bliver sendt retur til brugeren.



SSD - Flight Reservation

Brugeren vil oprette en reservation, hvor systemet så melder tilbage med mulige afgange for afrejsen. Derefter får brugeren mulighed for at vælge en afgang og systemet returnere detaljerne omkring afrejsen. Nu vil brugeren så vælge sæder, og systemet melder tilbage med de ledige sæder på flyet som knytter sig til afgangen.
Brugeren vælger de sæder vedkommende ønsker, og systemet melder tilbage med besked om at brugeren skal godkende reservationen. Derefter godkender brugeren reservationen, og systemet melder tilbage med betalingsmuligheder. Brugeren betaler for afrejsen, og system sender en kvittering til brugeren.

Operationskontrakter

Her er en af vores operationskontrakter tilknyttet SSD'en "Flight Reservation". Denne indeholder de pre- og post conditions, som operationen måtte kræve for at kunne udføres. Postcondition beskriver ændringer i associationerne eller variabler. De resterende operationskontrakter kan ses i bilag #7

getShortestPath

```
Operation: getShortestPath(location, destination, date, amount);
Use Case: Flight Reservation

Precondition:
• Flight, Route, AirPort, Plane, Seat exist

Postcondition:
• Multiple instances flight of Flight is created
• flight.DepartureTime, flight.ArrivalTime, flight.Price, flight.Routes is assigned values
```

Design Klassediagram

Vores design klassediagrammet kan ses i bilag #6. Det viser klasser, interfaces og deres associationer. Diagrammet har til formål at give et statisk syn over systemets information.

Software Design Opsummering

I designdelen har vi redegjort for systemets funktionelle krav samt fundet systemets use cases, og har beskrevet den mest sammensatte use case i Fully Dressed format. På baggrund af en kandidatliste fik vi konstrueret en domænemodel, der visuelt hjælper os til bedre at forstå systemet. På baggrund af domænemodellen fik vi lavet et databasedesign, der hjalp os med at modellere vores tabeller. I sidste del fik vi analyseret systemets handlinger via et Systemsekvensdiagram. Vi fik oprettet operationskontrakter med gennemgang af pre- og post conditions samt lavet et design klassediagram til at illustrere klassernes interfaces og deres associationer.
I det næste afsnit vil vi se nærmere på nogle af de teknologier vi bruger samt rette fokus på den programmeringsmæssige del af projektet.

Softwarekonstruktion

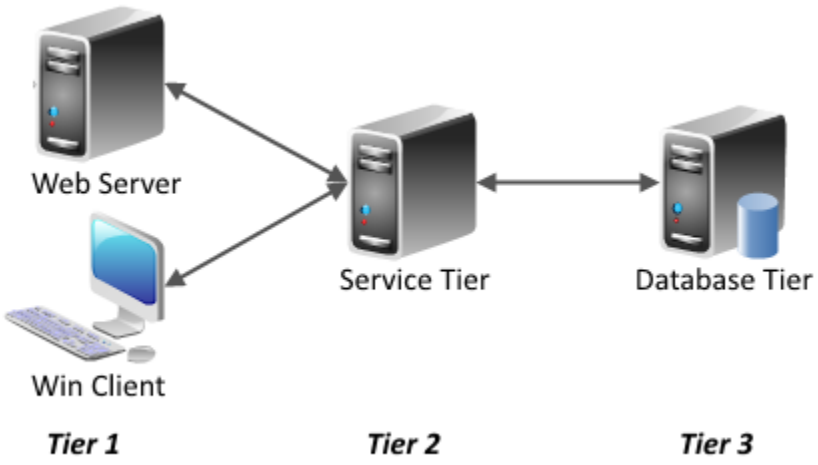
I dette afsnit vil vi kigge nærmere udvikling af vores system, i forhold til de valg som blev truffet i det foregående afsnit. Derudover vil vi også komme bag om teorien bag nogle af de teknologier, vi benytter i udviklingen af vores system.

Distributed System

Et af kravene til vores system-arkitektur var at det skulle være i form af et distribueret system. I et distribueret system er data og datakraft ikke centraliseret et enkelt sted, men derimod er visualisering, data og beregningsmodeller fordelt over flere noder, som arbejder sammen.

N-Tier arkitektur

Vi har valgt at lave en N-Tier arkitektur, hvor applikationen er delt op i flere tiers:



- Tier 1 - Presentation: Dette er hvad kunden ser i browseren
- Tier 2 - Application: Det er laget med vores business logik
- Tier 3 - Data: Vores database

De forskellige tiers udgør tilsammen vores system, men de opererer også uafhængigt af hinanden. Hver tier er placeret på forskellige computere/servere. Denne opbygning giver følgende fordele:

- Security: Ved at adskille lagene kan vi konfigurere sikkerhed forskelligt for hver tier
- Scalability: Mulighed for at hvert lag kan udvides uden at røre de andre tier
- Simple Maintenance: Hvert lag kan administreres uafhængigt af de andre tier
- Resource Sharing: Det er lettere at dele data og informationer

Foruden de praktiske fordele der kan være forbundet med et et distribueret system bør det nævnes, at det ofte er mere kosteffektivt, da flere billige spredte processorer ofte giver bedre pris/performance ratio en et enkelt stort centraliseret system. Dette er desuden ensbetydende med et hurtigere system, dog kunne en potentiel flaskehals her være et dårligt netværk.
På trods af mange fordele er der også problemer forbundet med et distribueret system:

- Concurrency: Håndtering af samtidighedsproblemer
- Data synchronization: Behov for kontrol af data, mellem tiers og flere instanser af samme tier
- Security: Mulighed for djævleklienter, behov for kryptering og autentificering

Windows Communication Foundation

Vi har i vores system valgt at bruge Windows Communication Foundation (WCF), som er specifikt designet til at lave distribuerede systemer. API'en giver en samlet programmeringsmodel, der kan bruges sammen med mange andre tidligere teknologier til distribuerede systemer.
Tidligere anvendte API'er var ofte begrænset af at ikke alle typer klienter kunne bruge funktionaliteten uden at kende til de underliggende detaljer af systemet.
Med WCF API'en kan vores service tilgås af flere typer klienter og kan eksempelvis blottes med HTTP protokollen over for en type klient mens et in-house system ville kunne udnytte en TCP-baseret protokol og binary formatting til dataen.

Vores WCF-Application er opbygget på følgende måde:

- WCF Service assembly: Dette er et class library med classes og interfaces. Det er her den overordnede systemfunktionalitet findes.
- WCF Service Host: Vores system er på nuværende tidspunkt hostet i Windows service host.
- WFC Client: Systemet består af en Administrator-applikation med et UI lavet i Winform og en Webapplikation til kunde interaktion

Binding

Host og klienter kommunikerer med hinanden via enighed omkring Adresse, binding og Contract (ABC). Adressen angiver lokationen på servicen, binding angiver hvilken netværksprotokol der er brugt, og kontrakten beskriver, hvilke metoder kan ses fra servicen.
I vores applikation har vi valgt at bruge HTTP som binding til de fleste af vores services dog bruger vi WSHttpBinding til vores ReservationService, da denne type binding understøtter sessions. Dette gøres for at sikrer mod en klient som ikke fuldfører en reservation (evt. pga. crash). Hvis sessionen får timeout, sørger servicen for at de reserverede sæder igen bliver ledige.

```
<wsHttpBinding>
  <binding name="FastTimeOut" receiveTimeout="00:00:40">
  </binding>
</wsHttpBinding>
```

Viser et eksempel på receiveTimeout sat til 40 sek.

WCF Instancing

WCF Concurrency hjælper med at konfigurere hvordan vi håndtere flere servicekald på samme tid. Vi er derfor interesseret i at vide noget om serviceobjekternes tilstand og levetid.

Kontrollen over WCF serviceobjekternes levetid på serveren kan håndteres i WCF. Vi kan lave service instances på følgende måder:

- Per call (for hver WCF klient-metodekald bliver en WCF service instance lavet og efterfølgende ødelagt efter behandling.)
- Per session (samme WCF service instance behandler flere metodekald før den bliver ødelagt efter endt behandling.)
- Single instance (en global WCF service instance behandler forespørgsler fra flere klienter)

Da de fleste af vores services er stateless bruger vi per call, som også er default tilstand ved brug af HTTP-binding. Vores ReservationService kører med wsHttpBinding som understøtter sessions. Default setting for denne service er derfor per session, hvilket også er hvad vi bruger.

WCF Concurrency

Der er tre måder hvorpå WCF kan håndtere concurrency:

- Single
- Multiple
- Reentrant

I ServiceBehavior attributten kan vi angive hvordan vores service skal konfigureres.

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single,
InstanceMode = InstanceMode.PerSession)]
public class ReservationService : IReservationService {
```

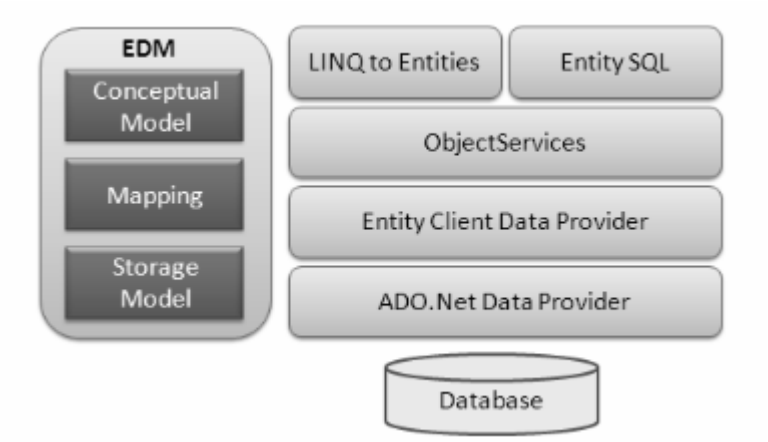
Pr. default er servicen konfigureret til at bruge PerCall og Single Concurrency Mode, hvilket betyder at hver klient får tildelt en thread, som vil håndtere hver af klientens forespørgsler. For hvert kald bliver der oprettet et nyt serviceobjekt, hvorpå der kun kan køre en Thread. Vores ReservationService er konfigurere med InstanceMode til PerSession og ConcurrencyMode single.

Entity Framework

Vores endelige database er konstrueret på baggrund af Entity Framework - Code First, som er et Object Relational Mapping framework. Dette kan automatisk oprette databasetabeller og omvendt også oprette klasser på baggrund af en eksisterende database. Med Entity Framework kan vores applikation arbejde med relationel data såsom domæne specifikke objekter. En af fordelene ved at bruge frameworket er at der ikke skal skrives så meget kode, og vi kan arbejde på et højere abstraktionsniveau, da vi ikke behøver at bekymre os så meget omkring databasens tabeller og kolonner, og hvor det er gemt. Frameworket er baseret på ADO.NET og har følgende arkitektur:

Entity Framework Arkitektur

EDM(Entity Data Model): Conceptual Model, Mapping, Storage



- Conceptual Model: Model klasser og deres forhold
- Storage Model: Database design med tabeller, udpluk, relationer og nøgler
- Mapping: Information omkring hvordan Conceptual Model er mappet til Storage Model
- Linq to Entities: Med Linq kan der skriver queries mod modellen som returnere entities som er defineret i den konceptuelle model
- Entity SQL: En anden måde at skrive queries på som også kan bruges
- Object Service: Står for at konvertere dataen fra det underliggende lag (Client Data Provider) til objekter
- Entity Client Data Provider: Dette lag konverterer LINQ to entities og LINQ SQL til SQL query som databasen forstår. Den kommunikere med ADO.NET (Data Provider)
- ADO.NET Data Provider: Kommunikere med databasen via standard ADO.NET

Frameworket understøtter konventionerne “Code First”, “Database First” og “Model First”, alle metoder kan bruges med eller uden en eksisterende database. Vi har brugt fremgangsmåden “Code First”, hvor Database og tabeller bliver oprettet på baggrund af vores model klasser. Vi har valgt denne fremgangsmåde, da den ikke autogenerere kode for os. Vi har på den måde fuld

kontrol over koden, og vi skal ikke til at modificere i noget autogenereret kode, hvilket kan give problemer.
Relationerne mellem tabellerne er baseret på relationerne mellem vores model klasser. Primærnøglen på alle vores tabeller er dannet på baggrund af det ID-property, som vi har placeret på model klasserne. Til at override nogle af frameworkets standard konventioner bruger vi Data Annotation samt Fluent API. Nedenfor kan ses eksempler på, hvordan vi bruger disse former:

```
[DataMember]
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int ID { get; set; }
```

Viser vores ID-Property markeret som identity på databasen

```
[NotMapped]
[DataMember]
public string PasswordPlain { get; set; }
```

Her fortæller vi databasen at PasswordPlain ikke skal med i databasen

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
    base.OnModelCreating(modelBuilder);
    //Fix ForeignKey Between Route And Airport;
    modelBuilder.Entity<Route>().HasRequired(r => r.From).WithMany(a => a.Routes);//.WillCascadeOnDelete(false);

    modelBuilder.Entity<Flight>().HasRequired(f => f.Route).WithMany(r => r.Flights).WillCascadeOnDelete(true);
```

Fluent API konfiguration af mange til mange tilhørsforhold, samt On Delete Cascade, når en Flight slettes.

Concurrency i Entity Framework

Til at håndtere concurrency udnytter vi at frameworket supporter optimistic concurrency. EF gemmer en entity i databasen i håbet om at det samme data ikke er ændret siden det sidst var loaded. Hvis dataen har ændret sig smider den en exception, som skal håndteres inden dataene forsøges gemt igen.

```
[DataMember]
[Timestamp]
public byte[] Concurrency { get; set; }
```

I projektet har de klasser der skal håndterer concurrency fået et property af samme navn, der er markeret med Timestamp attributten. Datatypen er af byte, da Timestamp er binary i c#. EF inkluderer nu et property i where clausen, som bruges under opdatering.

Datastrukturer

I dette delafsnit vil vi komme ind på nogle af de datastrukturer vi benytter i vores projekt og forklare, hvordan de fungerer.

List<T>

En generisk List er defineret ved List<T>, hvor T er datatypen. Dette kan være et objekt eller simple type som int, string, mv. Man kan kun putte elementer ind af den type listen er initialiseret med. Som andre collections indeholder en generisk List en række metoder som eksempelvis add, remove, mv. hvor man kan tilføje eller fjerne elementer fra listen. Derudover er der også metoden Contains, hvor man kan tjekke om listen indeholder et bestemt element.

LinkedList<T>

En generisk LinkedList er defineret ved LinkedList<T>, hvor T er datatypen. Sat op imod en generisk List som er index baseret, er LinkedList baseret på at hver node kender den næste node. Standard implementationen fra .NET 2 er en Doubly LinkedList, hvor hver node kender både den foregående og efterfølgende node.

I forhold til den generiske List som fordobler sit array, hver gang den rammer sit loft, er LinkedList mere dynamisk og forhøjer sit array med en enkelt. Derpå kan man forsvare at en LinkedList ville være bedre i situationer, hvor man har meget data i en List.

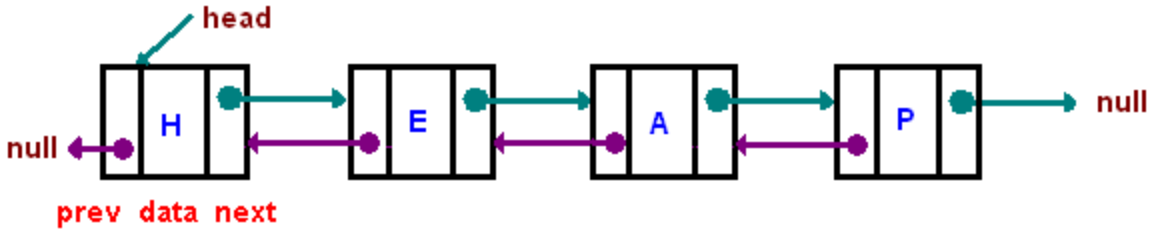


Diagram over en Doubly LinkedList.

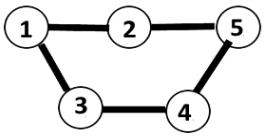
Hvert element (node) i en Doubly LinkedList består af tre ting, dataene, reference til listens næste node og den foregående node. Begyndelsen er hovedet på listen som indeholder en reference til den næste node og en reference til null. Den sidste reference er en reference til null, og dens foregående node.

Algoritmer

Et af succeskriterierne for vores application var implementationen af en algoritme der kunne beregne den billigste flyrute fra start til slutdestination med mulighed for mellemlandinger i flere lufthavne. Til at løse problemet tog vi udgangspunkt i to eksisterende algoritmer A-star og Dijkstra. Begge algoritmer minder om hinanden, men A-star gør brug af heuristikker.
Da vi er stillet opgaven at finde den billigste rute, og givetvis ikke den korteste rute, kan vi ikke gøre brug af A*, da vi ikke har nogle heuristikker, f.eks. hvilken retning der vil være billigst. Derfor er valget faldet på Dijkstra, hvor vi bruger prisen som den vægtede faktor.

Dijkstra

Dijkstra's algoritme er baseret på grafteori. En graf består af knuder og kanter. En knude er illustreret (på nedenstående figur) som en cirkel og kanten som en linje eller kurve, der forbinder to knuder i grafen. En graf er defineret ved et par (V, E), hvor V er en mængde af knuder som ikke er tom. E er en mængde af knudepar.



I denne graf er der 5 knuder med tallene 1-5, der er også 5 kanter.
V = {1, 2, 3, 4, 5} og E = {{1,2 },{1,3},{2,5},{3,4},{4,5}}

I vores implementation har vi en række lufthavne som vi kan illustrere som knuder(vertices). Forbindelsen mellem de forskellige lufthavne kan vi betragte som kanter(edges) i en graf. Vi har valgt prisen som kriterium til at vægte de forskellige kanter (forbindelser). Den billigste vej gennem grafen er den rute, hvor de adderede kanter er vægtet lavest. Ideen er at der vælges en knude(lufthavn), hvor der laves ruter ud fra. Dennes start pris er 0 kr. Der laves en liste med alle ubesøgte lufthavne. Den foreløbige pris mellem start lufthavnen og dens naboer beregnes herefter. Vi sammenligner derefter de beregnede afstande med hinanden, og vælger den med den laveste pris til at være besøgt. Med udgangspunkt i den sidste besøgte lufthavn med laveste pris fortsættes processen til vores destinations lufthavn er markeret som besøgt.

Kodeeksempler

I dette delafsnit vil vi gennemgå et kodeeksempel, som omhandler løsningen af vores vigtigste use case "Flight Reservation". Samtidig vil der også blive vist, hvordan Dijkstra's algoritme er implementeret i vores system.

Der vil herunder være en gennemgang af metoden MakeSeatsOccupiedRandom, i ReservationService. Denne Service kører med PerSession, og metoden har følgende annotation: [OperationContract(IsInitiating = true, IsTerminating = false)]. Hvilket betyder at metode er en "start"-metode.

```
public Ticket MakeSeatsOccupiedRandom(List<Flight> flights, int noOfSeats, User user) {
            if (ticket == null) {
                try {
                    CreateTicket(user);
                } catch (NullException) {
                    throw new FaultException<NullPointerFault>(
                        new NullPointerFault("user is not valid or not found in database",
"user"));
                }
            } else {
                if (ticket.User.ID != user.ID) {
                    throw new FaultException<NotSameObjectFault>(new NotSameObjectFault(new
NotSameObjectException("Added user is not the same as first run")), new FaultReason("added
user is not the same as first run"));
                }
            }
            if (flights == null || flights.Count == 0) {
                throw new FaultException<NullPointerFault>(new NullPointerFault("flights is
not valid", "flights"), new FaultReason("flights is not valid parameter"));
            }
            if (noOfSeats < 1) {
                throw new FaultException<NullPointerFault>(new NullPointerFault("noOfSeats
is not valid", "noOfSeats"),new FaultReason("noOfSeats is not valid"));
            }
```

I den første del af metoden bliver der oprettet en ticket med en tilhørende user. Herudover sker der validering af inputs fra klienten.

```
            OperationContext.Current.InstanceContext.Closed -= InstanceContext_Closed;
            OperationContext.Current.InstanceContext.Closed += InstanceContext_Closed;

            Ticket oldTicket = ticket.Clone();
```

Her bliver der sat en event-metode på, som bliver kørt hvis sessionen på servicen udløber. Endvidere bliver der oprettet en klon af ticket objektet, hvis noget i det følgende skulle fejle.

```
        try {
            using (var db = new FlightDB()) {

                flights = GetFlights(flights, db);

                // ReSharper disable once PossibleNullReferenceException
                ticket.SeatReservations = GetRandomSeatReservations(flights, noOfSeats,
db);
```

Her bliver flights hentet fra databasen. Efterfølgende genereres en liste med det ønskede antal sæder i form af SeatReservation objekter, som har forbindelse til et flight og et sæde. Denne liste bliver sat på ticket.

```
                if (ticket.ID == 0) {
                    db.Users.Attach(ticket.User);
                    db.Tickets.Add(ticket);

                    db.SaveChanges();
                }
```

Hvis der er tale om et nyt ticket objekt, bliver User objektet først tilføjet til konteksten. Det samme gør ticket. Nu vil EF forsøge at gemme ændringer (ticket og tilhørende seatReservations).

```
else {
                    var tempUser = ticket.User;
                    ticket.User = null;
                    var existingSeatRes = db.SeatReservations.Where(x => x.Ticket.ID ==
ticket.ID).ToList();
                    db.SeatReservations.RemoveRange(existingSeatRes);

                    foreach (var sr in ticket.SeatReservations) {
                        db.Entry(sr).State = EntityState.Added;
                    }

                    db.Tickets.Attach(ticket);
                    db.Entry(ticket).State = EntityState.Unchanged;

                    db.SaveChanges();
                    ticket.User = tempUser;
                }
```

Hvis det ikke er et nyt ticket objekt oprettes en lokal variable med User objektet og efterfølgende pilles user af ticket. Dette gøres for at undgå at EF vil indsætte User'en i databasen på ny. Dernæst fjernes eksisterende seatResevation og de nye seatReservation sættes som tilføjet i konteksten. Endvidere tilføjes ticket, og der fortælles til EF at ticket er uændret. Der forsøges at tilføje de slettede, ændrede og nye ting til databasen. Lykkes det sættes User igen på ticket.

```
            UpdateDijkstra();

            } catch (NotFoundException ex) {
                throw new FaultException<NotFoundFault>(new NotFoundFault(ex), new
FaultReason(ex.Message));
            } catch (NotEnouthException ex) {
                throw new FaultException<NotEnouthFault>(new NotEnouthFault(ex), new
FaultReason(ex.Message));
            } catch (Exception ex) {
                ticket = oldTicket;
                Trace.WriteLine(ex);
                Trace.WriteLine(ex.Message);
                throw new FaultException<DatabaseFault>(new
DatabaseFault("MakeSeatOccupiedRandom Error"), new FaultReason("MakeSeatOccupiedRandom
Error"));
            }
            return ticket;
        }
```

Det sidste som sker i try'et er at vores dijkstra bliver opdateret med de ændringer som er sket. Hvis noget går galt i try'et bliver fejlene fanget og smidt videre som faultexceptions, endvidere bliver ticket sat til klon objektet. Herefter returneres ticket objektet.

Systemet

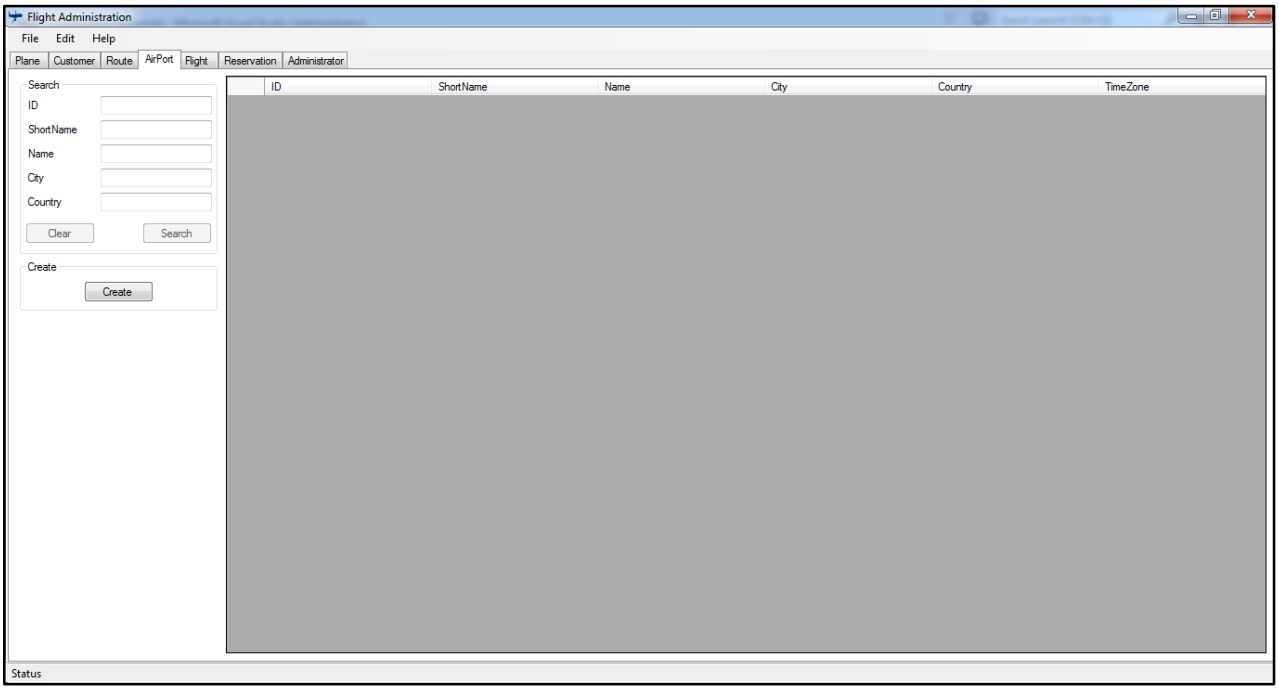
I dette delafsnit vil vi se på vores system på baggrund af de valg vi tog i design afsnittet. ligeledes gives en kort gennemgang af vores to klienter.

Vi har igennem konstruktionsfasen opbygget to forskellige klienter til brug af servicen, henholdsvis en Winform Application til administration, og en Webklient som giver mulighed for at bruge vores implementation af Dijkstra algoritmen, til at finde den billigste rute fra *A* til *B*.

Administrator Klienten

Formålet med vores administrator klienten er at medarbejderne i virksomheden kan benytte den til interne operationer. Med klienten kan der udføres simple CRUD operationer i systemet, f.eks oprettelse af nye lufthavne, fly, m.v.

Ud fra bilag #1 kan mockup for denne klient ses, her vil vi blot vise et screenshot af klienten.



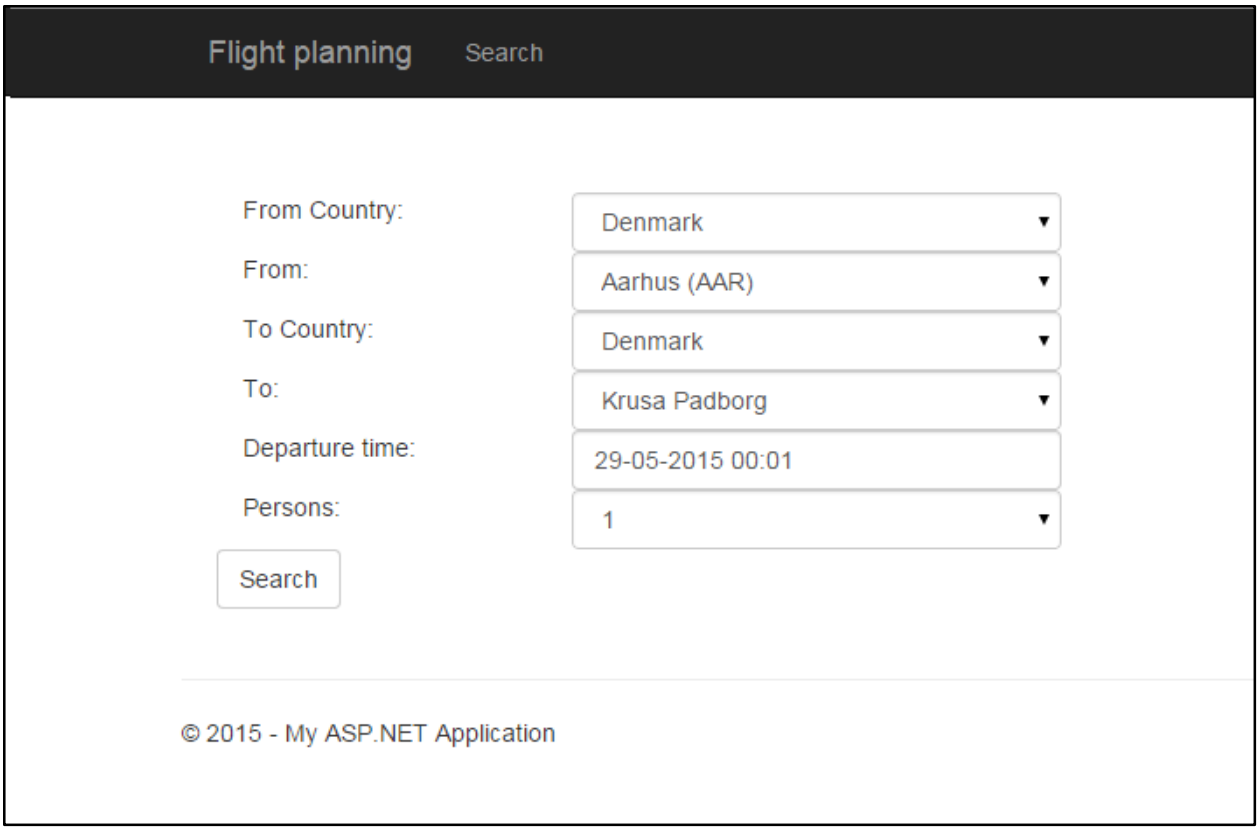
Klienten er opbygget med tab layout som gør det muligt at skifte mellem de forskellige vinduer i programmet. Hver tab er opbygget på samme grundlæggende udseende, inddelt i to bestanddele. Venstre side består af en række tekstfelter og knapper til operationer, højre side består af en tabel.

Tekstfelterne giver brugeren mulighed at kunne søge ud fra en række søgekriterier, f.eks en rute kan findes baseret på fra og til destinationen. Når der trykkes på søgeknappen dukker de fundne elementer op i tabellen. Brugeren kan så højreklikke på et element for at redigere det. Derudover er der også en knap som åbner et nyt vindue. Her kan brugeren oprette et nyt element, f.eks. en ny rute.

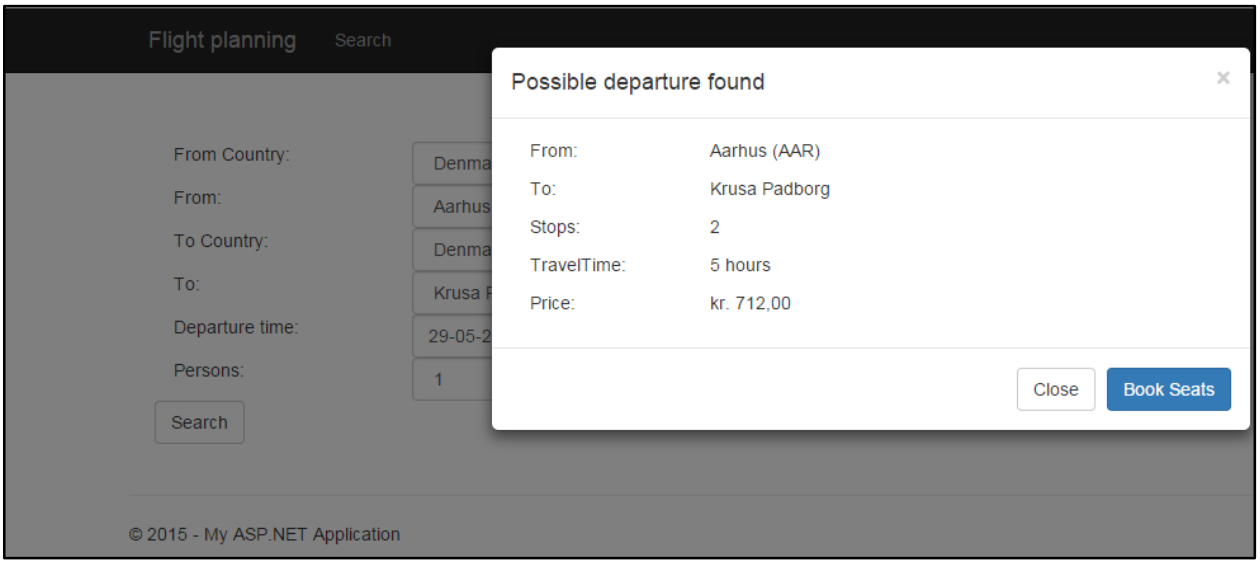
Web Klienten

Vores webklient er en online klient, hvor igennem der kan bookes en rejse. Når brugeren booker en rejse, vil systemet beregne den billigste mulig rejse ud fra de valgte lufthavne. Efterfølgende bliver denne rejse tilbudt til brugeren. Systemet holder ved hjælp af en session på servicen styr på hvilke afgange brugeren er i gang med at bestille. Ligeledes vil der ved sessionens udløb, f.eks hvis kunden forlader computeren/siden for længe, vil reservationerne af sæderne blive annulleret, og igen gjort tilgængelig for andre kunder.

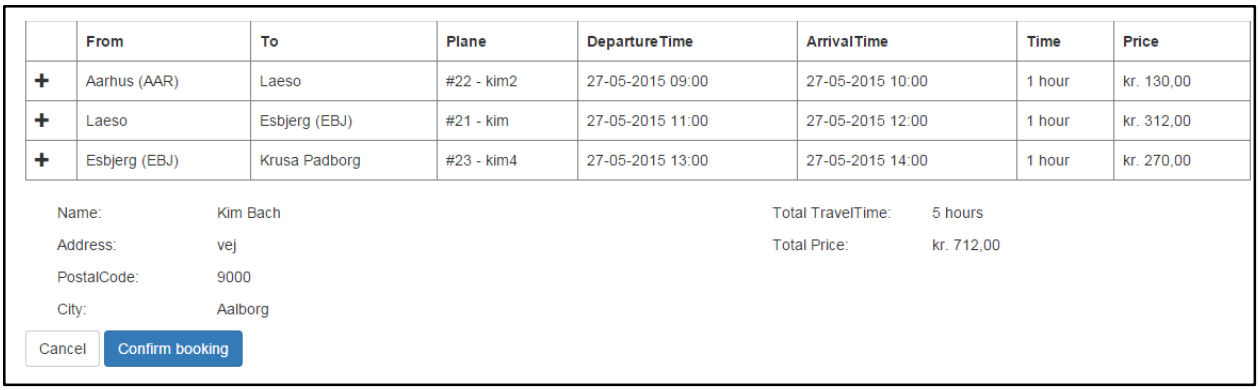
Det følgende viser webklienten i brug og samtidig et eksempel på en kunde, der ønsker at rejse fra Aarhus til Kruså som enkeltperson:



Kunden udfylder hvor vedkommende vil rejse fra og til, samt hvornår denne ønsker at rejse, og endvidere hvor mange personer, der ønskes billet til.



System returnere en afgang, hvor der er 2 stop undervejs. Kunden får mulighed for at booke sæder på afgangene. Kunden får endvidere information om den samlede rejsetid og pris.



Derefter kommer der en side med en udspecificering af rejsen, hvor kunden har mulighed for at bekræfte reservationen.

Sikkerhed

Der har i forbindelse med udarbejdelsen af vores system været visse sikkerhedsovervejelser.
Der er valgt at passwordet er gemt i databasen med en unik salt for hver bruger. Den sensitive logik er placeret på vores service, og hashen bliver aldrig returneret til klienten.
Vi har fra start haft planer om at bruge vores egen authentication i form af vores administrator objekt til at tilgå servicen.

Tests

Der er kørt forskellige tests, blandt andet i form af hosting på Microsoft Azure og VPN (gennem TeamViewer). Hvorigennem der er testet forskellige samtidighedsscenarier.

Software Konstruktion opsummering

Der er igennem Software Konstruktion opbygget et klient / server system, baseret på arkitekturen over n-tier systemer. Der er blevet forklaret om WCF, og hvordan det bruges i forhold til concurrency og instancesmode. Endvidere blev der gennemgået hvilke fordele Entity Framework giver.

Desuden blev der kigget på udvalgte algoritmer som mulige kandidater for vores ruteplanlægger, samt hvordan algoritmerne Dijkstra og A* fungerer. Valget faldt på Dijkstra's algoritme, da det ikke var muligt at identificere heuristikker til brugen af A*.

Ligeledes er der fortaget en gennemgang af administrator- og webklienten, samt sikkerhedsaspekterne.

Konklusion

Der er igennem denne rapport arbejdet med vores primære use case "Flight Reservation". Hvor vi i afsnittet Software Design har identificeret systemets klasser, og ud fra dem opstillet en domænemodel. Der er ligeledes udarbejdet databasemodel, SSD og operationskontrakter. Endvidere har vi i afsnittet Software Konstruktion fundet frem til de teknologier og løsninger, som der er gjort brug af. Her kan herunder nævnes WCF og Entity Framework, samt Dijkstra's algoritme.
På baggrund af ovenstående har vi implementeret et bookingsystem, i form af et klient /server system med tilhørende winform- og webklient, hvorved medarbejdere kan håndtere CRUD funktionaliteterne, og kunder kan reservere flyrejser. Kundernes søgningen er baseret på algoritmen med billigste rejse som succeskriterie for, hvilken rute kunden får præsenteret.

Derudover har vi undersøgt, hvordan vores system kan håndtere concurrency. Vi har konkluderet at Entity Framework kan håndtere concurrency problemer i form af optimistic concurrency. På servicen gav WCF, flere muligheder til håndtering af concurrency, her bruges PerCall og Single Concurrency Mode, samt InstanceMode til PerSession og ConcurrencyMode single. Ligeledes har vi til vores Administrations klient valgt at gøre brug af BackgroundWorkers, for at udnytte multithreading, og derved få et responsivt UI.

Evaluering

I forhold til kravene til produktet er vi godt tilfredse med det opnåede resultatet. Dog er der visse ting vi gerne ville have haft implementeret. På webklienten ville vi gerne have lavet mulighed for sædevalg, således kunderne kunne vælge, hvor de ville sidde på flyet.

Endvidere fandt vi ud af hvis vi ønskede at bruge administratorklassen til authentication, skulle vi lave Custom Authentication / Authorization til dette. Derfor gik vi over til at servicen i stedet skulle hostes, hvor med mulighed for at bruge Windows Authentication. Hvilket ville have givet fordelen, at der kunne oprettes brugere med tilhørende roller under et givet domæne. Vi har desværre ikke nået at teste dette i praksis.

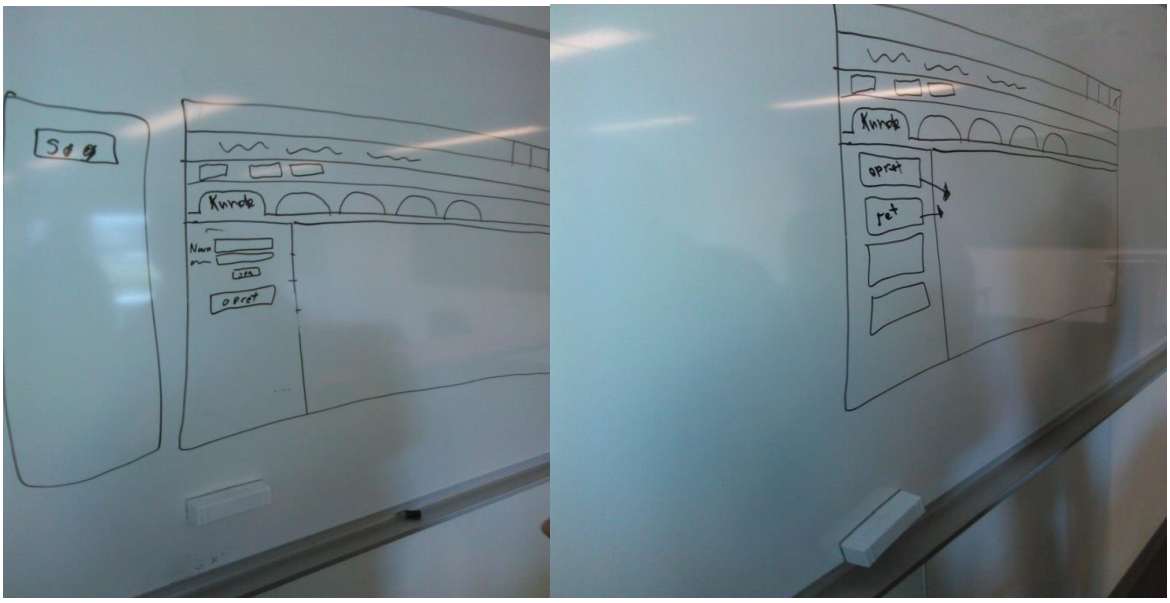
Vi havde under implementering af Dijkstra's algoritme, udarbejdet et stort selectscript igennem EF, som indeholdte mange join- og unions, i alt ca. 280 linjer. Efterfølgende prøvede vi at optimere på dette script, og prøvede derfor at loade hver enkelte table med simple select scripts (uden joins eller where), hvorefter EF sammensatte forbindelserne. Dette reducerede tiden fra ca. 38 sek til under 1 sek. Hvilket vi var ret overrasket over.

Bilag

Bilag 1: Mockup

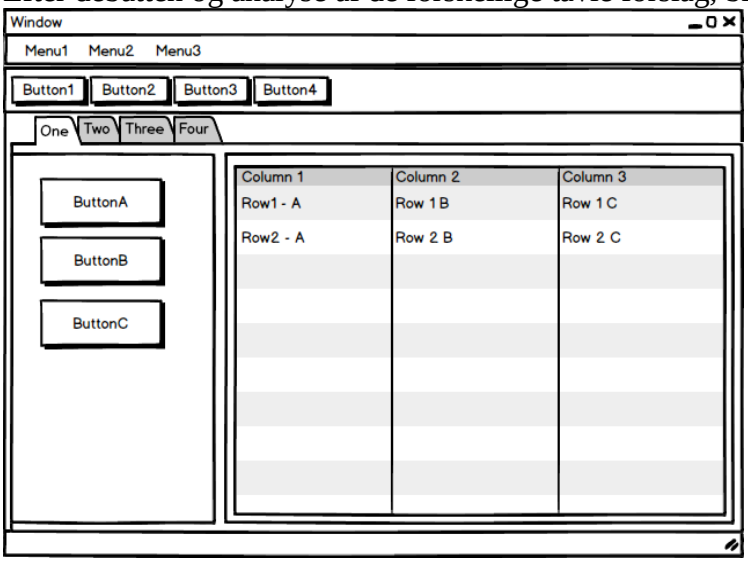
Administrator

Vi havde en debat omkring designet for de dele af system, som havde brug for et design. Under denne proces blev der skitserede et par enkelte mockups på tavlen, hvor resten af gruppen kom med input til mockupet.



Billederne viser henholdsvis to mockups, men det højre er en videretænkning af det første.

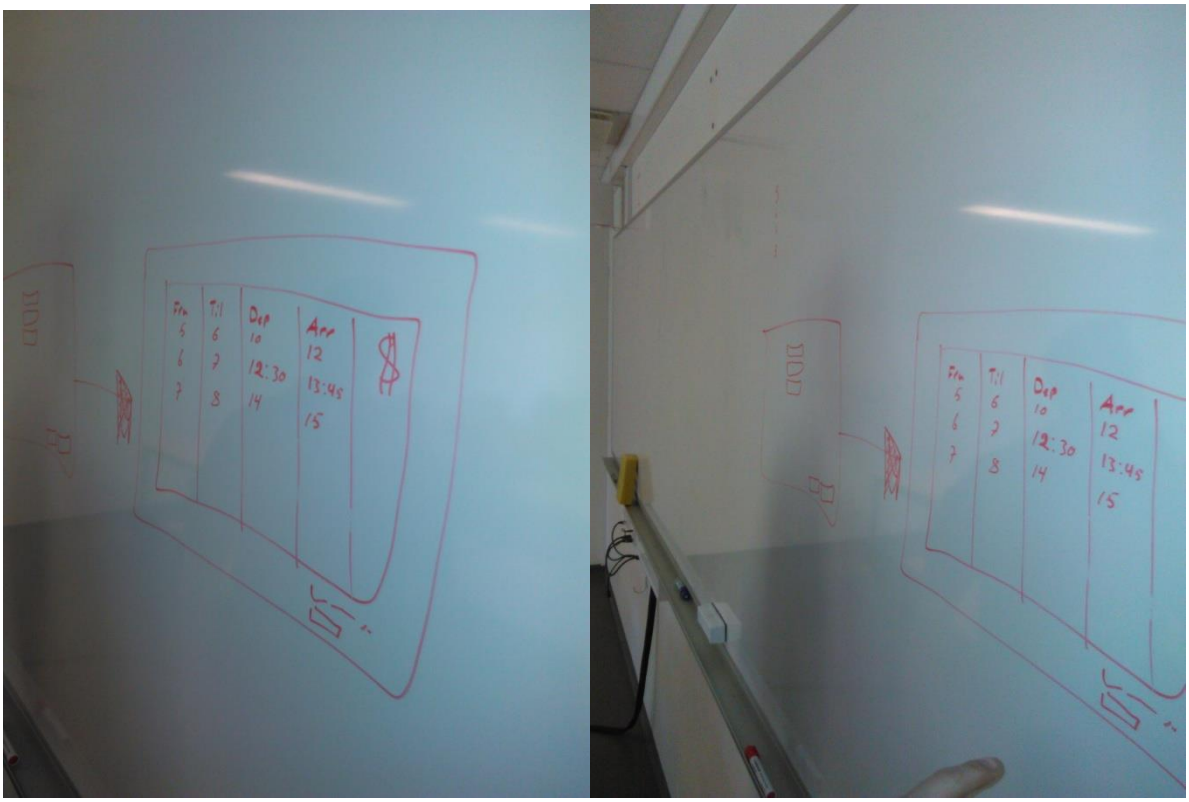
Efter debatten og analyse af de forskellige tavle forslag, blev en mere formel udgave udviklet:



Dette layout giver en god struktur over programmet og giver umiddelbart adgang via tabs til de forskellige funktionaliteter.

Web-Klienten

Dette afsnit omhandler Mockup af Web-Klienten, som skal give kunder mulighed for at søgning / fremvisning af rejser. Ligesom Administrator designet valgte vi at tage endnu en ”tavle debat”, og udvikle Mockupen, ud fra en sådan debat.



På billederne er der skitserede planen for hvordan en rejse skal vises til kunden. Der er ikke lavet en egentlig mockup af web-klienten, da den vil ligne meget eksisterende løsninger.

Bilag #2: Orginal Fully Dressed

Use-case	Planning a route	
Actor	User	
Frequency	Peak load: 0-200 an hour; must be scalable.	
Pre	Users exist and are logged on to the system. The network of flight routes exists.	
Post:	A flight route has been chosen with a maximum of 3 stops.	
	1. Enter starting point, time for the flight, and destination.	
	2. The user chooses criteria for the optimal route. Enters “Planning a route”.	3. The system calculates possible routes and the three most optimal routes are shown.
	4. The user chooses one of them.	5. The system shows available seats on the flight.
	6. The user selects seats on the flight.	
	7. The user enters “Make reservation”.	8. The system reserves tickets and system updates available seats on the respective flights.
Alternative flows:	4a. The user rejects all possible routes. 8a. Another reservation have been made of the same	

Bilag #3: Hændelsestabel (TO BE)

Hændelse (TO BE)	**Use case**	**Step i use case**	**Aktør**
Aktøren opretter, retter, sletter eller indhenter information om kunder	Kunde CRUD	- Opret kunde - Se kunde - Ret kunde - Slet kunde	Kunde, Administrator
Aktøren opretter, retter, sletter eller indhenter information om et fly	Fly CRUD	- Opret fly - Se fly - Ret fly - Slet fly	Administrator
Aktøren opretter, retter, sletter eller indhenter information om et rute	Rute CRUD	- Opret Rute - Se Rute - Ret Rute - Slet Rute	Administrator
Aktøren opretter, retter, sletter eller indhenter information om et lufthavn	Lufthavn CRUD	- Opret Lufthavn - Se Lufthavn - Ret Lufthavn - Slet Lufthavn	Administrator
Aktøren gør noget	Opret Reservation		
Aktøren retter, sletter eller indhenter information om en reservation	Reservation RUD		
Aktøren opretter, retter, sletter eller indhenter information om en administrator	Administrator CRUD	- Opret Administrator - Se Administrator - Ret Administrator - Slet Administrator	Administrator
Aktøren opretter, retter, sletter eller indhenter information om en afgang	Afgang CRUD	- Opret Afgang - Se Afgang - Ret Afgang - Slet Afgang	Administrator

Tabel 2.2.2 - "*TO BE*" hændelsestabellen

Bilag #4: Udvælgelse af Klasser

I denne del af designfasen kigges der på kandidater til klasser, som vil give mening for systemet. Den nedenstående tabel er udarbejdet ud fra de tidligere identificerede use cases og giver et samlet overblik over hvad vi har valgt at medtage.

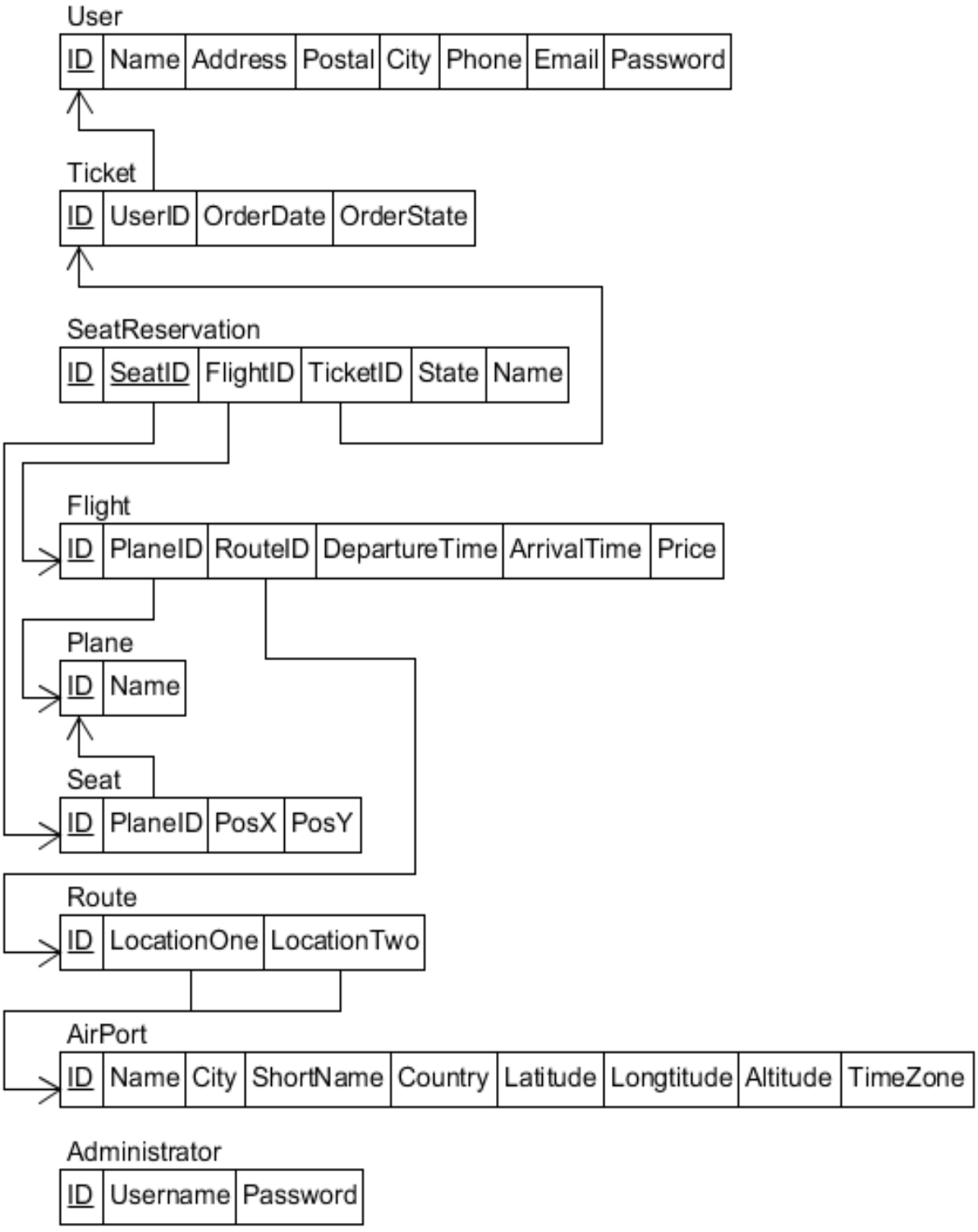
Kandidat til klasse:	**Vurdering:**	**Medtages:**
User	Klasse til håndtering af kundeoplysninger	Ja
Administrator	Klasse til håndtering af administrations oplysninger og rettigheder	Ja
Ticket	Klasse til den overordnede håndtering af reservation for den enkelte kunde	Ja
SeatReservation	Klasse til håndtering af sæde reservationen	Ja
Flight	Klasse til håndtering af afgange og dertil hørende information	Ja
Plane	Klasse til håndtering af fly information	Ja
Seat	Klasse til håndtering af sæde information, placering m.m	Ja
Route	Klasse brugt i sammenhæng med AirPort til håndtering af ruten mellem lufthavne	Ja
AirPort	Klasse til håndtering af lufthavnsinformation.	Ja

Bilag #5: Brief Use Case Beskrivelse

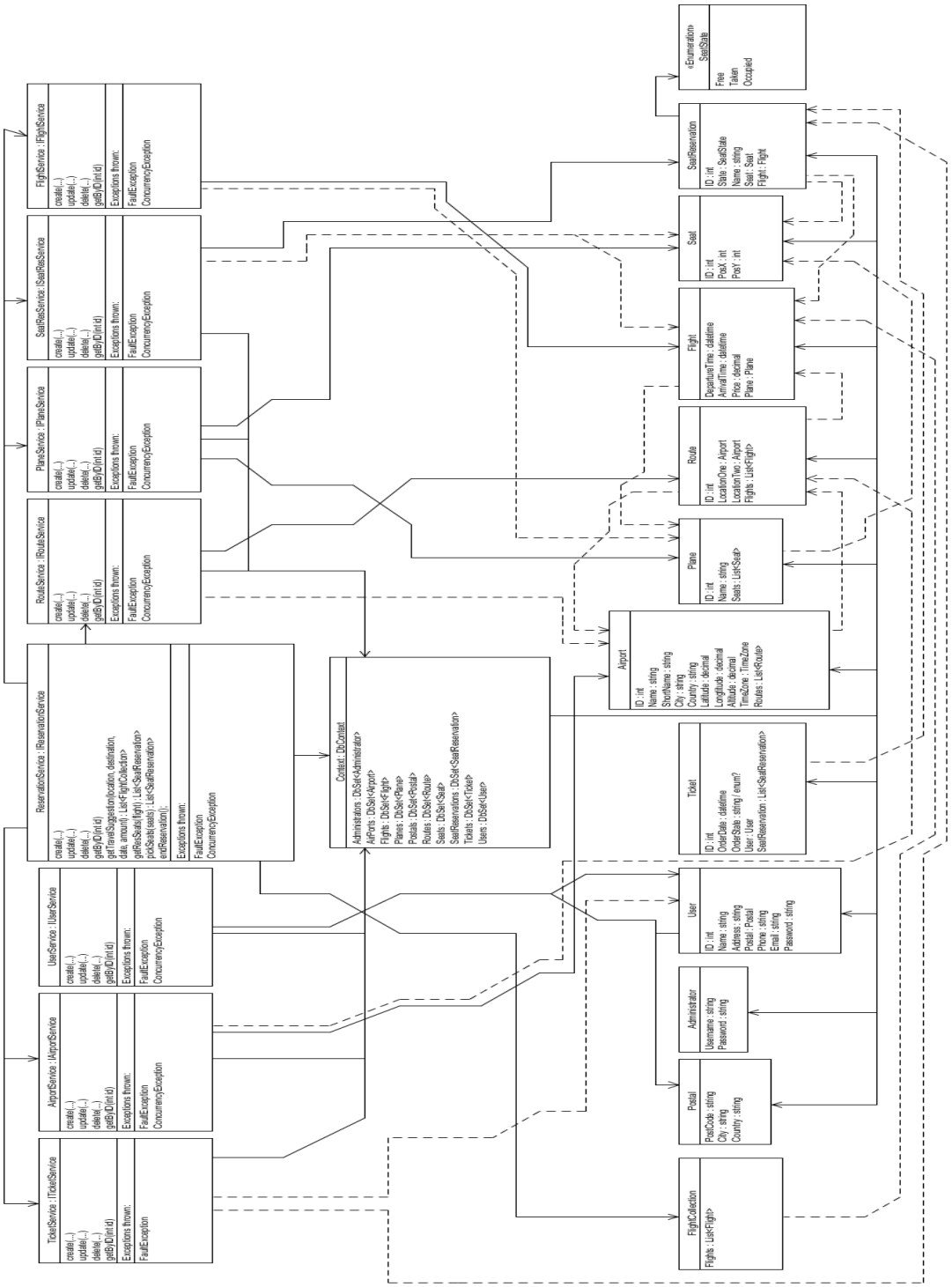
Dette vil være en oversigt over de forskellige use cases, men meget kort beskrivelse.

Use case: Kunde CRUD ● Kunde CRUD bruges af en Administrator eller en Kunde til at oprette, rette, slette eller se kunder.
Use case: Fly CRUD ● Fly CRUD bruges af en Administrator til at oprette, rette, slette eller se Fly
Use case: Rute CRUD ● Rute CRUD bruges af Administrator til at oprette, rette, slette eller se en Rute mellem 2 destinationer.
Use case: Lufthavn CRUD ● Lufthavn CRUD bruges af en Administrator til at oprette, rette, slette eller se en Lufthavn.
Use case: Reservation RUD ● Reservation RUD bruges af en Kunde eller en Administrator til at rette, se eller slette en Reservation.
Use case: Administrator CRUD ● Administrator CRUD bruges af en Administrator til at oprette, rette, se eller slette en Administrator.
Use case: Afgang CRUD ● Afgang CRUD bruges af en Administrator til at oprette, rette, se eller slette en Afgang

Bilag #5: Relationel model



Bilag #6: Design klassediagram

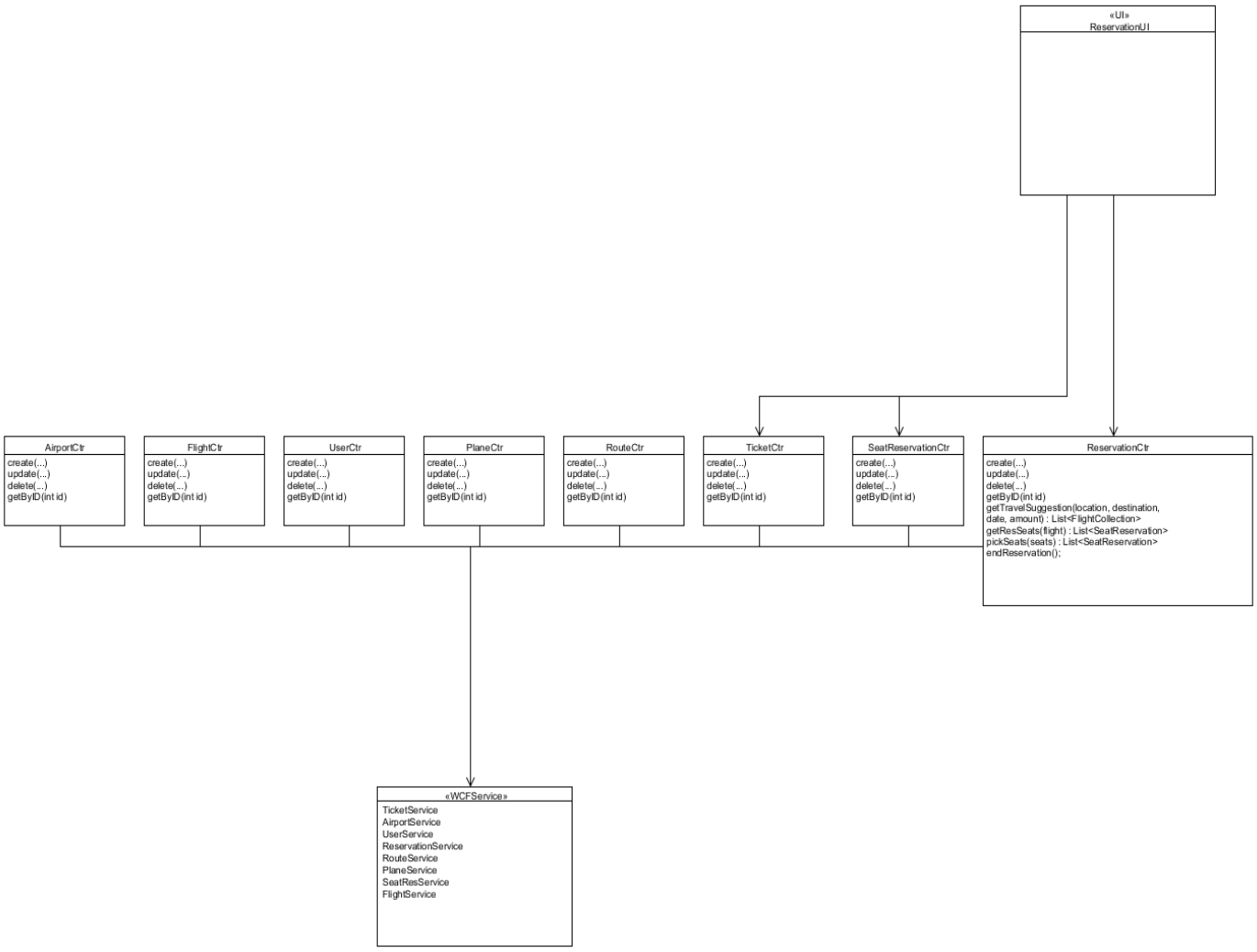


Designklassediagram: Service

Diagrammet er inddelt i to dele for at gøre det nemmere at vise, hvordan system skal se ud. Første del viser opbygningen af servicen, den anden del viser service interaktion med den grafiske brugerflade.

Dette diagram viser forholdet mellem klasserne og services. Hver service står for CRUD funktionalitet for hver klasse.

Navnene på services er logisk angivet i forhold til hvilket klasse de repræsentere. Nogle service håndtere CRUD funktionalitet på kryds af de andre services. F.eks bliver sæderne tilknyttet flyet når der oprettes et fly. Dette sker i henhold til aggregeringsstrukturen fra vores domænemodel.



Designklassediagram: Administrator

Diagrammet viser oversigten over controllere. Den grafiske brugerflade (UI'en) sender information fra brugeren til den rette controller. For hver tilsvarende service er der en controller som kalder en MainService, som så sender information videre til den rette service i det foregående diagram.

Bilag #7: Operationskontrakter

getTravelSuggestion

Operation: getTravelSuggestion(location, destination, date, amount);
Use Case: Flight Reservation

Precondition:
- Flight, Route, AirPort, Plane, Seat exist

Postcondition:
- Multiple instances flight of Flight is created
- flight.DepartureTime, flight.ArrivalTime, flight.Price, Flight.routes is assigned values

getSeats

Operation: getSeats(flight);
Use Case: Flight Reservation

Precondition:
- Flight, Route, AirPort, Plane, Seat exist

Postcondition:
- Multiple instances seat of Seat is created
- seat.ID, seat.PosX, seat.PosY is assigned values
- Multiple instances seatRes of SeatReservations is created
- seatRes.State, seatRes.Seat is assigned values

pickSeats

Operation: pickSeats(seats);
Use Case: Flight Reservation

Precondition:
- Flight, Route, AirPort, Plane, Seat exist
- SeatReservations for selected seats, does not exist

Postcondition:
- Multiple instances seatRes of SeatReservation is created
- seatRes.State, seatRes.Name is assigned values from *seats*
- instances ticket of Ticket is associated with seatRes

Bilag #8: Normal Former

- Normal form 1: Alle attributter skal være enkle og ikke sammensatte
- Normal form 2: Der må ikke være partielt funktionelle afhængigheder. Delvise afhængig af primærnøglen er derfor ikke tilladt.
- Normal form 3: Der må ikke eksistere transitive funktionelle afhængigheder. Alle attributter skal være afhængig af primærnøglen og ikke af en anden attribut som er afhængig af primærnøglen.