EASJ

Hovedopgave

EveryBooking

Kim Lindhardt & Rasmus Ketelsen

07-01-2016

Sider: 75 (Inklusiv forside)
Ord: 19.882
Tegn (Uden mellemrum): 102.756
Tegn (Med mellemrum): 122.267

Normalsider af 2.400 tegn med mellemrum: 51

Indholdsfortegnelse

1 Introduktion

I dag er der stadig mange, der manuelt holder styr på deres bookinger ved at skrive ned på papir når der bliver lavet en booking over telefonen, og de bookingsystemer der allerede eksisterer, er oftest begrænsede til en bestemt branche.

Vi vil derfor gerne udvikle et generelt bookingsystem der henvender sig til både restauranter, frisører, natklubber, massører og lignende. Et system der gør det nemt og overskueligt for virksomheden at håndtere og se bookinger, vagtplaner og diverse statistikker.

Vi har også selv prøvet at skulle bestille tid og være irriteret over, at der ikke er mulighed for online booking, eller at den online booking der er til stede ikke er særlig brugervenlig. Derfor vil vi også gerne lave en brugervenlig løsning til kunderne så de nemt kan booke online.

2 Vision

Vi vil gerne udvikle et generisk booking system som skal kunne håndtere alle typer af bookinger og med rig mulighed for at kunne tilføje flere typer af bookinger.

For at omfanget af opgaven ikke bliver alt for omfattende, har vi valgt kun at fokusere på frisører. Vi vil gerne have et stabilt bookingsystem med en driftsikker server arkitektur, og for at opnå dette er det vigtigt at vi ikke bruger alt tiden på at inddrage for mange brancher.

Vi ser at måden vores produkt vil blive brugt på er, at for frisøren er det muligt at kunne håndtere bookinger og vagtplaner samt se statistikker via en web grænseflade eller en app.

Vi vil implementere funktioner som, tilføj/opdater/slet bookinger, hvor man under tilføjelse og opdatere kan indtaste kundens oplysninger samt dato og tidspunkt og hvilket produkt kunden ønsker og hvilken frisør som skal servicerer kunden, vi vil også implementere en SMS service hvor kunden kan få en SMS notifikation hvis bookingen opdateres eller slettes.

Vi vil gøre det let for frisøren at have en oversigt over alle bookinger enten på dagen, ugen eller måneden, vi vil også implementere statistik så frisøren kan se hvor mange bookinger de har og på hvilke dage men også se hvor deres kunder oftest kommer fra om det er bookinger fra web, telefon eller om de kommer ind fra gaden.

Frisøren skal også have adgang til en vagtplan hvor man kan se hver af de ansatte, hvornår de er på arbejde og har fri, her skal det også være muligt for en frisør at melde sig syg og systemet skal i den sammenhæng kunne se om det er muligt at flette den syges bookinger ind i en andens frisørs tidsskema.

Det skal også være muligt at kunne booke tid direkte fra frisørens egen hjemmeside.

3 Problem definition

Frisører synes det er besværligt at skulle håndtere en almindelig papir kalender for alle deres bookinger, den kan godt blive lidt uoverskuelig når der er flere frisører med bookinger.

Det er også et problem hvis en frisør bliver syg og manuelt skal finde ud af hvilke bookinger som passer ind i de andre frisørers tidsskema og til de bookinger hvor der ikke er plads skal frisøren ringe og meddele at deres booking må udskydes.

Frisørerne kunne derfor godt tænke sig et let tilgængeligt IT system som kan hjælpe dem med at håndtere alle deres bookinger, vagtplan og en let måde at håndtere sygdom og kommunikation med kunderne.

Dette leder frem til tre hovedspørgsmål:

- Hvordan er det muligt at lave en skalérbar server arkitektur, uden et Single Point of Failure, der selv kan distribuere arbejdet ud på servere med mindre kapacitet?
- Hvordan kan vi udvikle et generelt bookingsystem der kan håndtere forskellige typer af bookinger?
- Hvordan kan vi have kommunikation mellem en hjemmeside og et eksternt system?

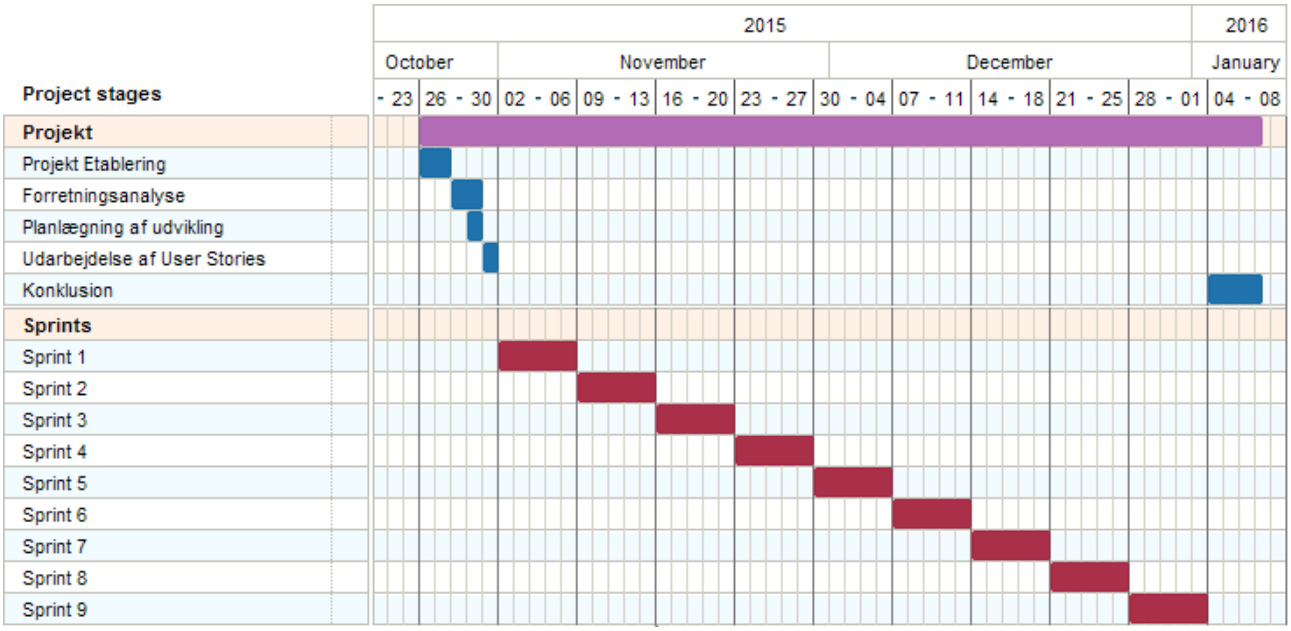
4 Aktiviteter

- Studere litteratur om skalerbar server arkitektur.
- Beskriv skalerbar server arkitektur i teori og praksis.
- Studere litteratur om kommunikation mellem to systemer.
- Beskriv kommunikation mellem to systemer i teori og praksis.
- Udføre et systemudviklingsprojekt med fokus på skalerbar server arkitektur.
- Lav en forretningsanalyse

4.1 Gantt Chart

For at skabe et overblik over projektets forløb, har vi lavet et Gantt Chart, som er et diagram der viser aktiviteter over tid. Gantt diagrammet vil vi opdatere løbende hvis der forekommer ændringer.

Til venstre er en liste af projektets aktiviteter, i toppen er der en tidslinje og i skemaet er hver aktivitet repræsenteret af en farvet linje, der viser hvornår aktiviteten starter og slutter.



5 Metoder

5.1 Systemudviklingsmetoder

Vi har kigget på de udviklingsmetoder vi har fået erfaring med under uddannelsen og vurderet deres fordele og ulemper efter hvordan de bedst kan være med til at hjælpe os med projektstyring, planlægning og udvikling.

Vi har valgt at bruge Agil udvikling hvor vi gør brug af dele fra metoden eXtreme Programming(XP), dertil gør vi også brug af nogle artefakter fra Unified Process, og til projektstyring gør vi brug af SCRUM.

Vi syntes Agile SCRUM og XP er passende fordi:

- Vi ikke kender til alle projektets krav til at starte med og kommer derfor til at tilføje, fjerne og ændre krav undervejs.
- Scrum og XP giver et godt overblik over projektet med brug af user stories, product-/sprint backlog og estimering.
- Det giver fokus på integration og test af software.

Vi vælger at gøre brug af nogle artefakter fra Unified Process for at:

- Hjælpe os med at designe og dokumentere de mere kritiske dele af systemet.

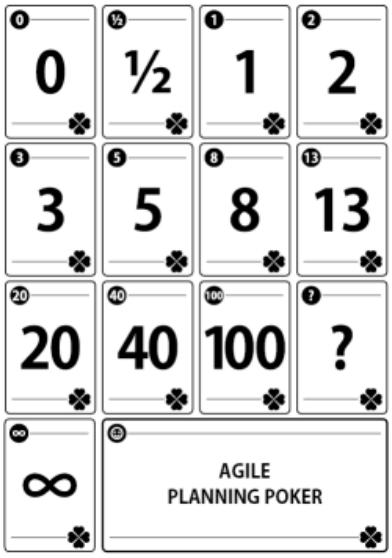
Vi har valgt kun at beskrive nogle af de praksis vi har valgt eller valgt fra. Vi har valgt dem vi synes der er interessante, og som mangler lidt uddybelse.

5.1.1 eXtreme Programming

eXtreme Programming er en agil udviklingsmetode der kendetegnes ved at være åben for ændringer løbende, og på sine 12 grundlæggende praksis.

- Planning Game, hvor forretning og udvikling samarbejder om at skabe værdi for kunden så hurtigt så muligt.
- Små udgivelser, tidligt og hele tiden med få features hver udgivelse.
- System Metafor, der giver teamet en navngivningskonvention der er nem at huske.
- Simpelt design, der løser opgavens krav og intet mere, da kravene nok ændres hyppigt.
- Testing, sker før og efter kode skrives. Unit tests før og Acceptance tests efter.
- Refaktorering, sker hyppigt undervejs for at forenkle koden.
- Parprogrammering, sker under al udarbejdelse af kode.
- Fælles kode ejerskab, så alle udviklere har adgang til at arbejde på alle dele af koden.
- Kontinuerlig integration, så alle ændringer integreres undervejs.
- Overkommelige arbejdstider, hvor overarbejde ikke er tilladt.
- "On-site" kunde, så udviklere altid har mulighed for at komme i kontakt med en bruger af systemet.
- Fælles kodestandard, så alle koder efter samme standarder.

5.1.1.1 Planning poker



Vi vil gerne gøre brug af planning poker, da vi syntes det hjælper os med at estimere hvor meget arbejde vi kan nå pr. sprint, det hjælper os også med at lave et burndown chart så vi kan se hvor langt vi er nået overordnet.

Planning poker fungerer således at når første udkast til product backlog er lavet kan vi begynde på planning poker. I planning poker tager scrum master en user story og læser den op for udviklingsholdet, hvert medlem sidder med en række kort hvor på hvert kort er der et antal story points.

Når en user story er læst op skal hvert medlem vurdere hvor mange story points de hver især mener hører til den gældende user story, hvis der er større uenighed betyder det at holdets medlemmer har fået en forskellige forståelse af den user story og hvad den indebærer, så skal man som hold snakke om hvad den user story betyder, går ud på og teknisk indebære og så skal man igen vende kortene og se om holdet er kommet til en enighed, hvis holdet er enig kan man gå videre til næste user story.

Når man som team skal i gang med planning poker, skal man først definere hvad story points er. Det gør man ved at finde en user story som alle medlemmer af teamet kan forholde sig til, og så blive enige om hvor mange story points der hører til den user story.

Når teamet så går i gang med planning poker, har de et fælles udgangspunkt til at bestemme antal story points til andre user stories.

5.1.1.2 Acceptance test

For dette produkt er det os som er product owner vi føler derfor ikke at det er nødvendigt for os at holde en acceptance test i slutningen af hvert sprint fordi at når vi selv er product owner og udvikler så kan vi undervejs hurtigt se hvad som ikke fungerer for os og ændre det.

5.1.1.3 Unit Testing

I hvert sprint vil vi udarbejde unit tests for den software vi laver i det pågældende sprint, det hjælper os med konstant at kunne holde styr på at softwaren virker efter hensigten.

Unit tests hjælper os også med at hvis vi skal refaktorere noget software så behøver vi ikke manuelt at skulle teste at softwaren virker men vi kan nøjes med at køre softwaren igennem vores unit tests.

5.1.1.4 Test Driven Development (TDD)

Test Driven Development er en udviklingsmetode hvor man skriver tests der definerer den ønskede forbedring eller funktionalitet, før man går i gang med at implementere den. Når testen så er skrevet og den fejler, producerer man den minimale mængde kode der skal til for at gå igennem testen. Til sidst, når koden er gået igennem testene, refaktorerer man koden til en acceptabel standard.

Test Driven Development kan på lang sigt gøre udviklingsprocessen hurtigere, men kræver en del arbejde i starten der kan få udviklingen til at føles langsom[1]. TDD er især en tidskrævende process når man ikke er vant til at arbejde ved at lave tests som det første[2], og eftersom at vi ikke har arbejdet med TDD før og vi ikke føler at vi har tid nok til at sætte os ordentligt ind i TDD, har vi valgt ikke at følge Test Driven Development.

TDD kan også medføre meget refaktorering da man kun tænker på at testen skal lykkedes nu og her, og ikke tænker på fremtidige implementeringer. Det kan også give en del refaktorering, og være tidskrævende, hvis designet af systemet ændres oftest så man skal tilbage og skrive sine tests om, og måske ende ud med at have brugt tid på at skrive tests til features der bliver fjernet undervejs.

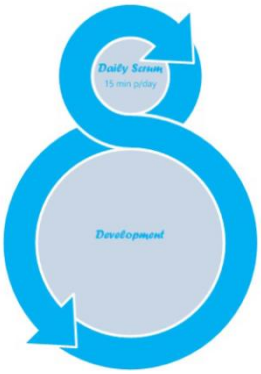
5.1.1.5 Fælles kodestandard

Igennem projektet vil vi opretholde en fælles kodestandard. Vi vil sørge for at tabulering, mellemrum og parenteser sættes ens gennem hele projektet, vi vil holde en ens navngivning af klasser, metoder osv.

En fælles kodestandard gør det nemmere at vedligeholde, refaktorere og udvide kode på tværs af teamet, og gør generelt koden mere overskuelig i projektet.

[1] https://leantesting.com/resources/test-driven-development/
[2] http://www.amzur.com/pros-and-cons-of-using-test-driven-development-for-web-applications-built-with-ruby-on-rails-ror/

5.1.2 Scrum



5.1.2.1 Daily Scrum

I starten af hver dag holdes der et Daily Scrum på omkring 15 minutter, på mødet deltager udviklingsholdet samt scrum master, her får alle mulighed for at fortælle resten af holdet hvad de har arbejdet med den foregående dag, hvad de skal arbejde med i dag og også eventuelle problemstillinger og smarte løsninger.

Daily Scrum giver holdet som helhed et indblik i hvor langt alle er nået siden det sidste Daily Scrum møde, og om der er nogle problemstillinger der forhindre det planlagte arbejde.

5.1.2.2 Sprint

Et sprint er en periode typisk på 1-4 uger, hvor man fokuserer på at implementere arbejdsopgaver fra sprint backloggen. Formålet med et sprint er at ende ud med working software der giver værdi for kunden.

Hvert sprint starter med planlægning af sprintet, hvor product owner, scrum master og scrum teamet mødes for at diskutere de højest prioriterede user stories i product backloggen. Efterfølgende bliver de aktuelle user stories opdelt i detaljerede tasks, og en udvikler påtager sig ansvaret for de enkelte tasks. Når alle tasks er delt ud, estimerer udviklerne hver for sig for at sikre sig at de ikke har påtaget sig for meget arbejde.

Når sprintet er i gang afholder man daglige scrum møder, hvor hver enkelt teammedlem fortæller hvad de har lavet dagen før, hvad de skal lave på dagen, og om der er noget der forhindre dem i at komme videre. Daglige scrum møder er med til at give alle på teamet et indblik i hvad der er lavet og hvad der mangler.

Et sprint afsluttes ved at holde et scrum review møde hvor en demonstration af det working software man er nået frem til i løbet af sprintet vises til product owner og andre interesserede. Efterfølgende mødes scrum teamet til et sprint retrospektiv møde hvor der kigges på hvordan sprintet er gået, og om der er noget der skal ændres inden næste sprint.

5.1.2.3 User Story

Features kan beskrives som User Stories.

En User Story fortæller os hvem der udfører opgaven og en kort beskrivelse af opgaven set fra perspektivet af personen som udfører opgaven.

Opbygningen af en user story kan se således ud:

Som <hvem udføre opgaven>, vil jeg gerne kunne <udføre en opgave> så jeg <kan nå et mål>

En user story kan også have vedlagt acceptkriterier, som fortæller os hvornår funktionaliteten på en user story er accepteret.

Et godt eksempel på en user story er:

Som frisør,
skal jeg kunne tilføje bookinger i min kalender selv,
så jeg kan tage imod folk der ringer eller kommer ind i min salon fra gaden.

Acceptkriterie:

- Det skal være muligt at vælge om bookingen er sket over telefon eller ansigt til ansigt.

Her kan vi se hvilken person der udfører opgaven, hvilken arbejdsopgave det er og hvorfor arbejdsopgaven skal udføres. Vi kan også se at for at denne User Story kan blive accepteret er det nødvendigt vi kan vælge hvordan bookingen er sket.

User stories skrives i samarbejde med product owner, og det er product owner der bestemmer hvordan de forskellige user stories skal prioriteres. Prioriteringen bliver gjort klar ved at product owner tildeler en risiko faktor til hver user story, der så fortæller os om en user story er nødvendig for at systemet kan anvendes, om det er en feature der skal implementeres men ikke er absolut nødvendig for at systemet kan anvendes, eller om det er en "nice to have" feature der skaber værdi for brugeren, men ikke nødvendigvis skal implementeres.

Vores user stories vil få tildelt risikofaktoren, low, medium eller high hvor low er "nice to have", medium skal implementeres men ikke absolut nødvendige, og high er absolut nødvendige for at systemet kan anvendes.

5.1.2.4 Product Backlog

Product Backloggen består af en liste af user stories og bugfixes skrevet af Product Owner, listen kan ændres af Product Owneren dvs. tilføje, ændre, fjerne og ændre prioriteringen af elementer på Product Backloggen.

I sprint planning mødes product owner, scrum master og scrum teamet for at diskutere de højest prioriterede user stories i product backloggen.

Scrum teamet vælger de højest prioriterede user stories som de har estimeret til at kunne nå i det kommende sprint, når user stories er valgt bliver de frosset og derved kan product owner ikke ændre i de valgte user stories.

5.1.2.5 Sprint Backlog

I starten af et nyt sprint kigger scrum teamet på de øverste elementer i product backloggen og kigger på hvor lang tid hver feature/bugfix er estimeret til, og ser derefter på holdets arbejdseffektivitet.

Ved at se på features/bugfixes estimering og holdets arbejdseffektivitet kan vi finde ud af hvor mange vi kan tage ind i vores sprint backlog.

Når vi har fundet ud af hvilke user stories vi skal arbejde med i dette sprint så kan vi begynde at dele user stories op i tasks.

Måden vi finder ud af hvilke tasks som gemmer sig i en user story er ved at, vi kigger på user storyen og spørger os selv "Hvilke software komponenter skal vi implementere eller ændre for at fuldføre denne user story?"

Ved at spørge os selv på den måde så kan vi se på den user story fra software verdenens perspektiv og gøre brug af vores erfaring fra software verdenen og se hvilke tasks som skal laves for at fuldføre den user story.

5.2 Software sprog, Udviklingsværktøjer & Servere

5.2.1 Softwaresprog

5.2.1.1 Backend – Servere

- C#
- PHP
- SQL
- JSON

Vores backend består af vores skalérbare server arkitektur, hjemmesiden og ikke mindst databasen.

På vores skalérbare servere har vi valgt at udvikle vores server applikation med C# da vi har arbejdet med det igennem uddannelsen og har et godt kendskab til det.

Vi har valgt at bruge PHP som backend for hjemmesiden da vi har en smule erfaring med det.

For at serverne skal kunne kommunikere med databasen gør vi brug af SQL som vi begge har erfaring med.

For at hjemmesiden skal kunne kommunikere med vores skalérbare servere, så har vi valgt at bruge JSON som data format, fordi vi har fået uddybende kendskab til det i vores praktikforløb. JSON er let at arbejde med og veldokumenteret, det er også let at fejlsøge i og læse for mennesker som vi føler er vigtigt for os.

Som alternativ kunne vi have brugt XML som har mange af de samme kvaliteter som JSON, eksempelvis er de begge ikke afhængige af afsenderen og modtageren, begge har en læsbar struktur for mennesker.

Vi valgte JSON fremfor XML fordi at JSON ikke bruger nær så meget markup kode for at beskrive den samme data, det syntes vi også gør JSON en tand nemmere at læse.

5.2.1.2 Frontend – Hjemmeside

- HTML
- CSS
- JavaScript

Vores frontend er en hjemmeside hvor frisøren kan håndtere bookinger og vagtplaner, og se statistikker.

5.2.2 Udviklingsværktøjer

Som udviklingsværktøjer gør vi brug følgende:

Visual Studio Online

Fordi vi har valgt at gøre brug af Agile SCRUM og kender til de forskellige processer vi skal igennem, har vi valgt at bruge Visual Studio Online som vores digitale arbejdsplads. Hos Visual Studio Online kan vi oprette alt vi skal bruge, team, user stories, backlog, sprint backlog, sprint, test cases, charts og ikke mindst kan vi også hoste vores kode så alt ligger samme sted. Kort sagt tilbyder Visual Studio Online os en komplet digital Agile SCRUM arbejdsplads.

MySQL Workbench

MySQL Workbench er et stort database værktøj specifikt for MySQL[3]. Normalt tilgås MySQL via terminalen. Men med MySQL Workbench har vi alle MySQLs faciliteter i et grafisk interface som gør det langt lettere for os at kunne administrere vores database.

GIT – Versionsstyring

Vi vil gerne gøre brug af GIT fordi det hjælper os med at kunne arbejde på hver vores issues men stadigt arbejder vi på samme kodebase.

GIT tilbyder også smarte funktioner for at kunne lave nye branches og afprøve nye ideer uden at det behøver at blive lagt over på main branch. Eksempelvis kan vi have to branches Main Branche som er køreklare versioner af softwaren og så en Development branch hvor hele udviklingen foregår. Dertil kan vi efter nødvendighed oprette en branch mere hvis vi gerne vil prøve en ny ide uden at påvirke development branche.

Servere

Vi lejer tre servere igennem cloud host DigitalOcean.

Hver server er udstyret med følgende konfiguration.

- CPU: 1 Core af 1.8 - 3.0 Ghz
- RAM: 512 MB
- Disk: 20 GB SSD
- OS: Ubuntu Server

[3] https://www.mysql.com/

6 Delkonklusion

Vi har nu kigget lidt på nogle af de problemstillinger der er omkring online booking, og vores vision for hvordan vi kan løse dem med et generelt bookingsystem.

Fremadrettet vil vi arbejde på, at besvare de tre hovedspørgsmål vi har stillet os selv i vores problemdefinition. For at besvare spørgsmålene vil vi følge en række aktiviteter, og til at udføre disse aktiviteter gør vi brug af dele fra Agil udvikling, eXtreme programming, Unified Process og SCRUM.

Vores projekt er nu etableret og vi har opstillet nogle retningslinjer for hvad vi skal arbejde hen imod, og hvilke aktiviteter vi skal udføre for at nå målet. Som det næste vil vi lave en forretningsanalyse hvor vi vil se nærmere på hvordan vores position på markedet er.

7 Forretningsanalyse

I dette afsnit vil vi analysere og beskrive markedet for vores booking system. Vi vil kigge på hvilket marked og hvilken kundegruppe vores system henvender sig til, og hvilke produkter der allerede findes på markedet. Når vi ved hvilke produkter der allerede eksisterer på markedet, vil vi kigge på om der er eventuelle konkurrenter i blandt.

Når vi kender eventuelle konkurrenter, vil vi kigge på hvilke muligheder vi har for at markedsføre vores booking system.

7.1 Kunder

Vores produkt henvender sig til markedet for bookinger, hvor de fleste af vores konkurrenter specificere bookinger til et bestemt område så vil vi ramme alle typer af bookinger i et og samme produkt.

Vores kunder vil derfor bestå af mange forskellige brancher så som. Frisører, massøre, restauranter, fitness hold, konference rum m.m. Hvis der er noget som kan bookes så skal vores produkt kunne håndtere det.

7.2 Konkurrentanalyse

Når vi ser på hvilke konkurrenter som eksisterer, ser vi på om de tilbyder samme værktøjer som vi har tænkt os, og hvad prisen er for deres system.

Efter at have undersøgt markedet for at se hvilke bookingsystemer som allerede eksisterer, har vi fundet en del som har nogle af de samme løsninger, men vi har udvalgt tre virksomheder som tilbyder stort set det samme, bare på lidt forskellige måder. For at finde frem til om vores forretningsmodel/strategi kan udfordre eksisterende konkurrenter, vil vi gennemgå de tre virksomheder vi har udvalgt.

Hairtools ApS[4]

Hairtools er en dansk virksomhed der tilbyder en række IT løsninger, fx. online booking, bestillingsbog(vagtplan), lønberegning, lagerstyring og statistikker. Hairtools har også forskelligt hardware bl.a. bonprinter, kasseapparat, kortterminal og stregkodelæser til lageroptælling. De tilbyder kun løsninger til frisører og kosmetologforretninger.

Hairtools har gjort så kunder kan vælge imellem to abonnementer.

Hairtools Light, er deres lille pakke hvor kunden får bestillingsbog, kundekartotek, onlinebooking og muligheden for at sende nyhedsmail, påmindelser og lignende på e-mail eller sms. Prisen for Hairtools Light er 295 kr. om måneden og 0,79 kr. pr. sms.

Hairtools store pakke hvor kunden får, alt fra hairtools light, plus kassesystem med værktøjer som lagerstyring og statistikker. Denne løsning koster 495 kr. om måneden og 0,79 kr. pr. sms. Desuden er det påkrævet at købe en bonprinter til denne løsning, som koster 2.395 kr.

Udover abonnement er der også et engangsbeløb på opstart og undervisning, på 2.000 kr. eller 4.000 kr. alt efter om det er telefonisk eller personligt.

TimeCenter[5]

TimeCenter tilbyder et online bookingsystem til skønhed, fysioterapi, sundhed, sport, og mødebooking. Bookingsystemet indeholder kalender, kundekartotek, historik, statistik, journaler og muligheden for at sende nyhedsbreve og påmindelser via e-mail og sms.

TimeCenters bookingsystem er abonnementsbaseret, og der er tre forskellige pakker, hvor hver pakke indeholder et antal af "kalendere". Meningen er at man skal bruge én kalender for hver person eller ressource der skal kunne bookes.

Pakker:

One, indeholder 1 kalender til 249 kr. om måneden og kan prøves gratis i 30 dage.
Pro, indeholder 10 kalendere til 499 kr. om måneden og kan prøves gratis i 30 dage.
Premium, indeholder 50 kalendere til 799 kr. om måneden og kan prøves gratis i 30 dage.

Uanset pakke er prisen pr. SMS 0,99 kr.

[4] www.vierhairtools.dk
[5] www.timecenter.com

Admind A/S[6]

Admind tilbyder fire forskellige IT løsninger som dækker over forskellige områder inden for butiks og webshop markedet.

- *Admind Care*
  Er vores direkte konkurrent, de tilbyder samme funktioner som vi gør, kalender, booking, online booking, statistik, sms påmindelser, og udover det tilbyder de også andre funktioner som vi ikke har. Kundekartotek, behandlingsbeskrivelser, kassefunktion, salgs posteringer, løn og provisionsberegning, finansposter og regnskabsprogram.

Admind Care Light pr. md.	Admind Care u. booking pr. md.	Admind Care m. booking pr. md.	Admind Care Komplet pr. md.
295,-	395,-	495,-	595,-
• Kalender	• Kalender	• Kalender	• Kalender
• Kundekartotek	• Kundekartotek	• Kundekartotek	• Kundekartotek
• Behandlingsbeskrivelser	• Behandlingsbeskrivelser	• Behandlingsbeskrivelser	• Behandlingsbeskrivelser
• Online Booking	• Online Booking	• Online Booking	• Online Booking
• Kassefunktioner	• Kassefunktioner	• Kassefunktioner	• Kassefunktioner
• Salgsposteringer	• Salgsposteringer	• Salgsposteringer	• Salgsposteringer
• Statistikker	• Statistikker	• Statistikker	• Statistikker
• Løn-og provisionsberegning	• Løn-og provisionsberegning	• Løn-og provisionsberegning	• Løn-og provisionsberegning
• Udlæsning af finansposter	• Udlæsning af finansposter	• Udlæsning af finansposter	• Udlæsning af finansposter
• Regnskabsprogram	• Regnskabsprogram	• Regnskabsprogram	• Regnskabsprogram

[6] www.admind.dk

Deres funktionaliteter er delt op i fire pristabeller af 295, 395, 495, 595.
Der bliver tilføjet flere funktionaliteter jo dyre abonnementet er, dog finder vi det morsomt at den billigste til 295 giver muligheden for Online Booking men den næste til 395 tilføjer flere funktionaliteter dog uden Online Booking så hvis man gerne vil have flere funktionaliteter inklusiv Online Booking så skal man som minimum vælge abonnementet til 495.

Admind har også en venteliste, hvis jeg som kunde gerne vil klippes kl 13:00 men som desværre er optaget så kan jeg tilmelde mig ventelisten. Hvis kunden der har kl 13:00 melder afbud så kan frisøren slå op i ventelisten og tage kontakt til den kunde som sidder på ventelisten.

Admind tilbyder også at frisøren kan tilgå deres bookinger, kundekartotek m.m hvis nettet er nede. Dette tror vi dog kun er et halv salgstrick, vi kunne godt tænke os at se hvordan de synkronisere dataen hvis nettet er nede i et par dage.

- *Admind Retail*
  Er et komplet IT system for den almene forretning med integration af betalingsterminal, kasse, varelager, kunder med statistikker samt bogføring. Dette system er dog ikke en konkurrent til vores produkt.

- *Admind Webshop*
  Er et webshop system der gør det muligt for en butik at kunne sælge deres produkter online, dette er dog ikke et konkurrerende produkt til vores produkt.

- *Admind Office*
  Et bogholdnings system, dette har vi ikke undersøgt videre fordi det ikke er et konkurrerende produkt til vores produkt.

7.2.1 Konkurrentanalyse Konklusion

Efter at have undersøgt de tre konkurrenter nærmere, har vi fået et bedre billede af hvordan vores markedsføring/strategi vil fungere.

Hairtools bookingsystem er forholdsvis store inden for frisører og kosmetologer. De har været på markedet siden 2001, og er i brug i mere end 1.200 frisørsaloner og kosmetologforretninger dagligt[7]. Vores mulighed for at konkurrere med Hairtools er ved at holde en lavere pris, hvilket vi synes at vi gør. Vi ville kunne konkurrere med deres lille pakke hvor vi kan tilbyde mere funktionalitet til en billigere pris. Vores største fordel er at Hairtools kun fokuserer på frisører og kosmetologer, da vi vil ramme et meget bredere marked.

TimeCenter er muligvis dem der har et system der ligner vores mest, de tilbyder stort set samme funktionalitet som vi gør, de har dog også kundekartotek og kundejournaler.

[7] http://www.vierhairtools.dk/hvem-er-vi/

Selvom TimeCenter er en stor konkurrent, tror vi på at vores forretningsmodel er mere attraktiv. Eftersom at TimeCenter kræver en kalender for hver medarbejder eller resource, bliver det hurtigt en del dyrere end det vi tilbyder. Allerede ved mere end én bruger/resource bliver deres system dobbelt så dyrt, så vi tror helt sikkert vi kan være med og konkurrere på abonnement prisen. Deres SMS pris synes vi også er meget dyr, med 0,99 kr. pr. SMS, hvor vi ville kunne gøre det til omkring den halve pris.

Admind er også en stor konkurrent med deres Admind Care løsning. Lidt ligesom TimeCenter har Admind også forskellige abonnementer, hvor de tilbyder forskellig funktionalitet til forskellige priser. Også her synes vi hurtigt det bliver dyrt i forhold til hvad vi vil tilbyde af funktionalitet til én pris.

Efter at have undersøgt vores konkurrenter har vi kunnet se en tendens til, at i et bookingsystem vil frisøren, kosmetologen eller lignende gerne have mulighed for at kunne gemme kundeinformationer, og have en tilhørende kundejournal.

Eftersom at vores konkurrenter tilbyder kundekartotek og kundejournaler, er vi kommet til den konklusion at vi også skal tilbyde vores fremtidige kunder den funktionalitet. Vi vil derfor gerne tilføje kundekartotek og kundejournaler som en del af funktionaliteten i vores bookingsystem.

Generelt er det et marked hvor vi kommer til at møde konkurrence, men vi tror på at vi har et stærkt produkt til en pris der er væsentlig bedre end hvad der ellers er på markedet.

7.3 Forretningsmodel

Baseret på informationen vi har fået fra vores konkurrentanalyse kan vi se at det vil være meget svært at få nye kunder fordi vores konkurrenter allerede er veletableret på frisør segmentet.

Vi skal derfor bruge en anden indgangsvinkel som kan vende nye frisører til vores produkt samt tilbyde allerede etablerede frisører så godt et tilbud at de skifter til vores og bliver vant til at bruge vores.

7.3.1 Fagforening & Forbund

Fagforeninger og forbund har gerne mange frisør medlemmer, dvs. en stor kundebase for os.

Vi kan derfor henvende os til frisør fagforeninger og forbund med det formål at fremvise vores produkt og tilbyde foreningen/forbundet at deres medlemmer kan få vores produkt gratis i en længere tidsperiode, med det resultat at foreningen/forbundet vil råde nye og etablerede frisører til at bruge vores booking system.

7.3.2 Reklamer

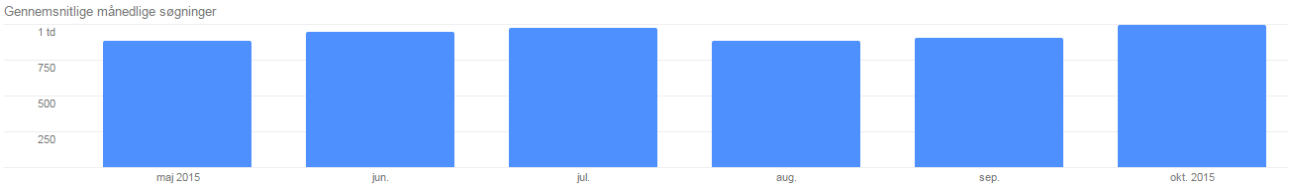
Vi har kigget på muligheden for at reklamere på Google inde for bestemte søgeord.

Vi har derfor lavet en analyse over hvor populære søgeordene er, hvor mange gange der bliver søgt på dem månedligt samt konkurrence og gennemsnittet for pris pr. klik.

Søgeordene vi har valgt er:

- frisør booking
- booking system
- bookingsystem
- online booking

I nedenstående billede kan vi se en oversigt over gennemsnitlige søgninger på de fire søgeord over de sidste seks måneder i Danmark.



I gennemsnittet er der mellem 900-1000 søgninger om måneden, derfor kan vi konkludere at der er stor interesse for booking systemer i Danmark.

I nedenstående billede kan vi se detaljer for de søgeord vi har valgt. Vi kan eksempelvis se hvor mange søgninger der er gennemsnitligt pr. månede pr. søgeord, vi kan også se konkurrenceniveauet pr. søgeord og ikke mindst kan vi se et foreslået bud som er pris pr. klik.

Søgeord (efter relevans)	Gns. antal månedlige søgninger maj 2015 -okt. 2015	Konkurrence	Foreslået bud
booking system	480	Højt	44,57 kr
online booking	320	Højt	30,05 kr
bookingsystem	90	Højt	29,32 kr
frisør booking	30	Højt	9,69 kr

Som vi også fandt frem til i vores konkurrent analyse kan vi se der er stor konkurrence, og priserne pr. klik er høje for en ny spiller på markedet uden en startkapital.

7.3.3 Abonnement

For nye kunder tilbyder vi en gratis prøveperiode på seks måneder, med det forbehold at hvis kunden ønsker at gøre brug af vores SMS service, så skal der betales for det forbrug kunden bruger.

Vi har valgt at tilbyde seks måneder gratis fordi vi er en ny spiller på markedet uden kunder, derfor skal vi tilbyde noget ekstraordinært ud over "bare" vores produkt. Formålet med det er at tiltrække nye kunder og lade dem bruge systemet i en lang periode så de bliver vænnet til systemet og får en god følelse af systemet.

Det er lidt samme princip som når man har fået en god service fra et firma så danner man sig et godt indtryk og bliver ved med at bruge det firma.

Hvis kunden ønsker at bruge vores produkt efter prøveperioden vil det koste 249 Kr,- om måneden inklusiv moms.

For at gøre brug af vores sms service er det ikke nødvendigt at købe et kvantum af sms'er på forhånd, vi syntes det er nemmest hvis vores kunde kan bruge de sms'er kunden skal bruge og så afregnes der samme dag hvor abonnementet skal fornyes. Eksempelvis hvis vores kunde bruger for 50 Kr,- sms'er så skal kunden på fornyelse datoen betale 299 Kr,-

Vi er dog indforstået med det betyder at betaling for sms forbruget er bagudbetalt og vores kunde derfor har mulighed for at kunne stoppe abonnementet og også vælge ikke at betale for det sms forbrug kunden har brugt, dog i den situation kan vi sende en opkrævning for sms forbruget til kunden.

7.3.4 SMS

Vi vil gerne tilbyde vores kunder, at kunne meddele deres kunder om ændringer af booking m.m.

For at vi skal kunne stå stærkere som produkt iblandt vores konkurrenter må vi affinde os med at kunne presse prisen ned som i sidste ende kan give os flere kunder. Vi har derfor undersøgt markedet for levering af SMS gateways.

En SMS Gateway tillader os at kunne implementere funktionalitet der gør vores produkt i stand til at kunne sende sms'er.

Vi har kigget på fire forskellige leverandører af SMS Gateways.

Den ene, Amazon SNS som tilbyder rigtig gode priser, helt ned til fem øre pr. sms, problemet er dog at servicen kun tilbydes i USA.

To andre leverandører kræver dog et månedligt abonnement og derefter betaler man efter forbrug, det er ikke det vi leder efter, vi vil have en leverandør hvor vi kun betaler efter forbrug.

Den sidste leverandør CpSMS gør det næsten muligt for os at betale efter forbrug, vi skal købe en pakke af sms'er, eksempelvis 1.000 sms'er for 300 Kr,-. Dvs. prisen pr. sms for os er 30 øre.

Vi kan ved at købe større sms pakker spare nogle øre pr. sms, men fordi vi slet ikke er nået til det stadie endnu så syntes vi det er smartest at starte med den mindste sms pakke.

CpSMS gør det også nemt for os at implementere sms servicen i vores produkt, ved at tilbyde et API som vi kan gøre brug af.

7.3.5 Forretningsmodel Konklusion

Vores forretningsmodel er meget påvirket af resultatet fra vores konkurrentanalyse, derfor måtte vi se på andre indgangsvinkler end den almindelige. Kort prøveperiode og derefter betale for abonnement.

Vi kom frem til muligheden at vise vores produkt for fagforeninger og tilbyde deres medlemmer et godt tilbud, på den måde kunne vi få nye og måske etablerede frisører ind i vores system.

Vi havde overvejet at reklamere via Google for at finde nye kunder, men efter vores analyse af de søgeord der matcher vores segment fandt vi frem til at det vil være alt for dyrt idet vi er en ny spiller på markedet uden en startkapital.

Baseret på den anden indgangsvinkel og vores konkurrentanalyse, syntes vi at vores abonnement skulle stå ud fra mængden. Vi valgte derfor at tilbyde nye kunder hele seks måneders gratis prøve periode, men det vil til gengæld give frisørerne mere lyst til at bruge vores produkt fremover hvis vi kan levere en god service, idet den gode service men også fordi de er vænnet til produktet.

Vi har kigget på forskellige SMS Gateways for vores SMS service, vi kom frem til at den vi først havde udset os for ikke var tilgængelig i Danmark. Dog fandt vi en anden gateway som tilbyder mange af de funktionaliteter vi ønsker, uden at det vil koste os for meget.

7.4 SWOT Analyse

Eftersom der er en stor konkurrence på markedet vil vi gerne lave en SWOT analyse over vores bookingsystem, for at finde ud af hvilke styrker og svagheder vores system har i forhold til de andre på markedet, og for at finde ud af hvilke muligheder og trusler der er for vores system.

Strength:

- Vores system er udviklet fra bunden til at være driftssikker, lav nedetid, nem skalering for øget efterspørgsel.
- Billigere løsning end mange af dem der allerede eksisterer.
- Vores system er billigt at holde kørende, en kunde er nok til at kunne holde 5 servere kørende med nuværende konfiguration.
- Vores system er ikke låst fast til én type booking.

Weakness:

- Vi tilbyder ikke salg, betalingsterminaler og bogføring, som nogle af vores konkurrenter gør.

Opportunities:

- Flere konkurrenter tyder ikke på at have brugt længe på brugervenlighed.

Threats:

- Mange bookingkonkurrenter. Nogle med speciale inden for specifikke brancher og andre som har bookingsystemer som henvender sig til et bredere marked.

7.4.1 SWOT Analyse Konklusion

Baseret på vores SWOT Analyse kan vi konkludere at vi har nogle interne strengths punkter som kan gøre vores produkt eftertragtet, vi kan garantere lave priser og høj oppetid.

Vi har også fundet frem til nogle interne weakness, den kan vi dog vende til at være en opportunity, det giver os en mulighed for at forbedre vores produkt så på længere sigt når vi har implementeret samme features, og måske flere samt forbedret dem, kan vi bruge det som en strength.

En klar ekstern opportunity er at mange af vores konkurrenter bærer præg af dårlig brugervenlighed som også afspejler sig i måden de sælger deres produkt på, på deres hjemmeside gør de brug af gamle metoder for promovering af deres produkt. Vi har erfaring med nye måder at præsentere vores produkt som giver vores kunde et bedre indblik i hvad de får, hvordan det kan hjælpe dem og løse nogle af deres daglige problemer, det kan give os en forkant med konvertering af besøgende til kunder.

Eksterne threats må vi desværre se i øjnene at vi er på vej ind på et område med mange konkurrenter, som både specialiserer sig i specifikke brancher men også konkurrenter som har systemer der kan håndtere flere brancher på én gang.

8 Planlægning af udvikling

Vi vil gerne lave en god arkitektur som er skalérbar og driftssikker selvom en af komponenterne i arkitekturen går ned.

8.1 Single Point of Failure (SPOF)

Betyder at hvis der er én del i systemet som går ned og det også stopper resten af systemet.

Et eksempel på hvordan vores system kunne have været i risiko zonen for SPOF er hvis vi kun havde en server for hele systemet, hvis den gik ned ville vi slet ikke have noget system før serveren kørte igen eller vi havde fået systemet over på en anden server.

Et SPOF punkt kan også være andre dele end kun en server, det kan lige så godt være en router eller et software komponent som en database.

For at finde en passende løsning til vores problemstilling skal vi prøve at undersøge forskellige arkitekturer, forskellige metoder og se på hvordan andre løser samme problem. Hvis vi gør det vil det give os den rette forståelse for problemet og indsigt i hvordan vi kan løse det.

8.2 Website

For at frisøren skal kunne bruge websitet skal frisøren først have en konto som frisøren kan lave igennem en oprettelsesformular på websitet, derefter kan frisøren logge ind.

Menu & Diverse

I toppen af websitet forestiller vi os at vi har nogle "quick actions" eksempelvis hvis vi har en Tilføj booking quick action, så lige meget hvor på siden frisøren er og en kunde ringer ind så kan frisøren hurtigt med ét klik på tilføj booking få grafikken frem, i stedet for at skulle navigere til forsiden og trykke på kalenderen.

I venstre side vil vi placere menuen, med følgende menupunkter.

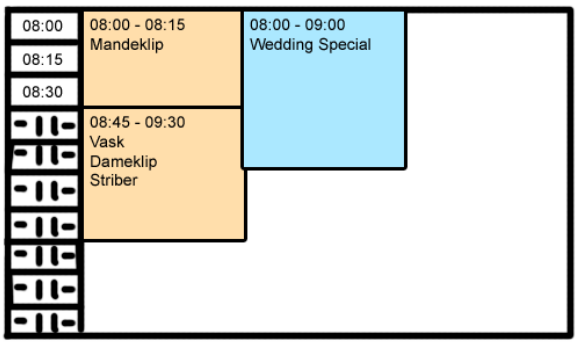
- Booking
  - Tilføj Booking
  - Kalender
- Services
  - Tilføj Service
  - Oversigt
- SMS
- Statistik
- Vagtplan
- Konto
- Hjælp

Forside

Når frisøren er logget ind skal de præsenteres for tre bokse der fortæller, hvor mange bookinger der er i dag, udførte bookinger i dag og hvor mange aflysninger der er i dag.

Frisøren bliver også præsenteret for en kalender som automatisk sættes til dagens dato, som viser et tidsskema og alle dagens bookinger.

Vi har lavet et kort udkast til hvordan vores kalender kan se ud.



Her er kalenderen sat til dagens dato og vi kan se dagens bookinger, i venstre side kan vi se et tidsskema og så i den resterende del af kalenderen kan vi se dagens bookinger.

Hver booking er en grafisk boks hvor boksen er placeret så den matcher starttidspunktet og ligeledes slutter så den matcher sluttidspunktet.

Boksen indeholder start og sluttidspunkt, kort beskrivelse af produktet kunden ønsker og som vi også kan se er der forskellige farver, på denne dato er der to frisører på arbejde og hver bookings baggrundsfarve matcher den valgte frisørs farve. På den måde er det nemt og overskueligt hvilken frisør der har hvilke bookinger.

Når frisøren skal tilføje en ny booking syntes vi det er nemmest hvis frisøren bare kan trykke direkte på kalenderen og så åbner der et nyt vindue hvor frisøren kan tilføje kundens oplysninger samt dato og tidspunkt.

Det er også smart hvis frisøren kan vælge hvilket produkt der skal udføres, og ved hvert produkt er der på forhånd vedlagt en pris og hvor lang tid det tager at udføre, derfor behøver frisøren ikke at skulle indtaste det manuelt, dog skal det stadigt være muligt at ændre hvis der er specielle ønsker eller andet som gør at prisen, eller hvor lang tid det tager, ændrer sig.

Vi syntes også det kunne være smart hvis vi kunne finde ud af hvilken dato og hvilket tidspunkt frisøren trykkede på fordi så kan vi automatisk fylde den information ind, derved gøre det nemmere for frisøren.

For at opdatere en booking syntes vi det vil være smart hvis man i kalenderen bare kan klikke direkte på bookingen og så åbner et nyt vindue med bookingens informationer og så kan man ændre dem som ønsket, når man gemmer opdateringen så skal det afspejle sig i kalenderen med det samme, ikke noget med at skulle opdatere siden for at se den nye information.

Når man har trykket på en booking og det nye vindue er åbent, så skal man i det vindue også have mulighed for at slette bookingen.

Services
Her kan frisøren tilføje og redigere samt slette i alle de services de tilbyder.
Når frisøren tilføjer en ny service kan de give servicen et navn, pris og hvor lang tid det tager at udføre.

Disse services vil blive vist når frisøren eller kunden booker en ny tid og derved kan vi også vise servicen direkte i kalenderen.

SMS Påmindelser
Her kan butiksejeren vælge at aktivere vores SMS service, der vises også en pristabel og en tekstbesked der fortæller butiksejeren, at hvis sms servicen aktiveres vil det medføre ekstra omkostninger.

Butiksejeren skal kunne se en liste over forrige perioder hvor hver vises med antal SMS'er som er sendt og prisen for hver periode, samt teksten "Betalt" eller "Ikke betalt".

Butiksejeren skal også kunne se den aktive periode med hvor mange SMS'er som allerede er sendt samt prisen for de allerede sendte SMS'er.

En periode betyder butiksejerens abonnement som er af 30 dage af gangen.

Butiksejeren har mulighed for at vælge hvilke typer af SMS'er som skal sendes til kunden, her er en liste over mulige SMS typer.

- Ny booking, kunden får en SMS med dato og tidspunkt
- Opdatering af booking, kunden bliver informeret med ny dato og tidspunkt
- Sletning af booking, kunden bliver informeret

- Påmindelse. Ved den her SMS type kan frisøren vælge hvor mange dage i forvejen forud for bookingen at en påmindelse skal sendes til kunden.

Hver type er bare en standard besked, eksempel på ny booking sms kan se således ud:
Du har nu en booking hos Kim & Rasmus Frisørsalon den 29/10/2015 klokken. 10:00. Vi glæder os til at tage imod dig. Mvh. Kim & Rasmus.

Statistik
Frisørerne skal kunne se følgende statistikker.

- En graf der viser hvor mange procent af kunderne som kommer ind fra gaden, hvor mange der ringer og hvor mange som booker online.

Vagtplan
Her skal det være muligt for frisørerne at se deres arbejdsdage og arbejdstider, samt kunne melde sig syg.

For butiksejeren skal det være muligt at kunne tilføje og fjerne frisører samt redigere i deres arbejdsdage og arbejdstider.

Det skal også være muligt for butiksejeren at kunne til- og fravælge om systemet selv skal prøve at passe bookinger ind i de andre frisørers kalender hvis en af frisørerne melder sig syg.

Konto
Her skal frisøren kunne se og ændrer i sine oplysninger og ændrer i sine præferencer.
Baseret på om frisøren er butiksejer eller almindelig frisør vises forskellige muligheder.

- Almindelig frisør
    - Redigere navn
    - Redigere e-mail
    - Redigere kodeord
- Butiksejer
  Udover samme muligheder som almindelig frisør er der også.
    - Rediger butiksnavn
    - Tilføje, Redigere, Fjerne kort for betaling af abonnement

Hjælp
Hjælp skal være en let og overskuelig manual for alle funktionaliteter på websitet.

8.3 Tekniske overvejelser

8.3.1 Responsivt website

For at udvikle et responsivt website måtte vi først kigge på de forskellige muligheder der er tilgængelige.

Ved at læse om responsive hjemmesider fra artikler kom vi frem til at vi har tre muligheder.

- Tilpas et færdigbygget framework

  Vi har fundet et færdigt tema som er bygget med Bootstrap og hedder AdminLTE[8].

  Det smarte ved AdminLTE er at det er et administrationspanel bygget på Bootstrap, dvs. det virker allerede til at starte med på forskellige devices og skærmstørrelser.

  Det inkluderer også rigtig mange plugins og features samt grafiske elementer som gør arbejdet for os endnu nemmere ved at vi ikke selv skal udvikle det fra bunden.

- Udvikle oven på Bootstrap

  Vi kan bruge en ren Bootstrap og udvikle direkte på det framework, det positive er at Bootstrap er bygget til at håndtere forskellige devices og skærmstørrelser dog skal vi selv lave et design og importere mange af de features og plugins vi vil bruge.

- Udvikle alt fra bunden

  Hvis vi skal udvikle alt fra bunden så betyder det at vi skal udarbejde et design først samt hvilke features designet skal have og hvordan det skal se ud.

  Derefter kan vi begynde at opsætte designet med HTML og CSS.

  Ulempen hvis vi valgte at udvikle det helt fra bunden er at vi vil have massiv test for at teste hvorvidt det virker på forskellige devices og skærmstørrelser.

8.3.1.1 Responsivt website delkonklusion

Baseret på de muligheder vi har fundet frem til, vælger vi at bruge AdminLTE der tilbyder mange af de funktionaliteter som vi skal bruge, samt det er færdigbygget og vi kan nøjes med at indsætte de elementer vi skal bruge.

De to andre muligheder vil tage for lang tid at udvikle baseret på vores tidsplan.

[8] https://almsaeedstudio.com/themes/AdminLTE/index.html

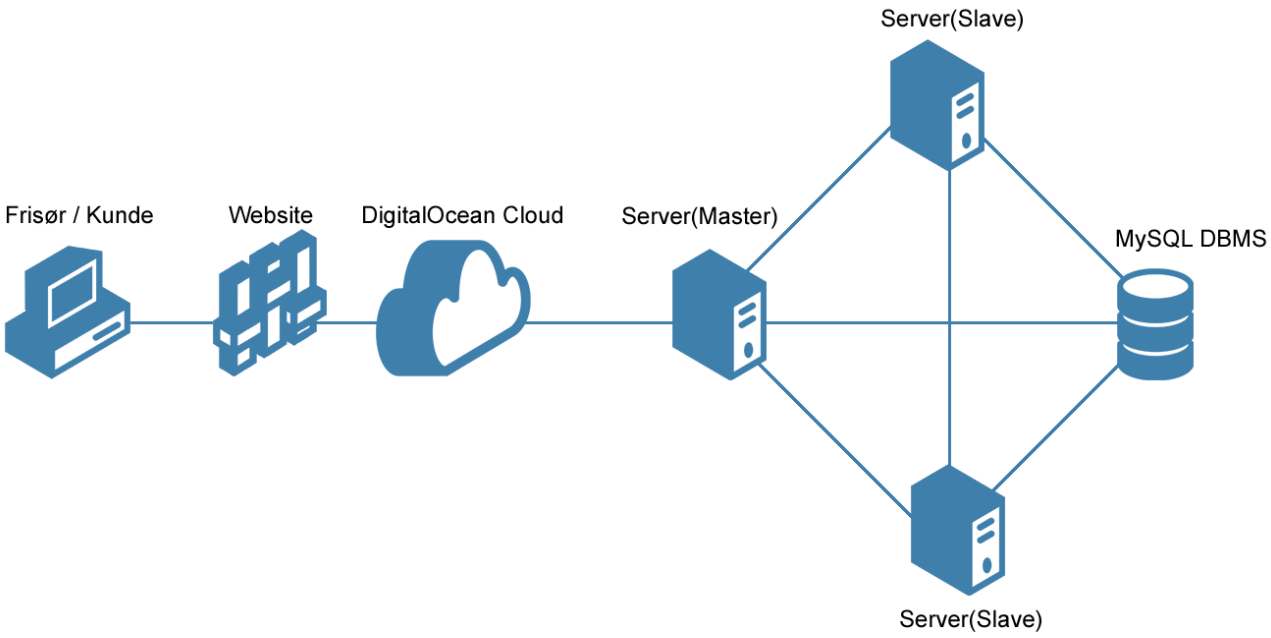
8.3.2 Server Arkitektur

Til at begynde med havde vi sat os for at lave Motoren med C# og compile det med Mono Project Compiler for at få det til at køre på linux operativsystemet.

Vi havde undersøgt nogle forskellige arkitekturer med deres fordele og ulemper, vi havde blandt andet kigget på en mulig web service arkitektur som vi stødte på i tredje semester men vi havde også kigget på hvordan Hadoop[9] gør brug af Master Slave arkitekturen og distribuerer arbejdet ud på flere servere og holder styr på hvilke som er til rådighed m.m.

Vi kom frem til at vi gerne ville lave en kombination af webservices og Master Slave, fordi så kunne vi gøre brug af de bedste elementer fra begge arkitekturer. Fra WebServices kan vi gøre brug af RESTful og fra Master Slave kan vi bruge den del der kan hjælpe os med at skalere servicen ud på flere servere.

Her er et diagram der viser hvordan arkitekturen ville se ud.



Her er en punktopstilling af rækkefølgen fra venstre til højre.

Frisør / Kunde & Website

Først har vi enten frisøren som gerne vil administrere deres salon via vores hjemmeside, eller vi har en kunde som kan booke igennem frisørens egen hjemmeside.

DigitalOcean Cloud

Alle anmodninger sendes igennem DigitalOcean's cloud og til vores Master server.

[9] https://hadoop.apache.org/

Motoren

Masteren modtager anmodningen og sender anmodningen videre til en af slaverne. For at fordele anmodninger havde vi ikke helt besluttet om, vi ved et bestemt interval ville beregne hver servers resterende kapacitet og sende resultatet til Master serveren, eller om vi skulle gøre brug af Round-Robin DNS.

Round-Robin DNS er en teknik til at distribuere arbejdet, ved at opsætte en server med en DNS service og tilføje alle Motorens server IP'er så ved hver request vil DNS'en sende requesten til den næste IP i listen, på den måde kan vi fordele belastningen på alle serverne.

Vi har dog stadigt SPOF risiko fordi DNS'en kan gå ned og så er resten af vores system også nede.

På diagrammet kan vi hurtigt se at vi har et Single Point of Failure ved vores Master server, det kunne vi løse ved at udvikle et stykke kontrol software som skulle ligge på hver server, dets arbejdsopgave ville bestå i at.

1. Sende ping request til de andre servere for at vide om de stadigt lever og er en del af Motoren.
    a. **Problem:** Problemet med ping request er, at de gør brug af UDP protokollen. Vi kan derfor komme ud i et scenarie hvor ping requesten ikke når frem til destinationen og derved skaber uorden i vores kontrol software.
    b. **Løsning:** Der er to hurtige løsninger på ovenstående problem.
        i. Første er at vi fuldkommen dropper ping request og implementere vores egen ping/pong request som opretter en forbindelse mellem de to servere via TCP protokollen, vi kan derfor se at hvis forbindelsen blev oprettet er serveren i live og hvis forbindelsen ikke blev oprettet er serveren utilgængelig. Vi får dog et større overhead med TCP fremfor UDP, eksempelvis med Three-Way Handshake.

            Three-Way Handshake: Når forbindelsen oprettes så skal klienten og serveren igennem et three-way handshake, hvor klienten først sender en pakke hvor SYN feltet er sat, serveren svarer tilbage med SYN og ACK og klienten svare endeligt tilbage med ACK.

        ii. Anden løsning er en kombination af ping request og TCP. Først kan vi sende en ping request og hvis vi ikke får noget svar tilbage så prøver vi at oprette en TCP forbindelse til serveren.

            På den måde gør vi brug af ping request så vidt som muligt, og vi kan garantere at serveren er utilgængelig fordi vi prøver at oprette en TCP forbindelse hvis ping requesten ikke svarede tilbage.

2. Have opdateret information på hvem som var Master og hvem som var Slave.
3. Have opdateret information om de andres serveres resterende kapacitet.
4. Have adgang til DigitalOceans server API for at kunne ændre i server konfiguration.

Dette stykke kontrol software løser to reelle scenarier for os.

Hvis Master serveren går ned vil de resterende serveres kontrol software opfange scenariet idet Master serveren ikke længere svarer på ping requests.

Det er nu de resterende kontrol softwares job at skulle finde den bedste kandidat til at være den nye Master, det ville de gøre ved at kigge på deres information og finde den server med mest overskydende kapacitet, det er nu kritisk at de hver især fortæller hinanden om den server de tror der er den bedste kandidat til at være master, fordi det kan være at der er en difference mellem kontrol softwarens information. Derfor ved at de fortæller hinanden om den bedste kandidat kan vi se på det som en afstemning om den server med flest stemmer får jobbet som Master.

Den nye master skal nu igennem sin tilgang til DigitalOceans API få den offentlige IP til at pege på sig selv, dvs. vi har én offentlig IP som altid peger på Master serveren dermed kan vores website, frisørernes website og eventuelle andre fremtidige indgangsvinkler til vores produkt nøjes med én IP fordi den altid peger på vores Master server.

Vi havde nu ideen til arkitekturen på plads så vi begyndte at undersøge om der var nogle elementer fra .NET frameworket som Mono Project Compiler ikke kunne kompilerer, vi fandt frem til at WCF(Windows Communication Foundation) som bruges til at lave webservices kun delvist var implementeret.

Det betød for os at vi ikke kunne bygge den arkitektur med C#, vi kom derfor frem til den konklusion at bruge C# på Motoren ikke var et særligt godt valg idet vi brugte et operativsystem som er bygget på linux.

Vi har derfor brugt lidt tid på at undersøge andre sprog som mulige kandidater til vores Motor, kravene til sproget er at det skal både kunne køre funktionsdygtigt og konfliktfrit på en ubuntu server og at det skal kunne bruges i et skalérbart miljø.

NodeJS

"As an asynchronous event driven framework, Node.js is designed to build scalable network applications. In the following "hello world" example, many connections can be handled concurrently. Upon each connection the callback is fired, but if there is no work to be done Node is sleeping." [10]

Vi fandt frem til NodeJS som opfylder begge vores krav, vi har også studeret hvordan opbygningen af en NodeJS applikation ser ud og afprøvet det på en af vores servere.

Her er en simpel 'proof of concept' applikation vi byggede og afprøvede på vores server.

```
var http = require('http');
http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('Hello visitor\nWelcome to NodeJs Enviroment .');
}).listen(8080, 'IP');
console.log('Server running at http://127.0.0.1:8080/');
```

[10] https://nodejs.org/en/about/

Hvorfor er det et proof of concept?
Kodestykket ovenover har gjort det klart for os at vi kan gøre som vi ønsker.

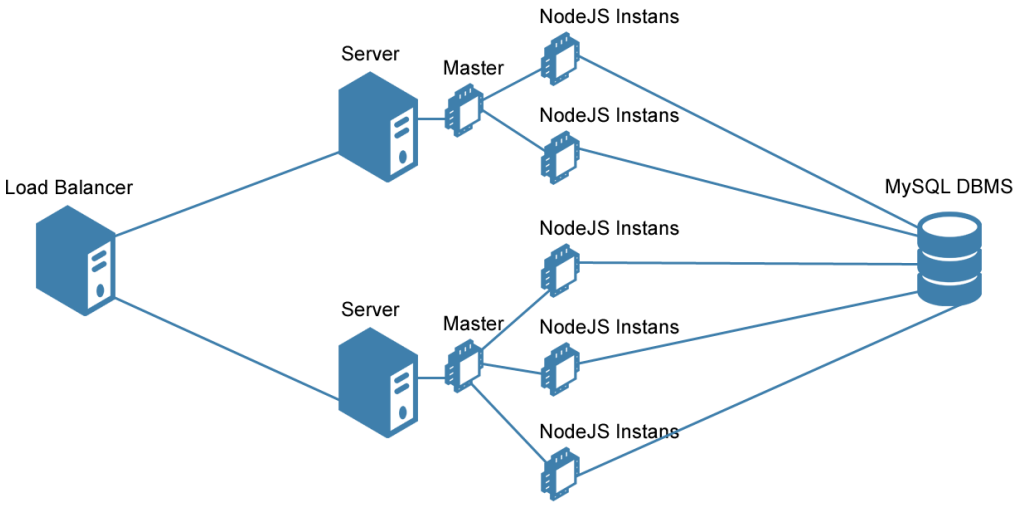
Vi kan lave en webservice og bruge HTTP protokollen der som regel bruger TCP protokollen så vi behøver ikke tage højde for problemer såsom packet loss.

Vi kan bestemme HTTP pakkens content-type, som i vores tilfælde skal være application/json

Vi ved at applikationen ikke blokerer for andre indkomne anmodninger.

Vi har også kigget lidt på mulighederne for at skalere en NodeJS applikation og her ser det også godt ud, vi kan lægge den ud på vores servere og putte en load balancer foran.

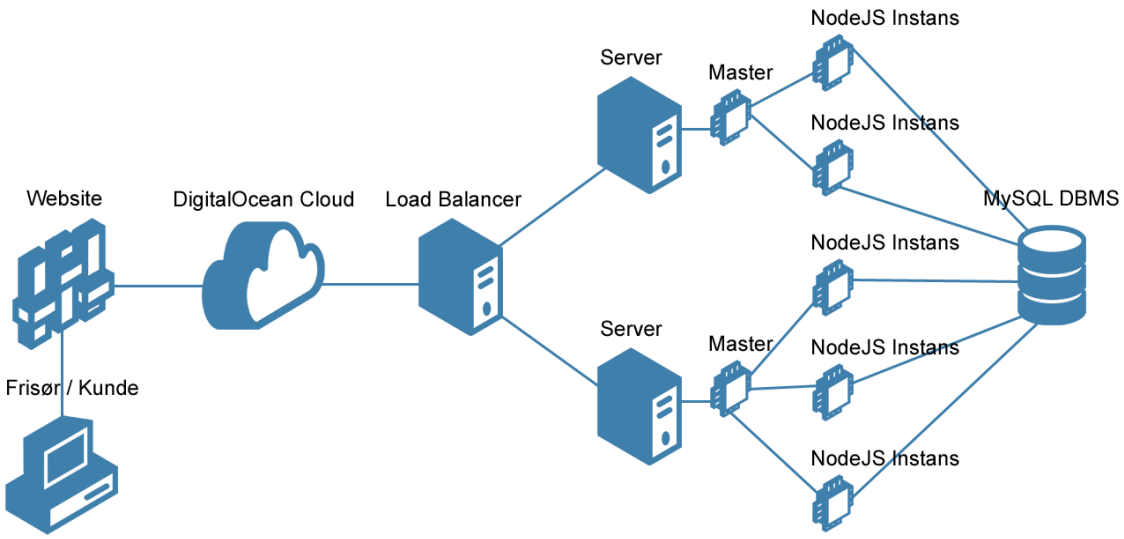
Èt problem med NodeJS er dog det er enkelttrådet, men det har vi fundet en løsning på ved at bruge NodeJS Cluster Module, det gør os i stand til at kunne køre en instans af applikationen per. CPU kerne. Vi har så per server en master instans som modtager den indkomne besked og via Round-Robin algoritmen delegerer beskeden videre til en af applikation instanserne.



Her kan vi se et diagram som viser hvordan vi kan skalere en NodeJS applikation, både med flere servere men også gøre brug af servernes CPU og vi kan også tage højde for servere hvor nogle har flere eller færre CPU kerner end andre servere.

Endelig serverarkitektur

Her et diagram som viser hvordan vores arkitektur kommer til at se ud.



Hvis vi læser diagrammet fra venstre mod højre.

Frisør / Kunde – Website

Først har vi enten frisøren som gerne vil administrere deres salon via vores hjemmeside, eller vi har en kunde som kan booke igennem frisørens egen hjemmeside.

DigitalOcean Cloud

Alle anmodninger sendes igennem DigitalOcean's cloud og til vores Load Balancer.

Load Balancer

Til at fordele arbejdet ud på vores servere, gør vi brug af teknikken Round-Robin DNS der indeholder IP adresserne på serverne, og sender arbejdet videre til serverne skiftevis.

Motoren

Når en anmodning når til en af vores servere vil anmodningen først gå ind i master NodeJS instansen på den pågældende server, derefter vil NodeJS Cluster Module bruge Round-Robin algoritmen for at dele anmodningerne imellem serverens CPU kerner.

Anmodningen vil blive eksekveret af vores applikation og nødvendige kald til databasen vil blive udført.

8.4 JSON

For at websitet kan kommunikere med vores Motor og for at serverne kan kommunikere internt med hinanden skal vi implementere JSON.

Vi skal implementere en serializer og de-serializer i både NodeJS og PHP, den i NodeJS skal ligge på vores server arkitektur og den i PHP skal ligge på website serveren.

Vi skal også sikre os at vi laver en standard data model for hele projektet for at undgå uregelmæssigheder i dataene, en standard data model vil også gøre det lettere for os at have ens JSON serializer og de-serializer på begge platforme.

8.5 Udviklingsproces

I vores product backlog har vi valgt at gøre brug af tre kategorier. To Do, In Progress og Done.

- To Do
  - Indeholder alle vores user stories
- In Progress
  - Indeholder hvilke user stories der arbejdes på nu.
- Done
  - De user stories der er færdige.

I vores sprint backlog har vi valgt at gøre brug af tre kategorier. To Do, In Progress og Done.

- To Do
  - Indeholder de user stories vi har estimeret vi kan nå i sprintet, og de gældende tasks.
- In Progress
  - Indeholder hvilke tasks der arbejdes på nu.
- Done
  - De tasks der er færdige.

Vi har valgt at definere done ved at man kan sætte kryds i følgende punkter:

- Koden er færdig så den opfylder den tasks arbejdsopgave og acceptkriterier
- Koden er kørt igennem Unit Test uden fejl.
- Koden følger vores kodestandard for navngivning m.m og koden er veldokumenteret med kommentare.

Og hvis følgende Non Functional Requirements er opfyldt:

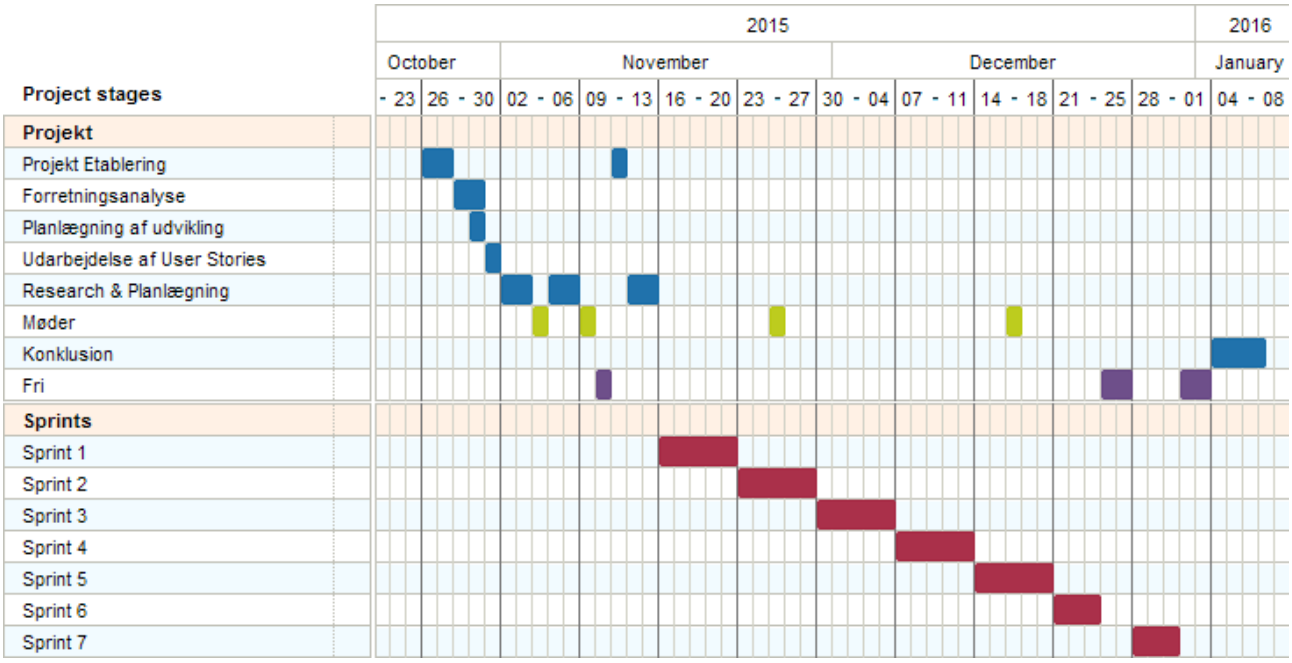
- Intet SPOF (Single Point of Failure) I Motoren
- Motoren skal kunne 'scale-out[11]' og 'scale-up[12]' imens systemet kører.

[11] Skalere ud til flere servere efter behov
[12] Skalere serverens hardware efter behov

8.6 Gantt Chart

Vores første Gantt Chart var et hurtigt udkast, så vi havde en overordnet ide om hvor meget tid vi har til de forskellige ting. Vi kunne dog hurtigt se at vi ikke kunne holde den tidsplan, og vi havde heller ikke taget højde for møder, undervisning og fridage/ferie.

Vi har derfor lavet en opdateret version af diagrammet der ser således ud:



9 User Stories

User story 1 - high risk

Story

Som frisør,
skal jeg kunne tilføje bookinger i min kalender selv,
så jeg kan tage imod folk der ringer eller kommer ind i min salon fra gaden.

Acceptkriterier

- Det skal være muligt at vælge om bookingen er sket over telefon eller ansigt til ansigt.

Story points: 40

User story 2 - high risk

Story

Som frisør,
skal jeg kunne ændre eller slette bookinger,
så jeg kan opdatere min kalender i tilfælde af aflysning eller lignende.

Acceptkriterier

- Det skal være muligt at kunne give mine kunder besked hvis jeg ændrer deres booking.

Story points: 13

User story 3 - high risk **Story** Som kunde, skal jeg kunne booke tid online, så jeg kan spare tid. **Acceptkriterier** • Det skal kun være muligt at se tider der er ledige. • Det skal være muligt at vælge hvilken medarbejder man vil klippes af. **Story points: 20**	**User story 4 - high risk** **Story** Som frisør, vil jeg gerne kunne se de bookinger der er sket online i min kalender, så jeg kan spare tid ved ikke selv at skulle føre dem ind. **Acceptkriterier** • Det skal være let at identificere kunden der har booket online når han/hun står i butikken. **Story points: 3**
User story 5 - high risk **Story** Som frisør, skal jeg kunne tilføje/ændre/slette hvilke services/produkter kunden kan booke, så jeg altid kan holde kunderne opdaterede på hvad jeg tilbyder. **Story points: 8**	**User story 6 - high risk** **Story** Som frisør, skal jeg have en vagtplan over alle medarbejdere, så jeg kan planlægge deres arbejdsdage og arbejdstider. **Acceptkriterier** • Det skal være nemt og tydeligt at identificere medarbejdere på vagtplanen- **Story points: 20**
User story 7 - med risk **Story** Som frisør, skal jeg kunne gemme oplysninger om kunderne, så jeg kan give en bedre service næste gang de kommer. **Acceptkriterier** • Det skal være muligt at gemme basale	**User story 8 - med risk** **Story** Som kunde, skal jeg modtage beskeder når en booking, oprettes, ændres eller aflyses så jeg bliver holdt opdateret omkring min booking. **Acceptkriterier**

oplysninger såsom navn, alder, nummer osv. • Oplysninger om kunden skal gemmes automatisk ved online booking. • Det skal være muligt at tilføje en beskrivelse/note om kunden. **Story points: 8**	• Beskeder skal sendes til kunderne via SMS. **Story points: 5**
User story 9 - low risk **Story** Som kunde, skal jeg kunne modtage påmindelser om min bestilte tid, så jeg ikke glemmer min booking. **Acceptkriterier** • Påmindelser skal modtages via SMS. **Story points: 5**	**User story 10 - low risk** **Story** Som frisør. skal jeg kunne se statistik over bookinger, så jeg kan følge med i hvordan forretningen går. **Acceptkriterier** • Der skal holdes statistik over antal bookinger der er sket i alt. • Der skal holdes statistik over antal online bookinger. • Der skal holdes statistik over antal bookinger over telefon. • Der skal holdes statistik over antal bookinger hvor folk er kommet ind fra gaden. **Story points: 8**

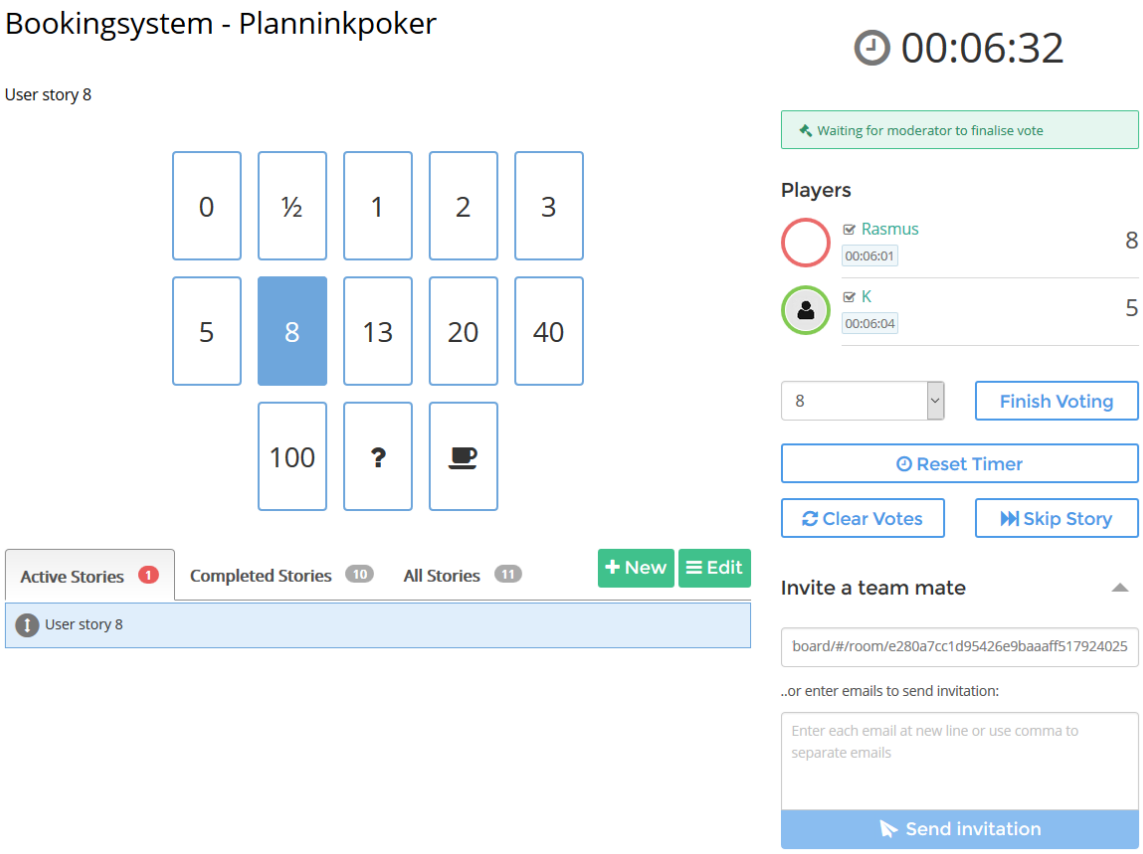
10 Planning Poker

Vi har som team taget udgangspunkt i en user story der lyder:

Som administrator,
skal jeg kunne oprette en bruger i vores system,
Så jeg kan tilføje nye brugere.

Vi har estimeret ovenstående user story til at have et omfang af 5 story points.

Vi har gjort brug af http://www.planitpoker.com/ som er et online værktøj til at lave planning poker. Man kan oprette user stories, flytte rundt på dem og når man så er klar går man i gang med at estimere. Man logger ind hver for sig og kan så vælge et kort, og først når alle har valgt bliver det synligt hvad andre har valgt.



Der var nogle user stories vi ikke estimerede ens, hvilket gav grundlag for at diskutere vores individuelle forståelse af den enkelte user story. Fx. ved estimering af user story 9 havde vi forskellige forståelser af omfanget af opgaven, og endte, efter at have snakket om den, ud med en fælles estimering der var højere end vi først havde sat.

Efter alle user stories var estimeret har vi indsat tidsestimatet til den aktuelle user story oppe i skemaet der indeholder alle vores user stories.

11 Sprint 1

11.1 Mandag

11.1.1 Daily Scrum

I dag skal vi først og fremmest holde et sprint planning meeting, hvor vi finder frem til de user stories vi skal arbejde med. Efter mødet skal vi have opdelt de gældende user stories til tasks, og derefter i gang med at arbejde med de tasks vi har fundet frem til. Der er ikke noget der forhindrer os i at komme videre.

11.1.2 Sprint Planning

Eftersom vi selv er product owner og scrum team så har vi ikke kunne have haft et rigtigt møde som vi ville have haft hvis vi udviklede et produkt for en anden.

Vi har sammen snakket om hvilke user stories der giver mest værdi for projektet i dette tidlige stadie.

I rollen som scrum team blev vi hurtige enige om at vi skulle begive os i kast med userstory 1, at kunne tilføje en booking. Vi har i rollen som product owner sat den øverst på product backlog med high risk.

11.1.3 User stories til tasks

Vi fandt frem til nedenstående tasks ved at se på user storyen fra software verdenens perspektiv og spørge os selv hvilke komponenter som skal implementeres for at user storyen opfylder de krav vi har fastsat i vores definition af Done.

User story 1: Som frisør, skal jeg kunne tilføje bookinger i min kalender selv, så jeg kan tage imod folk der ringer eller kommer ind i min salon fra gaden.

Tasks:

- Installer og konfigurer servere
- Implementer skalérbar arkitektur
- Skriv tests til arkitektur
- Implementer kalender frontend
- Implementer tilføj booking backend
- Implementer tilføj booking frontend
- Skriv tests til tilføj booking backend

11.1.4 Udvikling

11.1.4.1 Sequence Diagram

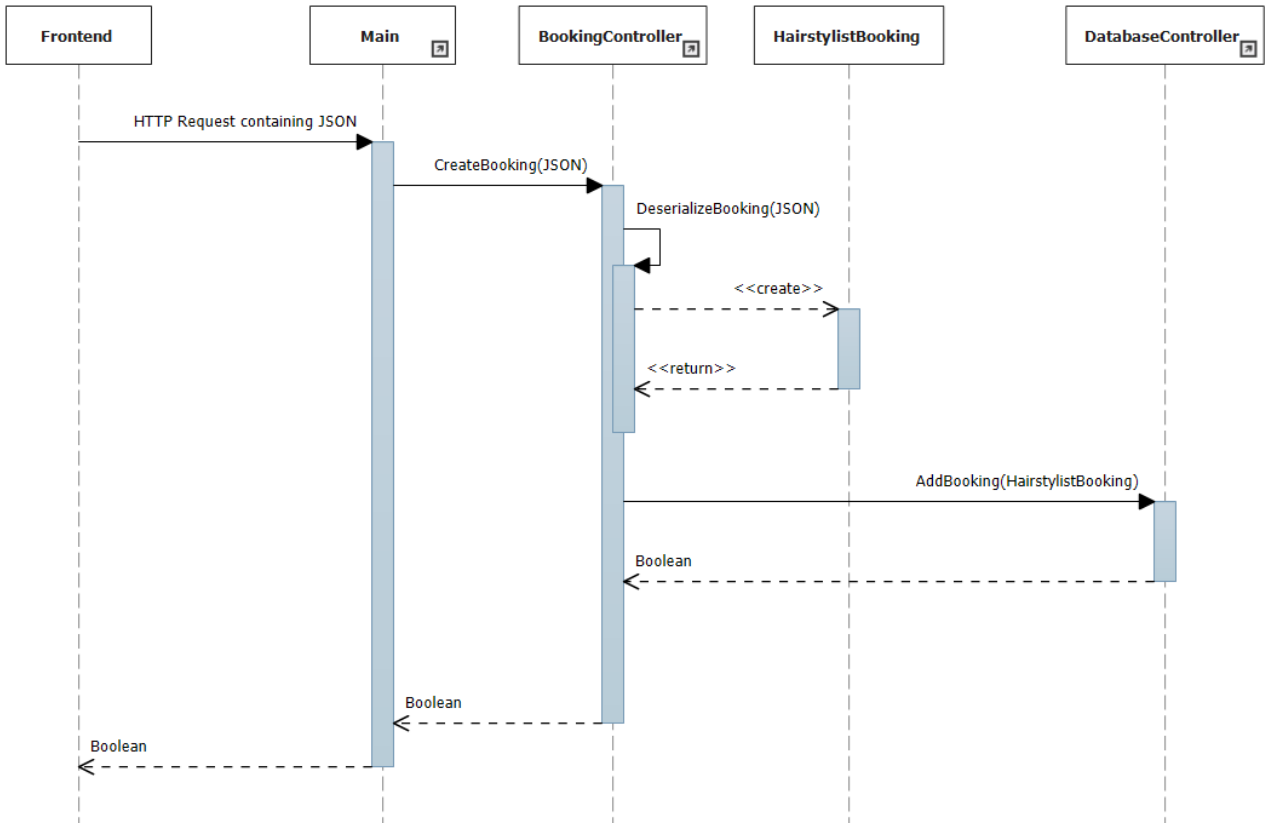
For at få en fælles forståelse og skabe et overblik over hvordan vi skal kode vores backend, har vi valgt at udvikle et sequence diagram. Vi vil også diskutere hvordan vi skal navngive klasser, metoder osv. imens vi udarbejder SD'et.

Vi er nået frem til to essentielle "regler" hvad navngivning angår:

- Al navngivning skal være på engelsk.
- Al navngivning skal følge CamelCase.

Udover de to regler snakkede vi om mere generelle ting fx at en frisør hedder en Hairstylist, og når en booking oprettes bruger vi Create internt i systemet, og Add så snart bookingen skal tilføjes til databasen.

I designet af diagrammet har vi sørget for at fokusere på hvordan vores backend overordnet skal udføre user storyen. Vi har kigget på hvilke klasser vi skal have, hvilke metoder der skal kaldes og hvad data der skal sendes med frem og tilbage.



11.1.4.2 Entity Relationship Diagram

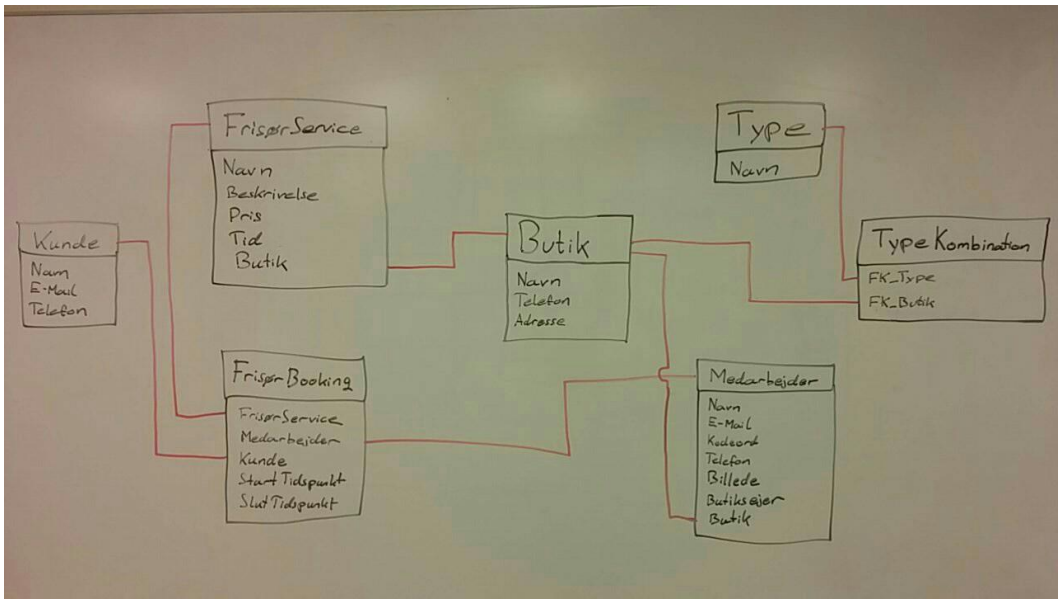
Til at designe vores database vil vi udvikle et Entity Relationship Diagram, for at give et bedre overblik både i designfasen og når vi skal sætte databasen op.

Som en start brugte vi en tavle så vi hurtigt kunne få vores ideer tegnet op, og vi nemt og hurtigt kunne ændre i designet. I første omgang gik vi heller ikke så meget op i normalformerne, og forskellige typer nøgler så som primær, kandidat, fremmed osv.

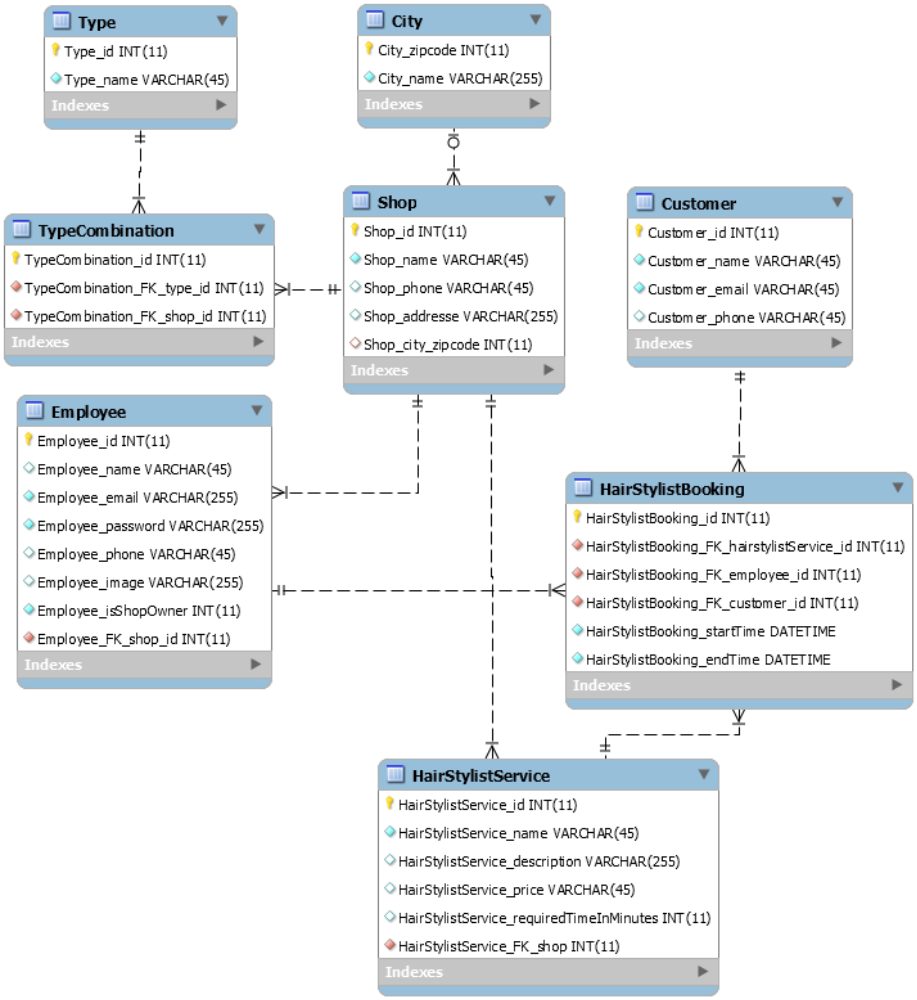
Vi havde nogle udfordringer omkring designet af selve bookingen, da en booking helst skal være generisk. Vi kom frem til en løsning hvor vi for hver type af booking, har en tilhørende x-Booking tabel og en x-Service tabel.

Vi overvejede forskellige løsninger før vi nåede frem til ovenstående, bl.a. kiggede vi på en løsning hvor booking og service skulle være overordnede tabeller for alle slags bookinger. Vi syntes dog ikke det ville være en særlig god løsning, da vi ville ende ud med rigtig mange felter i tabellerne der ville være null.

Her ses et billede af det overordnede design vi besluttede os for at gå videre med.



Her kan vi se vores endelige ER diagram over vores database.



Vi kan også se at vi har tilføjet en ekstra tabel ved navn "City" som indeholder to felter, City_name og City_zipcode hvor City_zipcode er primærnøglen.

Tabellen skal indeholde en liste over byerne i Danmark samt deres respektive postnumre.

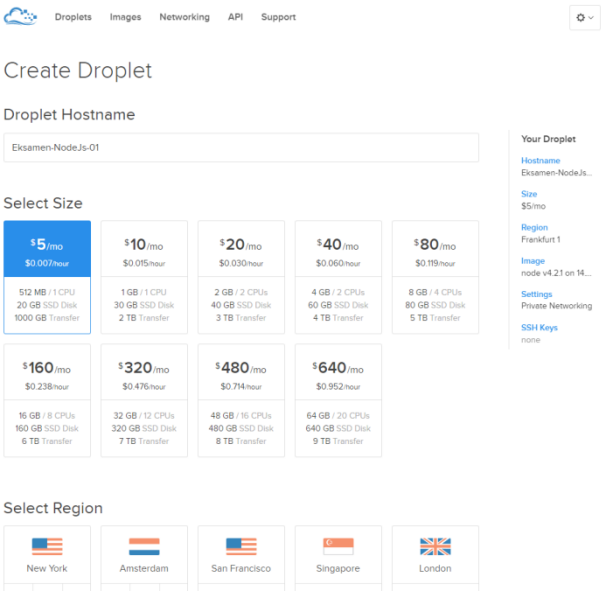
I tabellen Shop kan vi se at vi har tilføjet feltet Shop_city_zipcode af typen INT som referere til primærnøglen i tabellen City.

Vi har valgt at lægge bynavnet og postnummeret i sin egen tabel for at overholde tredje normalform.

11.1.4.3 Task: Installer og konfigurer servere

Igennem cloud servicen DigitalOcean har vi opsat to servere indtil videre.

- MySQL Server
- NodeJS Server



Serverne er placeret i DigitalOceans datacenter i Frankfurt med følgende konfiguration.

- CPU: 1 Core af 1.8 - 3.0 Ghz
- RAM: 512 MB
- Disk: 20 GB SSD
- OS: Ubuntu Server

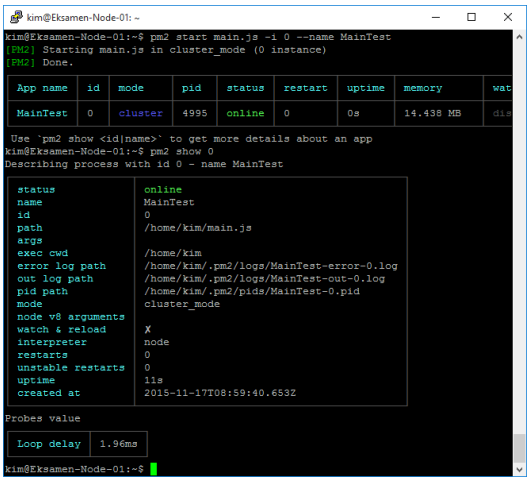
Det første vi gjorde var at logge ind på serverne med root brugeren via SSH og tilføje de nyeste opdateringer, derefter tilføjede vi to nye brugere, en til hver af os og med administratorrettigheder.

Vi konfigurerede SSH servicen til kun at acceptere logins via SSH nøgler med kodeord, derefter tilføjede vi SSH nøgler til vores brugere.

Vi installerede MySQL og fik opsat en database, vi fik også tilføjet to konti så vi begge kan tilgå databasen.

Vi installerede NodeJS og konfigurerede et produktionsmiljø som holder øje med vores kommende NodeJS applikation.

Produktionsmiljø: PM2 - Advanced, production process manager for Node.js



Vi valgte at installere PM2 fordi det gør os i stand til meget nemmere at kunne håndtere og holde øje med vores NodeJS applikation.

- PM2 tilbyder mange funktionaliteter, men vi har valgt at bruge følgende.
- Watch & Restart
  - Genstarter automatisk vores applikation hvis der er nye ændringer i applikationens filer
- Monitoring CPU & Memory
  - Vi kan i realtime se hvor mange ressourcer vores applikation bruger og se hvordan det ændrer sig ved forskellige belastninger.
- Cluster Mode
  - Gør det muligt for os at kunne skalere arbejdsbyrden let mellem tilgængelige CPU kerner. Denne feature er speciel vigtig fordi det gør os i stand til kunne skalere på servere med forskellige typer af CPU'er.
- Startup Script
  - Hvis serverne går ned eller genstarter kan vi få PM2 til automatisk at starte vores applikation når serveren starter op.

11.2 Tirsdag

11.2.1 Daily Scrum

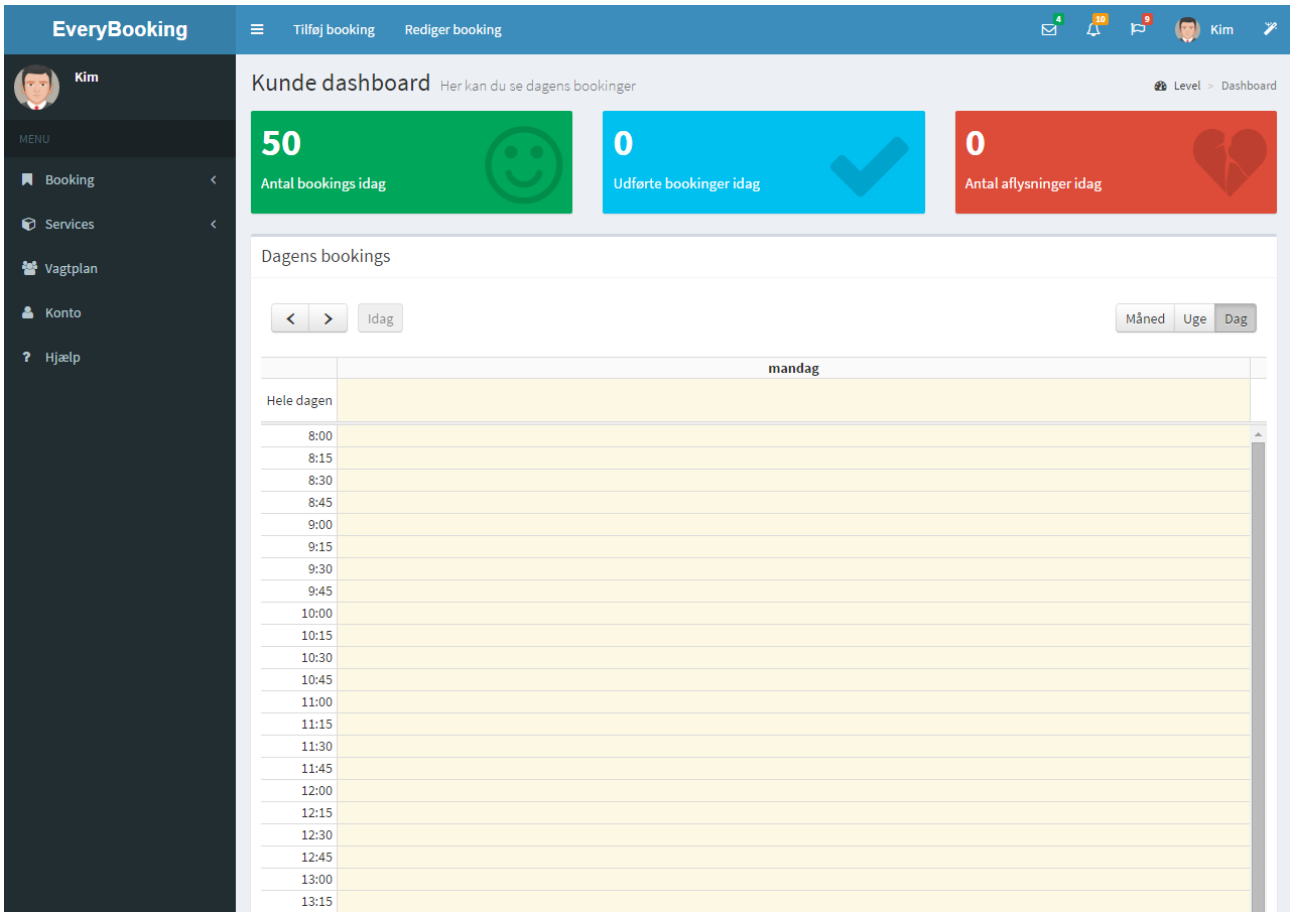
Frem til i dag har vi arbejdet på at opsætte og konfigurere vores servere, vi har udarbejdet et ER diagram over vores database og vi har designet backend og lavet en SD dertil.

I dag skal vi i gang med at udvikle på hjemmesiden og den skalerbare server arkitektur. Det kan blive en forhindring at vi har blandet erfaring med NodeJS.

11.2.2 Udvikling

11.2.2.1 Frontend

Vi har fået opsat AdminLTE på localhost og sammensat forsiden som møder frisører når de logger ind.



I venstre side har vi menuen med menupunkter.

I toppen har vi vores quickmenu, dvs. lige meget hvor i systemet som frisøren er, så kan frisøren altid med ét klik tilgå en vigtig feature, her er det tænkt som at hvis frisøren modtager et telefonopkald så skal frisøren kunne oprette en booking med det samme.

I toppen af midt på siden kan vi se tre store bokse, som hver definere en kort statistik over dagen med: Antal bookinger i dag, Udførte bookinger i dag samt antal Aflysninger i dag.

Nedenunder kan vi se kalenderen som altid er sat til at vise "i dag" men det er muligt at navigere i den via pilene til venstre og knapperne til højre.

11.2.2.2 Backend

For at finde ud af hvordan vi kan modtage JSON via HTTP requests, har vi opsat en simpel web service i NodeJS. Vores web service lytter på indkommende HTTP POST requests på IP:3000 og forventer at kroppen i HTTP requesten indeholder JSON data.

For at teste om motoren fungerede, prøvede vi først at sende et HTTP post request fra websitet, bestående af JSON data og med content type sat til application/json, hvilket lykkedes.

Efterfølgende har vi udviklet et NodeJS script der kan sende HTTP requests med JSON som indhold, som vi kan benytte til at teste i fremtiden.

main.js

```
var express = require('express');
var app = express();

var bodyParser = require('body-parser');
app.use(bodyParser.json());

app.post('/', function (req, res) {
 console.log(req.body);
 res.send("ok");
});

var server = app.listen(3000, function () {
 var host = server.address().address;
 var port = server.address().port;

 console.log('Example app listening at http://%s:%s', host, port);
});
```

I main.js, som er vores indgangspunkt til motoren, kan man se at vi gør brug af modulet 'express' til at lytte på indkomne requests og håndtere dem.

11.3 Onsdag

11.3.1 Daily Scrum

Frem til i dag har vi arbejdet på at udvikle både frontend og backend, vi har lavet nogle tilretninger i databasen og vi har fundet frem til en fælles JSON struktur.

I dag skal vi i gang med at udvikle på hjemmesiden og den skalerbare server arkitektur. Det kan blive en forhindring at vi har blandet erfaring med NodeJS.

11.3.2 Udvikling

11.3.2.1 JSON struktur opdateret

I går blev vi enige om hvordan JSON strukturen skulle være.

Efter lidt mere research om NodeJS Express modulet er vi kommet frem til at vi vil fjerne felterne Request og RequestType fra JSON strukturen og i stedet gøre brug af Express modulets måde at håndtere indkommende HTTP requests.

Med Express startede vi med at tilegne en anonym funktion på eventet post.

```
app.post('/', function (req, res) {});
```

Vores ide var at hver eneste indkommende HTTP post skulle igennem ovenstående anonyme function, og derinde ville vi se på JSON feltet Request som fortalte os om JSON dataen var relateret til booking, authentication eller andet. Derefter ville vi se på feltet RequestType som fortalte nærmere hvad der skulle udføres.

Dvs. hvis feltet Request er Booking så kunne RequestType være AddBooking, RemoveBooking, AlterBooking m.m.

Vi synes det er dumt at JSON dataen fortæller vores WebService hvad WebServicen skal gøre med dataene, JSON dataene skal være ren og kun indeholde den nødvendige data.

Et andet problem med ovenstående eksempel ville være at vi har én anonym funktion som modtager alle indkommende HTTP request og derfor skulle læse JSON Request feltet og derefter gå igennem en if eller switch for at delegere arbejdet videre til den korrekte controller.

For at gøre det smartere kan vi bruge Express modulet og lytte specifikt på HTTP post kaldene.

```
app.post('/Booking/Add', function (req, res) {});
app.post('/Booking/Remove', function (req, res) {});
app.post('/Booking/Alter, function (req, res) {});
```

Det smarte ved denne løsning er at hvis vi eksempelvis skal oprette en ny booking så sender vi det absolut nødvendige data i JSON format i HTTP kroppen til IP:port/Booking/Add.

Når dataene kommer ind til vores webservice behøver vi ikke kigge i JSON dataene for at se hvad det er, i stedet forventer vi at JSON dataene kun har med en ny booking at gøre, dog skal vi selvfølgelig validere dataene osv. men det kommer senere, nuværende er det kun kommunikationen mellem hjemmesiden og webservicen.

11.3.3 Retrospective

Bliv ved med at

- Mødes hver dag, med faste mødetider.
- Have en god dialog både om udvikling/fremgangsmåder samt værktøjer både indenfor systemudvikling og softwareudvikling.

Stop med at

- Afholde enkelt ugers sprint.

Start med at

- Holde styr på backloggen, tasks, burndown m.m. efter daily scrum.
- Holde to ugers sprint.

Mere af

- Fokus på den konkrete user story.

Mindre af

- Vi tænker for meget over fremtidige user stories, og hvordan vi skal udvikle så fremtidige features passer ind.

11.4 Konklusion – Sprint 1

Eftersom dette har været vores første sprint har vi brugt meget af tiden på at planlægge forskellige dele af softwaren og databasen.

Vi kom desværre lidt bagud fordi det tog længere tid end forventet, det resulterede i at noget af den tid vi havde tilsidesat til decideret udvikling blev tabt.

Vi nåede derfor desværre heller ikke at færdiggøre vores valgte user story, vi nåede dog at udføre nogle af de tasks vi fandt frem til, men de blev aldrig helt "done" idet vi heller ikke fik nået at skrive unit tests til dem.

Selvom vi kom bagud med selve udviklingen, kommer planlægningen til at gavne os fremadrettet. Vi valgte f.eks. at designe hele databasen, da vi ved at vi ellers kommer til at bruge rigtig meget tid på at ændre store dele af den, når der kommer ændringer, også selvom det strider imod det agile, hvor man kun laver tilstrækkeligt nok for at løse en user story.

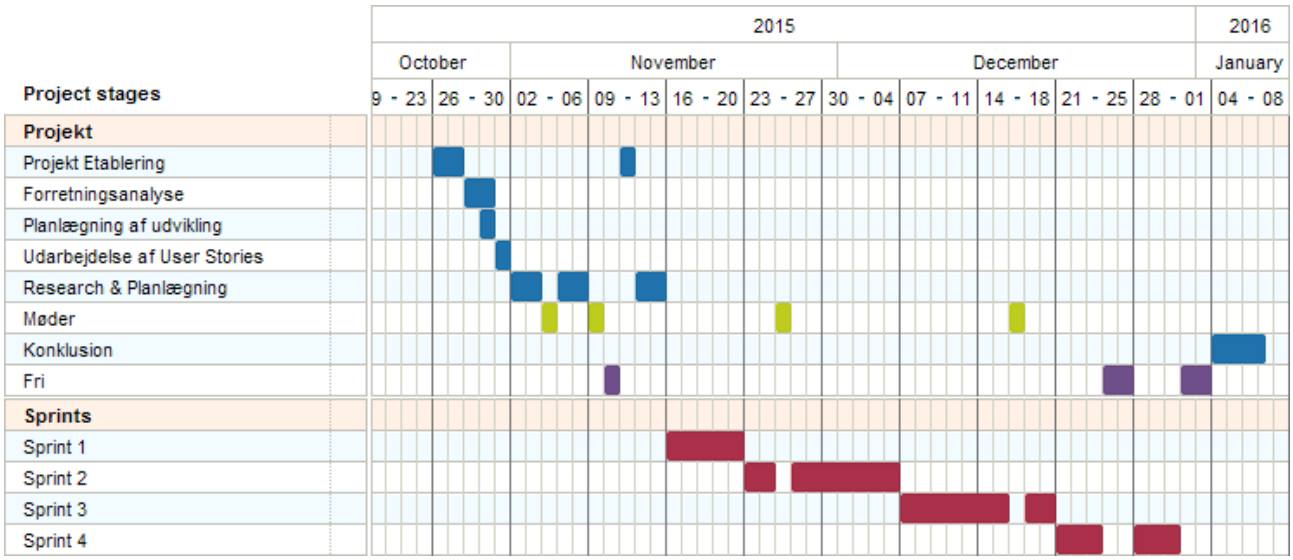
Vi har holdt os til kun at designe selve softwaren tilstrækkeligt nok til at løse user storyen, men eftersom at det er det første sprint kan vi ikke undgå at udvikle nogle features der vil blive brugt senere også. Bl.a. har vi i dette sprint brugt noget tid på at opsætte serverne, hvilket er noget der vil blive brugt i stort set alle fremtidige sprints.

Efter første sprint har vi også måtte indse, at sprints på én uge er for korte. Vi føler ikke at vi når at komme ordentligt i gang med sprintet, før vi skal til at afslutte det igen.

Da vi lavede retrospective over sprintet, snakkede vi om det og fik afklaret at vi fremover vil afholde sprints af to uger i stedet for en uge.

11.5 Gantt Chart

Vi har opdateret vores Gantt Chart for at tage højde for, at vi nu arbejder med to ugers sprint i stedet for en uges sprint.



12 Sprint 2

12.1 Mandag

12.1.1 Daily Scrum

I dag skal vi først og fremmest holde et sprint planning meeting, hvor vi finder frem til de user stories vi skal arbejde med. Efter mødet skal vi have opdelt de gældende user stories til tasks, og derefter i gang med at arbejde med de tasks vi har fundet frem til. NodeJS er stadig nyt hvilket kan være en forhindring.

12.1.2 Sprint Planning

Fortsættelse af User Story 1 og tilføjelse af User Story 2

I sidste sprint nåede vi desværre ikke at fuldføre alle de tasks vi havde fundet frem til på grund af megen planlægning og alt for kort et sprint.

Vi har derfor i samarbejde med product owner fundet frem til at vi tager det resterende arbejde fra user story 1 over i sprint 2.

Ved at se på hvor lang tid de resterende tasks fra User Story 1 tager har vi konkluderet at vi også kan nå at lave User Story 2.

Resterende tasks fra User Story 1:

- Implementer skalérbar arkitektur
- Skriv tests til arkitektur
- Implementer tilføj booking backend
- Implementer tilføj booking frontend
- Skriv tests til tilføj booking backend

12.1.3 User stories til tasks

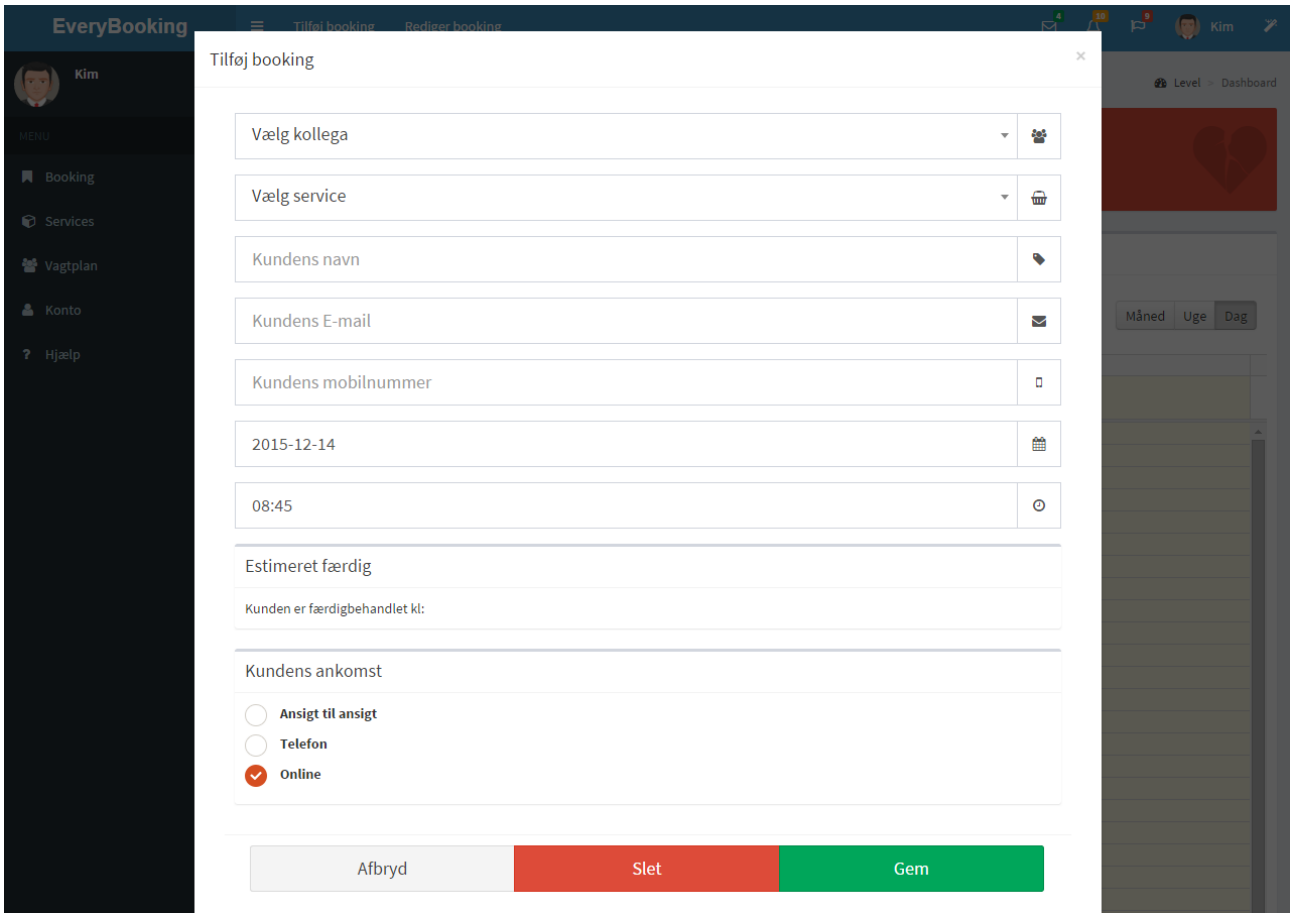
User story 2: Som frisør, skal jeg kunne ændre eller slette bookinger, så jeg kan opdatere min kalender i tilfælde af aflysning eller lignende.

Tasks:

- Implementer slette booking frontend
- Implementer ændre booking frontend
- Implementer slette booking backend
- Skriv tests til slette booking backend
- Implementer ændre booking backend
- Skriv tests til ændre booking backend

12.1.4 Udvikling

12.1.4.1 Frontend



Vi har fået implementeret således at når man trykker på et sted i kalenderen så åbner der en boks som kan ses i billedet ovenover.

Her skal man vælge: Kollega, service, kundens navn, kundens email, kundens telefonnummer samt hvordan kunden ankom, om det var i butikken eller telefon.

Ovenover kan man se at der er sat hak i “online”, men det er kun for at teste, det giver ikke så meget mening når det er frisøren som opretter bookingen her.

Baseret på hvilken service som er valgt kan vi vise hvilket klokkeslæt hvor kunden er færdigbehandlet.

Dato og tidspunkt bliver automatisk sat ind baseret på hvor man klikker i kalenderen.

I bunden er der tre knapper:

- Afbryd - Lukker boksen
- Slet - Vises kun hvis man har trykket på en eksisterende booking i kalenderen og bruges til at slette bookingen.
- Gem - Gemmer bookingen

12.1.4.2 JSON Validering

For at validere det JSON objekt der bliver sendt når der skal tilføjes en booking, gør vi brug af et modul til NodeJS der hedder 'jsonschema[13]'.

jsonschema virker ved at holde et prædefineret JSON skema op imod det JSON objekt der skal valideres. Det validere at datatyperne stemmer overens med det prædefinerede skema, og hvis der er sat nogle krav op, fx at et felt af typen integer skal minimum være 10.

Vores skema ser således ud:

```
var schema = {
    "type": "object",
    "properties": {
        "serviceId": {"type": "integer", "minimum": 0},
        "emplyeeId": {"type": "integer", "minimum": 0},
        "customerId": {"type": "integer", "minimum": 0},
        "startTime": {"type": "string", "pattern": "[0-9]{4,4}-[0-9]{2,2}-
[0-9]{2,2}   [0-9]{2,2}:[0-9]{2,2}:[0-9]{2,2}"},
        "endTime": {"type": "string", "pattern": "[0-9]{4,4}-[0-9]{2,2}-[0-
9]{2,2} [0-9]{2,2}:[0-9]{2,2}:[0-9]{2,2}"},
    }
};
```

Her definerer vi at det skal være et objekt og at det skal indeholde fem properties med hver deres constraints.

For at teste om skemaet er sat rigtigt op, og at de forskellige constraints fungerer, har vi lavet et objekt i NodeJS i stedet for at skulle sende et fra vores frontend.

Her ses vores test objekt, og i bunden skriver vi resultatet af valideringen ud:

```
var hairstylistBooking = {
   "serviceId": 1,
   "emplyeeId": 1,
   "customerId": 1,
   "startTime": "2000-01-01 00:00:00",
   "endTime": "2000-01-01 00:00:00",
};

console.log(v.validate(hairstylistBooking, schema));
```

[13] https://www.npmjs.com/package/jsonschema

12.1.4.3 PHP cURL[14]

Vi havde for brug en måde at kunne kommunikere mellem websitet og vores Motor, vi ved fra vores NodeJS applikation at vi lytter efter HTTP requests på port 3000. Så vores krav til det værktøj som skal sørge for forbindelsen mellem websitet og Motoren er at den skal understøtte HTTP og vi skal have muligheden for selv at angive porten.

Det første værktøj vi fandt frem til var PHP's implementering af cURL[15], libcurl[16].

cURL gør det muligt for os at kunne overføre data mellem serverne via mange forskellige protokoller for eksempelvis HTTP, HTTPS, FTP, FTPS og mange flere, ligeledes kan vi også bestemme hvilken port vi sender på, så kort sagt opfylder cURL vores krav og som et plus er det allerede implementeret som et modul i PHP.

I nedenstående kodeboks har vi sat nogle kommentarer som forklare lidt om de cURL funktioner vi gør brug af, vores kommentarer er på engelsk.

```
//Returns a new cURL handle with the specified URL
$curlHandle = curl_init('http://IP:3000' . $Designation);

if (!$curlHandle){
    //Sets the request to be of POST
    curl_setopt($curlHandle, CURLOPT_CUSTOMREQUEST, "POST");
    //Sets the body of the request to contain the JSON data
    curl_setopt($curlHandle, CURLOPT_POSTFIELDS, $serializedObject);
    //By setting this to true we require cURL to give us the return value
through curl_exec instead of printing the result on the page.
    curl_setopt($curlHandle, CURLOPT_RETURNTRANSFER, true);
    //Sets the HTTP header, we set which content type we use and the length
of the data we're sending
    curl_setopt($curlHandle, CURLOPT_HTTPHEADER, array(
        'Content-Type: application/json',
        'Content-Length: ' . strlen($serializedObject))
    );
    return curl_exec($curlHandle);
}
return false;
```

12.2 Tirsdag

12.2.1 Daily Scrum

Frem til i dag har vi arbejdet på at få websitet til at kommunikere med vores motor vis PHP cURL, og vi har arbejdet på at validere JSON i backenden.

I dag skal vi arbejde videre med kommunikationen mellem website og motor, og validering af JSON i backenden. Vi skal også i gang med at skrive Unit Tests til User Story 1 hvis JSON valideringen bliver færdig. NodeJS er stadig nyt men det er ved ikke at være en forhindring længere.

[14] http://php.net/manual/en/intro.curl.php
[15] http://curl.haxx.se/
[16] http://curl.haxx.se/libcurl/

12.2.2 Udvikling

12.2.2.1 JSON Validering

Efter at have arbejdet videre med at validere den JSON vi modtager når der skal oprettes en booking, er vi kommet frem til nogle forbedringer.

Bl.a. har vi tilføjet et 'required' felt i det JSON skema den indkommende JSON bliver holdt op imod. Det er med til at sikrer at de properties der er tilføjet i 'required', rent faktisk også er til stede i det JSON vi modtager.

Her ses det opdaterede skema, med 'required' feltet til sidst. De regex patterns vi bruger til at sikre at StartDate og EndDate er i rigtig format, er også blevet opdateret, da de ikke virkede helt korrekt før.

```
var hairStylistSchema = {
   "type": "object",
   "properties": {
       "HairStylistServiceId": {"type": "integer", "minimum": 1},
       "EmployeeId": {"type": "integer", "minimum": 1},
       "CustomerId": {"type": "integer", "minimum": 1},
       "StartDate": {"type": "string", "pattern": "^[0-9]{4}-[0-9]{2}-[0-
9]{2} [0-9]{2}:[0-9]{2}:[0-9]{2}$"}, //pattern = 'YYYY-MM-DD HH:MM:SS'
       "EndDate": {"type": "string", "pattern": "^[0-9]{4}-[0-9]{2}-[0-9]{2}
[0-9]{2}:[0-9]{2}:[0-9]{2}$"} //pattern = 'YYYY-MM-DD HH:MM:SS'
   },
   "required": ["HairStylistServiceId", "EmployeeId", "CustomerId",
"StartDate", "EndDate"]
};
```

12.3 Fredag

12.3.1 Daily Scrum

Frem til i dag har vi arbejdet på at få websitet til at kommunikere med vores motor vis PHP cURL, og vi har arbejdet på User Story 1.

I dag skal vi kigge på implementeringen af et system så vi automatisk kan deploy direkte fra vores GIT repository til vores server, vi skal arbejde videre med PHP delen til User Story 1, vi skal kigge på Mocha til NodeJS og skrive Unit Tests, og hvis Unit Tests bliver færdige skal vi i gang med backend delen af User Story 2. Mocha kan blive en forhindring da vi ikke kender modulet eller måden at lave Unit Tests på i NodeJS.

12.3.2 Udvikling

12.3.2.1 Automatic Deployment

Vi synes det vil være smart hvis vi kunne opsætte vores miljø til automatisk at tage de nyeste commits fra vores Git repository og lægge dem over på vores server, derved kan vi spare en masse tid.

De forskellige implementeringer vi havde kigget på er.

- Git Hooks[17]
    - Med Git Hooks kan vi lægge forskellige scripts på serveren som hoster vores git repository, hvert script bliver eksekveret efter et bestemt event.

      Eksempel: Efter et commit kan et script køres, efter et push m.m.

      Problemet med denne implementering er at Visual Studio Online, som også hoster vores git repositories ikke tillader Git Hooks på nuværende tidspunkt.

- Visual Studio Online
    - Visual Studio Online tilbyder en deploy service, den er ret fancy idet der er mange valgmuligheder, dog er det straks mærkbart at de promoverer deres Azure cloud.

      Der er dog også mulighed for at overføre filerne via cURL men problemet er at det ikke er muligt at overføre flere filer på én gang via deres grænseflade. For at det skal virke for os skal vi have fuld kontrol over hvilke mapper og filer som skal overføres til vores produktions server.

- Travis CI[18]
    - Med Travis-CI kunne vi lægge hele vores test og deployment ét sted, hos Travis CI.

      Travis CI ville være den ideelle løsning for os fordi den kan lytte efter events på vores git repository og pull koden, køre test, og baseret på resultatet kan den også deploy til vores produktions server. Vi kan også se resultatet af hver unit test.

      For at implementere Travis CI skal vi først flytte vores kodebase væk fra Visual Studio Online og over til GitHub og derefter synkronisere vores Travis CI med projektet på GitHub.

Efter at have undersøgt de forskellige måder at implementere et automatisk deployment system må vi konstatere at det kommer til at tage for lang tid i forhold til vores tidsplan og værdi for projektet.

[17] https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
[18] https://travis-ci.com/

12.3.2.2 Unit Testing – Mocha[19]

NodeJS modulet Mocha er et test framework til NodeJS. Mocha har utrolig mange features og er et udbredt test framework til bl.a. Test Driven Development i NodeJS. Vi vil benytte Mocha til vores Unit Tests i server backenden.

Før vi går i gang med at skrive tests til User Story 1 har vi fundet en tutorial[20] der beskriver installationen og konfigurationen af Mocha, og demonstrerer brugen af Mocha på et simpelt program. Vi vil gå igennem nogle af punkterne i tutorialen for at få mere styr på Mocha, og unit testing i NodeJS.

12.4 Tirsdag

12.4.1 Daily Scrum

Frem til i dag har vi arbejdet på User Story 1, både frontend og backend og skrevet Unit Tests dertil.

I dag skal vi færdiggøre User Story 1 og de Unit Tests der hører til. Der er stadig lidt detaljer der skal på plads omkring Mocha så det kan blive en forhindring.

12.4.2 Udvikling

12.4.2.1 Unit Tests – JSON Validator

For at teste om vores JSON validering virker efter hensigten, har vi opstillet fire test cases der har til formål at simulere indkomne requests med forskellig data.

De fire test cases er vist i skemaet herunder:

Test Case:	**Fremgangsmåde:**	**Forventet Resultat:**	**Resultat:**
Test Case 1	Send korrekt JSON og korrekt booking type.	Valideringen er succesfuld og returnerer True.	Validering succesfuld, og der blev returneret True. ✓
Test Case 2	Send forkert JSON og korrekt booking type.	Valideringen fejler og returnerer False.	Validering fejlet, og der blev returneret False. ✓

[19] https://mochajs.org/
[20] https://semaphoreci.com/community/tutorials/getting-started-with-node-js-and-mocha

Test Case 3	Send korrekt JSON og forkert booking type.	Valideringen fejler og returnerer False.	Validering succesfuld, og der blev returneret True. ✗
Test Case 4	Send forkert JSON og forkert booking type.	Valideringen fejler og returnerer False.	Validering fejlet, og der blev returneret False. ✓

Her ses de 4 Test Cases fra NodeJS:

```
describe("Test Case 1", function(){
    it("Checks ValidateJson with correct JSON and correct bookingType",
function(){
        assert.equal(validator.ValidateJson(dataCorrect, "HairStylist"), true);
    });
});

describe("Test Case 2", function(){
    it("Checks ValidateJson with incorrect JSON and correct bookingType",
function(){
        assert.equal(validator.ValidateJson(dataIncorrect, "HairStylist"),
false);
    });
});

describe("Test Case 3", function(){
    it("Checks ValidateJson with correct JSON and incorrect bookingType",
function(){
        assert.equal(validator.ValidateJson(dataCorrect, "asdasdasd"), false);
    });
});

describe("Test Case 4", function(){
    it("Checks ValidateJson with incorrect JSON and incorrect bookingType",
function(){
        assert.equal(validator.ValidateJson(dataIncorrect, "asdasdasd"), false);
    });
});
```

Efter at have kørt de fire tests kan vi se at vi har et problem ved test case 3 eftersom at resultatet ikke er hvad vi forventede, og den fejlede.

I vores ValidationController.js gør vi brug af en switch statement der sørger for at det rigtige JSON skema bliver brugt til validering, alt efter hvilken booking type der er gældende. Problemet med vores switch er at vi ikke har en default case, så hvis den modtager en booking type der ikke er defineret, risikere vi at der bliver kørt noget kode som ikke burde køres, hvilket har været tilfældet her.

Manglende default case er altså grunden til at Test Case 3 fejlede. Vi har derfor tilføjet en default case der sørger for at tage hånd om ukendte booking typer, og kan nu køre vores Test Case 3 igennem igen.

Test Case 3	Send korrekt JSON og forkert booking type.	Valideringen fejler og returnerer False.	Valideringen fejlede og returnerede False. ✓

12.4.2.2 PHP JSON Mapping

Vi havde nogle problemer med at strukturere objekterne i PHP fordi objektet skulle serialiseres til JSON og sendes til vores motor. Problemet lå i at vores PHP objekt HairStylistBooking indeholdte flere objekter såsom Service objekt, Employee objekt m.m. og når man serialisere et PHP objekt, så er det hele objektet man serialiserer.

Vi vil ikke serialisere hele Service og Employee objekterne, da vi kun interesseret i deres ID så vi kan referere til dem i databasen.

Desværre er der ikke lige så stor fleksibilitet angående mapping af et objekt til JSON i PHP som der eksempelvis er i C#.

Vi fandt dog frem til at PHP har en tom, generisk klasse ved navn stdClass. Dvs. vi kan ved runtime selv definere hvilke properties klassen skal have.

Vores løsning er at lave en metode i HairStylistBooking klassen som har til ansvar at oprette en instans af stdClass, tilføje de nødvendige properties, udfylde dem og derefter konvertere objektet til json og returnere resultatet.

```
public function GetJson(){
    $tempObj = new StdClass();
/*At this sprint we do not have any proper object, Test data = 1
    $tempObj->ServiceId = $this->Service->Id;
    $tempObj->EmployeeId = $this->Employee->Id;
    $tempObj->CustomerId = $this->Customer->Id;
*/
    $tempObj->HairStylistServiceId = 1;
    $tempObj->EmployeeId = 1;
    $tempObj->CustomerId = 1;
    $tempObj->StartDate = $this->StartDate;
    $tempObj->EndDate = $this->EndDate;
    $tempObj->Refer = $this->Refer;

    return json_encode($tempObj);
}
```

12.5 Onsdag

Frem til i dag er vi blevet færdige med User Story 1 or stort set færdige med Unit Tests dertil. Vi er også begyndt at implementere muligheden for at udtrække og vise bookinger i frontend, backend og database.

I dag skal vi færdiggøre Unit Tests, udtræk og visning af bookinger. Vi skal også påbegynde User Story 2, opdater og slet booking. Der er på dette tidspunkt ikke noget der forhindre os i at komme videre.

12.5.1 Udvikling

12.5.1.1 Stored Procedures

Vi har valgt at al kommunikation mellem vores NodeJS applikation og databasen skal foregå ved at NodeJS applikationen kalder stored procedures i databasen.

Der er flere grunde til vi har valgt denne fremgangsmåde.

Abstraktion & Low coupling

Ved at vi lægger alt SQL logik i databasen fjerner vi den high coupling som er til stede når man skriver sin SQL logik i sit program.

Det betyder også at vi fremover hvis ønsket, lettere kan udskifte vores DBMS med et andet DBMS så længe det dog understøtter stored procedures eller anden form for at gemme funktioner af logik på databasen.

I vores applikation kan vi derfor nøjes med at udskifte den driver vi bruger til at forbinde til vores DBMS.

Sikkerhed

Der er et problem at hvis en tredjepart får adgang til vores software så kan de se de oplysninger vi bruger til at forbinde til databasen med, derved have samme rettigheder som den database bruger vi logger ind med.

Med MySQL stored procedure og specifikt, attributterne DEFINER og SQL SECURITY kan vi indkapsle eksekveringen af SQL logikken i en anden brugers kontekst.

Det vil sige, vi har en bruger i vores NodeJS applikation som absolut kun har rettigheder til at forbinde til vores DBMS og eksekverer stored procedures og intet andet, denne bruger kan fx ikke select, update, delete eller noget andet.

Så hvis en person udefra får adgang til vores software og vil prøve at lave bøvl i vores DBMS, vil det ikke være muligt.

Det er her DEFINER og SQL SECURITY kommer ind i billedet. Vi har en ny bruger som kun ligger på databasen og som man ikke kan forbinde til udefra, denne bruger har derimod flere rettigheder, vi har derfor oprettet en stored procedure og via DEFINER defineret at denne stored procedure skal eksekveres i denne brugers kontekst.

Derfor når vores NodeJS applikation gerne vil eksekvere en stored procedure på databasen, så vil den stored procedure eksekveres i database brugerens sikkerheds kontekst.

Er det stadigt muligt at eksekvere SQL Injections?
Ja, hvis der er snavs i den SQL data vi sender til vores stored procedure så eksekveres det stadigt i konteksten af brugeren med flere rettigheder, derfor er det stadigt kritisk nødvendigt at vi validere SQL parametrene inden de sendes afsted til databasen.

Det gør vi ved at "escape" vores SQL query værdier, her er et fra vores DatabaseController fra NodeJS applikationen.

Her kan vi se at funktionen query gives en SQL streng hvori hver værdi er udskiftet med et spørgsmålstegn, derudover får query også en liste af de værdier som skal indsættes, i den rækkefølge de står i.

Herfra sørger vores database driver at "escape" værdierne og indsætte dem på deres rette plads, dermed undgår vi SQL Injections.

```
this.pool.query('CALL AddHairStylistBooking(?, ?, ?, ?, ?)',
[
bookingObject.HairStylistServiceId,
bookingObject.EmployeeId,
bookingObject.CustomerId,
bookingObject.StartDate,
bookingObject.EndDate
]
```

Fejlbeskeder
Stored procedures giver os også mulighed for at kunne opfange hvorvidt en stored procedure er succesfuld eller fejler og hvilke fejl der måtte opstå.

Det resultat kan vi analysere og selv bestemme hvordan det skal returneres. Vi har eksempelvis valgt at på INSERT returnerer vi primærnøglen på den akkurat indsatte kolonne eller -1 på fejl, på UPDATE returnere vi 1 for success og -1 for fejl.

Dette er kun et basis, vi kan uddybe det og returnere forskellige fejlbeskeder vi selv bestemmer baseret på hvordan den givne stored procedure blev eksekveret.

Udover det har vi også mulighed for at gøre brug af transaktioner, hvis vi eventuelt gerne ville eksekvere flere SQL statements i samme stored procedure, eller eksekvere flere stored procedures efter hinanden, så kan vi gruppere dem i en transaktion, analysere resultatet og hvis der var fejl undervejs kan vi rulle den transaktion tilbage og returnere en passende fejlbesked.

12.5.1.2 MySQL og Server Time Zones
Vi har valgt at bruge datetime formatet YYYY-MM-DD HH:MM:SS som kan se således ud (2015-12-02 15:00:00).

Vi har valgt ovenstående da vi så kan gøre brug af de datoobjekter som findes i de sprog vi arbejder med, og bruge de indbyggede funktioner til at ændre og beregne tidspunkter.

Vi løb ind i et problem, hvor vi via hjemmesiden valgte at booke klokken 08:00 der blev indsat korrekt i databasen, men når vi tog datoen ud igen var den ændret, til 13:00.

Efter megen søgen og efter at have læst en del af MySQL's dokumentation kom vi frem til at MySQL konverterer datetime baseret på serverens tidszone.

Vi ændrede derfor serverens tidszone til UTC med kommandoen: sudo dpkg-reconfigure tzdata.

Det virkede dog ikke selvom vi genstartede maskinen og sørgede for at MySQL kørte efter den nye tidszone som vi tjekkede med kommandoerne.

SELECT @@global.time_zone;

Her kan vi se hvilken tidszone MySQL lytter efter, som i vores instans er SYSTEM som også er ønsket.

SELECT NOW();

Her fik vi dato og tidspunkt for MySQL instansen, den kunne matche op imod datoen på serveren og se de er ens.

Vi prøvede nu at ændre tidszonen på vores server som kører vores NodeJS applikation, til vores store overraskelse virkede det.

Det vil sige at det er serveren som laver *SELECT* statement som afgør hvilken tidszone datetime objektet konverteres til.

12.6 Fredag

12.6.1 Daily Scrum

Til i dag har vi arbejdet på at implementere muligheden for at slette og opdatere en booking, vi er også næsten blevet færdige med at implementere udtræk og visning af bookinger.

I dag skal vi arbejde videre på at implementere opdater og slet booking, og vi skal færdiggøre udtrækning og visning af bookinger. Vi skal også have lavet Unit Tests til User Story 2. Vi har ingen forhindringer til dagens arbejde.

12.6.2 Udvikling

12.6.2.1 Backend – Opdater og Slet booking

Vi gør brug af samme struktur til at opdatere eller slette en booking, som når der skal tilføjes en booking. Vi modtager et http request på den adresse der svarer til den handling der skal ske, 'Booking/Delete/HairStylist' når en booking skal slettes og 'Booking/Update/HairStylist' når en booking skal opdateres.

Vi kalder derefter tilsvarende funktioner i vores BookingController, UpdateBooking eller DeleteBooking, der så sørger for at få valideret JSON objektet og efterfølgende sender det videre til vores DatabaseController så opdateringen eller sletningen også bliver sendt til databasen.

Ligesom ved tilføj booking, validerer vi opdater og slet ved at holde det indkommende JSON op imod nogle prædefinerede JSON skemaer der ser således ud:

```
var hairstylistDeleteSchema = {
    "type": "object",
    "properties": {
        "HairStylistBookingId": { "type": "integer", "minimum": 1 },
    },
    "required": ["HairStylistBookingId"]
};

var hairstylistUpdateSchema = {
    "type": "object",
    "properties": {
        "HairStylistBookingId": { "type": "integer", "minimum": 1 },
        "HairStylistServiceId": { "type": "integer", "minimum": 1 },
        "EmployeeId": { "type": "integer", "minimum": 1 },
        "CustomerId": { "type": "integer", "minimum": 1 },
        "StartDate": { "type": "string", "pattern": "^[0-9]{4}-[0-9]{2}-[0-9]{2}
[0-9]{2}:[0-9]{2}:[0-9]{2}$" }, //pattern = 'YYYY-MM-DD HH:MM:SS'
        "EndDate": { "type": "string", "pattern": "^[0-9]{4}-[0-9]{2}-[0-9]{2}
[0-9]{2}:[0-9]{2}:[0-9]{2}$" } //pattern = 'YYYY-MM-DD HH:MM:SS'
    },
    "required": ["HairStylistBookingId", "HairStylistServiceId", "EmployeeId",
"CustomerId", "StartDate", "EndDate"]
};
```

12.6.3 Retrospective

Bliv ved med at

- Mødes hver dag, med faste mødetider.
- Have en god dialog både om udvikling/fremgangsmåder samt værktøjer både indenfor systemudvikling og softwareudvikling.

Mere af

- Mere fokus på at opdatere backloggen løbende.

12.7 Konklusion – Sprint 2

Efter at have gennemført andet sprint, er vi ved at være godt på plads i arbejdsrytmen. Vi har især kunnet mærke en forskel på at arbejde med to ugers sprint, i stedet for en uges sprint, det har givet meget mere tid til at fokusere på at skabe værdi efter sprintet, hvor vi før skulle bruge for meget af tiden på at planlægge og afslutte sprintet.

13 Tekniske beslutninger

Baseret på vores erfaringer fra sprint 1 og 2, har vi valgt at holde et teknisk møde om hvorvidt brugen af MySQL er den smarteste løsning baseret på det produkt vi udvikler.

Vi har valgt at udskifte MySQL med MongoDB.
Hvorfor vi har valgt at tage så stor en beslutning kan læses herunder.

13.1 Database

Da vi brugte MySQL havde vi en separat tabel for hver type af booking, det er ikke så smart fordi nogle af felterne som er generelle for hver type af booking går igen.

Det betød også at hver gang vi skulle kunne håndtere en ny type af booking så skulle vi i databasen tilføje to tabeller for den ny type af booking,selve booking tabellen og en tabel der indeholdte services som den type af booking kunne håndtere.

Fordi at MongoDB er dokumentbaseret betyder det for os at vi har større mulighed for at kunne håndtere generisk data, det vil sige i vores tilfælde kan vi have alle vores bookinger i én tabel og alle vores services i en anden tabel.

Det skal ikke forstås således at MongoDB er bedre end MySQL, hver database har sine fordele og ulemper og en dårlig optimeret MongoDB er lige så dårlig som en dårlig optimeret MySQL.

Men i vores situation hvor vi skal håndtere generisk data er MongoDBs dokumentbaseret tilgang bedre end MySQLs skema tilgang.

Hvis vi skulle tilføje en ny type af booking ville det i MySQL kræve at vi oprettede en ny tabel men i MongoDB kan vores booking tabel godt indeholde den nye type af booking selvom den har nye felter, dvs. fremadrettet kræver MongoDB mindre vedligeholdelse end MySQL i vores situation.

Dog skal vi huske på at når data i databasen er generisk så skal vi i applikations logikken stadig holde styr på hvilken data som indsættes, hentes og opdateres. Det nytter ikke at vi ikke ved hvordan dataen er struktureret, det kræver at vi i et felt i booking tabellen definere hvilken type af booking det er, så når vi eksempelvis skal fremvise dataen for frisøren så er det de rigtige felter som vises i de rigtige HTML form elementer.

13.2 Kode

Vores skift til MongoDB betyder også at vi kan refaktorere en del af vores applikations logik.

Motoren

På motoren hvor vi lytter efter indkommende HTTP requests lytter vi også specifikt efter booking typen det vil sige vores nuværende lytter ser således ud.

```
app.post('/Booking/Add/HairStylist', function (request, response) {...}
app.post('/Booking/Get/HairStylist', function (request, response) {...}
app.post('/Booking/Delete/HairStylist', function (request, response) {...}
app.post('/Booking/Update/HairStylist', function (request, response) {...}
```

Skrækscenariet her er at for hver ny type af booking skal vi tilføje N antal nye HTTP lyttere, det kræver meget vedligeholdelse.

Vi skal, vide hvilken type booking som sendes ind for at kunne validere dataen med det rigtige JSON skema, men i stedet for at bruge URL'en til at se på hvilken type af booking vi får ind så kan vi gøre brug af HTTP headers, vi kan tilføje vores egen headers til HTTP pakken som specificerer hvilken type af booking vi sender.

Derfor ville vores HTTP lyttere se således ud.

```
app.post('/Booking/Add', function (request, response) {...}
app.post('/Booking/Get', function (request, response) {...}
app.post('/Booking/Delete', function (request, response) {...}
app.post('/Booking/Update', function (request, response) {...}
```

Denne optimering af Motoren er ikke kommet af at vi skifter fra MySQL til MongoDB.
Optimeringen ville gøre gavn i begge scenarier.

13.3 Stored Procedures

Da vi brugte MySQL gjorde vi brug af Stored Procedures fordi det gjorde det muligt for os at opdele applikation logik og database logik.

MySQL Stored Procedure gav os også en forøget sikkerhed, da vi kunne eksekvere en Stored Procedure fra en bruger som akkurat havde tilladelse til at logge ind og eksekvere Stored Procedures, men selve eksekveringen blev kørt i en anden brugers sikkerheds kontekst.

Det betød for os at hvis en udefrakommende fik adgang til vores kode ville personen ikke have gavn af login oplysninger til databasen fordi personen kun kunne køre de stored procedures der er tilgængelige.

Der er desværre ikke noget lignende i MongoDB. I MongoDB er det muligt at gemme JavaScript funktioner men det er ikke muligt at kalde funktionerne udefra, ikke i nuværende version af MongoDB 3.2.

I tidligere versioner af MongoDB var det muligt at kalde JavaScript funktioner udefra via MongoDBs *db.eval()*[21] funktion, men den er blevet forældet og understøttes ikke længere.

13.4 Hvem kan sende data til Motoren?

Lige nu kan alle sende data til vores Motor, det er absolut ikke optimalt.

Vi har derfor haft en dialog om hvilke fremgangsmåder vi kan tage for at sikre os at dataen kun kan komme fra hjemmesidens server.

Først snakkede vi om at opsætte en firewall som kun tillod indkommende forbindelser fra tilladte ip'er.

Problemet med den løsning er at hvis en angriber udefra fik fat i websidens IP og Motorens IP. Så ville den angribende kunne sende/opdatere/slette bookinger ved at se ud som om at pakkerne kom fra webserveren, det kunne angriberen gøre ved at ændre IP datagrammets Source Address til websitets IP.

Angriberen ville dog ikke kunne få svar tilbage da de vil blive sendt til Source Address som er websitets IP. Men at sende/opdatere/slette bookinger er slemt nok i sig selv.

Den anden løsning er at vi laver et simpelt authentication token system.

Det vil sige at websiden får en genereret token og den token er også gemt i databasen.

Så skal hjemmesiden sende sin token med i requesten hver gang, og så kan Motoren matche den token med den som ligger i databasen.

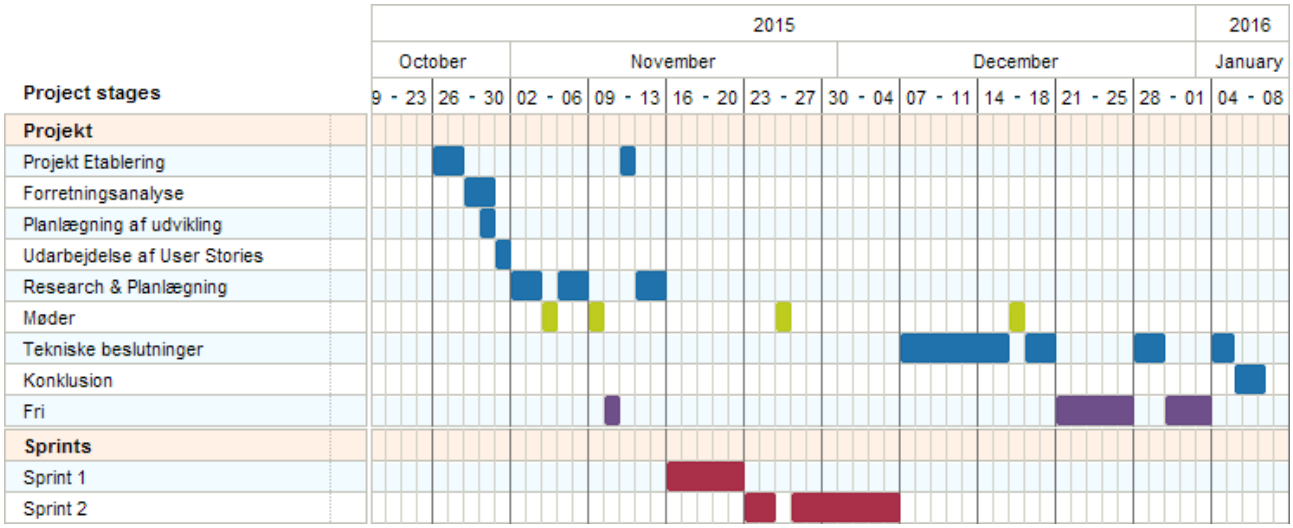
Dette system er simpelt til at starte på men det åbner op for muligheden at tilføje avancerede features fremover som måtte de blive nødvendige, eksempelvis.

- Tokens skal fornyes i et givent tidsinterval
- Token associeret med en specifik host
- Gøre brug af public-private keys.

13.5 Gantt Chart

Efter valget om at skifte database har vi også opdateret vores Gantt Chart, for at kunne holde overblikket over den resterende tid. Vi har desværre ikke tid til endnu et sprint, så vi fokuserer på arbejde med databasen, og at få helt styr på de ikke funktionelle krav.

[21] https://docs.mongodb.org/manual/reference/method/db.eval/



13.6 Backend

13.6.1 Refaktorering til MongoDB

For at kunne håndtere MongoDB kræver det også nogle ændringer i NodeJS motoren. De fleste ændringer er i vores DatabaseController da det er der, der kommunikeres med databasen.

Vi kan ikke længere gøre brug af stored procedures eller lignende, vi skal derfor fremover sørge for at oprette forbindelse til databasen, udføre de gældende database kald og så lukke forbindelsen igen.

De simple databasekald 'insert', 'find', 'update' og 'remove' minder meget om dem fra MySQL, den største forskel er når der skal indsættes data. I stedet for at skulle specificere dataene til hver kolonne som man gør i MySQL, kan vi i MongoDB indsætte et helt JSON objekt direkte.

Her ses det hvordan vi indsætter et booking objekt i vores MongoDB database:

```
DatabaseController.prototype.AddHairStylistBooking = function(bookingObject){
    MongoClient.connect("mongodb://IP", function(err, db) {
        if(err) { return console.dir(err); }
        console.log("Connected to MongoDB");
        var collection = db.collection('Booking');

        //Using the {w:1} option ensure you get the error back if the document
fails to insert correctly
        collection.insert(bookingObject, {w:1}, function(err, result) {
            if(err) { return console.dir(err); }

            //SUCCESS!
            console.log(result);
            db.close();
            console.log("Closed connection to MongoDB");
        });
    });
}
```

Vi forbinder til databasen med 'MongoClient.connect("mongodb://IP", function(){}', hvis der sker fejl under oprettelsen af forbindelse sørger vi for at håndterer fejlen, ellers går vi videre til at specificere hvilken collection bookingen hører til. Derefter laver vi database kaldet 'insert' og hvis der ikke sker fejl ved indsættelsen af bookingen skriver vi resultatet ud til konsollen, og lukker så forbindelsen til databasen.

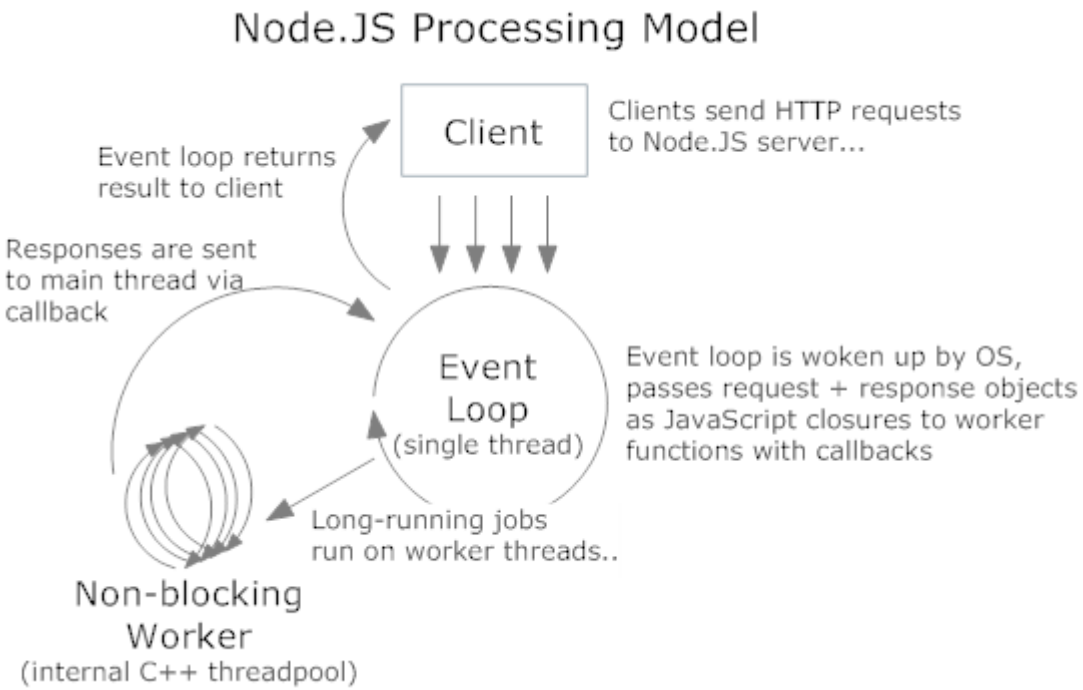
13.6.2 Callback

Indtil nu havde vi ikke overvejet hvordan NodeJS asynkroniske opbygning ville påvirke vores kode.

Først skrev vi vores kode synkronisk som gav os nogle problemer, eksempelvis havde vi en metode som kaldte en database metode som skulle hente nogle resultater fra databasen og returnere dem. Det problem vi stødte på var at den metode som kaldte database metoden ville returnere inden database metoden havde fået resultatet fra databasen.

Det betød at vi ikke havde nogen garanti for hvilken rækkefølge koden ville eksekveres med den kode vi havde skrevet.

Grunden for det er den måde NodeJS er opbygget på, vi kan eksempelvis godt starte med at hente et resultat fra databasen men i stedet for at vente på resultatet så vil NodeJS delegere arbejdet ud i en af sine interne tråde og så vil main tråden eksekvere næste opgave.



[22]

I billedet ovenover kan vi se at en klient sender en request til NodeJS applikationen, det spændende som man ikke umiddelbart lige kan se er opbygningen af NodeJS Event Loop.

Hver gang der kommer en ny request bliver den lagt nederst i NodeJS Event Queue.

[22] http://stackoverflow.com/a/14797071

NodeJS Event Loop er enkelttrådet og tager fat i en request af gangen, den som ligger øverst i Event Queue.

Requesten bliver eksekveret i NodeJS interne threadpool, når en af trådene har klaret opgaven som eksempelvis at hente resultatet fra databasen så vil tråden ligge dens callback i bunden af Event Queue.

13.6.2.1 Hvad gik der galt i vores kode

Efter vi fik analyseret hvordan NodeJS gør brug af asynkronisk programmering kan vi straks se hvor vores fejl ligger.

Vi brugte ikke callbacks, vi skrev alt vores kode synkront som resulterede i at vi lagde en masse blokke i vores event queue uden at fortælle NodeJS om hvilke blokke der skulle vente på andre blokke. Derfor returnerede nogle af vores metoder tomt og uden at vente på resultatet fra I/O operationer.

For at rette den fejl har vi måtte lave en mindre refaktorering af vores kode så vi gør brug af callbacks. Nu venter de rette metoder på at få et resultat som de derefter kan returnerer.

13.6.3 MongoDB – DBRefs & Manual References

MongoDB gør ikke brug af joins ligesom MySQL, så for at forbinde dokumenter, collections eller databaser gør man brug af 'Manual References' eller 'DBRefs'.

Med Manual References gemmer man _id feltet fra et dokument i et andet dokument som reference, lidt ligesom foreign keys i MySQL. Når man så skal have dataene ud, skal man kører en ekstra query på applikationen der returnerer det gældende data.

DBRefs refererer fra et dokument til et andet ved hjælp af det første dokuments _id felt, collection navnet og database navnet, hvor database navnet er valgfrit. Med disse informationer kan DBRefs nemmere forbinde dokumenter fra flere collections til en enkelt collection.

DBRefs er en fælles måde at repræsentere forholdet mellem dokumenter, men ens applikationen skal udføre yderligere queries for at returnere de refererede dokumenter.

Hvis ikke der er en bestemt grund til at bruge DBRefs anbefales Manual References[23].

13.7 Tekniske beslutninger delkonklusion

Det essentielle i vores bookingsystem er at det kan håndtere flere typer af bookinger, så selvom det tog en del længere tid end forventet at udskifte databasen, synes vi at det var tiden værd.

Fremadrettet har vi en database løsning der kan håndtere generisk data meget bedre, end hvad vi kunne designe med brugen af MySQL. Skiftet har også gjort det muligt at gøre vores backend en hel del mere generisk. Før krævede backenden stor vedligeholdelse hvis der skulle tilføjes nye booking typer, hvor vi nu blot behøver at tilføje nye skabeloner for hvordan det nye booking objektet skal se ud.

[23] https://docs.mongodb.org/manual/reference/database-references/#dbref-explanation

Denne store ændring i systemet, og erfaring med for korte sprints, har også betydet at vi må se i øjnene at vi ikke kan nå flere sprints i dette projekt. Fremadrettet vil vi derfor fokusere på vores arkitektur, og at arkitekturen overholder de ikke funktionelle krav.

14 High-Availability

Eftersom vi er kommet til et punkt hvor vi har kommunikation mellem hjemmesiden og vores motor synes vi det er rette tidspunkt at implementere en load balancer da vi har nok af systemet færdigt til at kunne teste en load balancer i funktion. Vi vil også gerne sikre os at vi overholder de ikke funktionelle krav vi har sat til systemet.

14.1 Load Balancer

For at vores motor skal kunne skaleres ud samt for at vi skal undgå at hvis vores app server går ned så går servicen ned, så skal vi implementere en load balancer.

Baseret på den tid der er tilbage til at projektet skal leveres har vi ikke mulighed for at undersøge markedet for forskellige løsninger men derimod har vi valgt at opstille nogle simple kriterier og hvis en given load balancer løsning møder de kriterier vil vi implementere den løsning.

De kriterier vi har sat er:

- Softwaren skal være gratis
- Skal være relativ nem at opsætte og konfigurerer
- Skal være kendt og brugt i flere sammenhænge

Vi havde originalt tænkt os at bruge en DNS og lade den skiftevis sende indkomne forespørgsler mellem vores servere, men da DNS er ret begrænset funktionsmæssigt i brugen som en load balancer vil vi prøve at finde en simpel load balancer som kan lidt mere.

Efter kort at have kigget på mulige løsninger stødte vi på HAProxy[24].

Først så vi på hvorvidt HAProxy mødte vores kriterier.

- Softwaren skal være gratis
  - Ja, det er gratis.
- Skal være relativ nem at opsætte og konfigurerer
  - Baseret på noget af dokumentationen og artikler som vi har læst så synes vi HAProxy virker nemt at opsætte og konfigurere
  - Dokumentation: HAProxy Dokumentation[25]
  - Artikel: An Introduction to HAProxy and Load Balancing Concepts[26]

[24] http://www.haproxy.org/
[25] http://www.haproxy.org/#docs
[26] https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts

- Artikel: How To Use HAProxy to Set Up HTTP Load Balancing on an Ubuntu VPS[27]

- Skal være kendt og brugt i flere sammenhænge
  - Baseret på HAProxy hjemmeside under menupunktet They use it! kan man se flere store firmaer som bruger HAProxy, eksempelvis. Twitter og DISQUS.

14.2 HAProxy

Efter at have fulgt artiklerne 'An Introduction to HAProxy and Load Balancing Concepts' og 'How To Use HAProxy to Set Up HTTP Load Balancing on an Ubuntu VPS' har vi successfuldt fået vores load balancer op at køre.

Dataen går nu fra hjemmesiden igennem vores load balancer som balancerer loadet i mellem vores app servere.

Vi har udført nogle simple test for at teste forskellige scenarier med den nye load balancer.

Test Case:	**Fremgangsmåde:**	**Forventet Resultat:**	**Resultat:**
Virker Load Balancer i best case scenario	Starte begge App Servere bag load balancer. Forespørge bookings via website, som skal gå igennem load balancer.	Korrekt booking data	Returnere korrekt booking data ✓
FailOver	Sluk en af app serverne	Load balancer delegerer kun forespørgslen til den aktive server.	Intet svar tilbage ✗
No Service Available	Sluk alle app servere	503 Service Unavailable No server is available to handle this request.	503 Service Unavailable No server is available to handle this request. ✓

[27] https://www.digitalocean.com/community/tutorials/how-to-use-haproxy-to-set-up-http-load-balancing-on-an-ubuntu-vps

Virker Round-Robin	Forespørg bookings flere gange og se i statistikken hvilken server som får forespørgslerne	Statistikken burde vise at serverne skiftevis får forespørgslerne	I statistikken under Bytes(In/Out) kan vi se at serverne skiftevis får og returnere data. ✓

14.2.1 Test Case: FailOver

I vores test case FailOver prøver vi at se hvorvidt om vores load balancer kun leverer forespørgsler til aktive servere. Som bevist i vores test case er det ikke altid tilfældet.

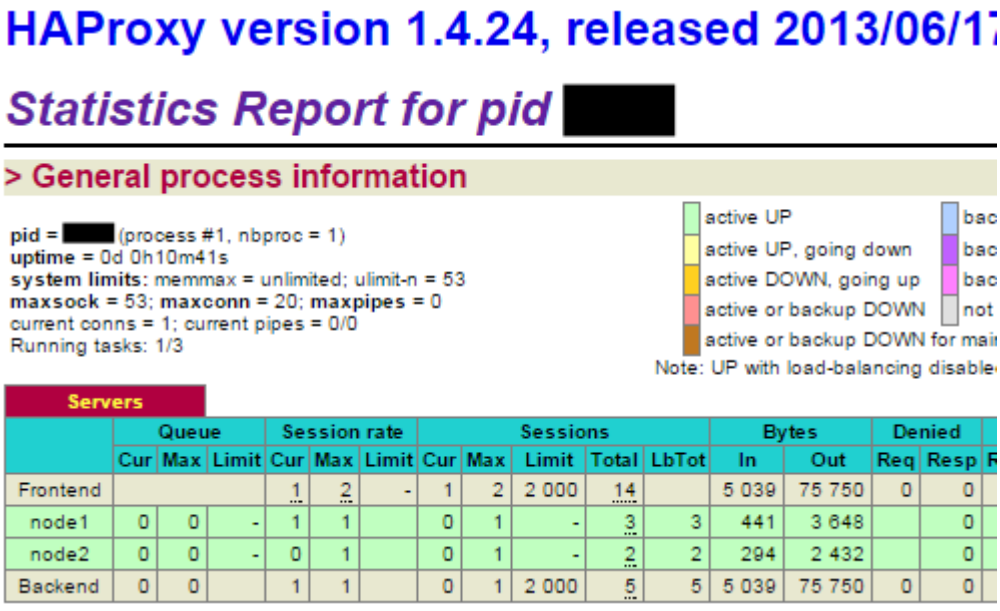
Nogle gange når vores load balancer ikke at lave et healthcheck på en server som er ved at gå ned, i denne sjældne og korte periode kan forespørgsler stadigt blive sendt til den inaktive server.

Dog efter healthcheck vil forespørgsler altid blive sendt til de(n) aktive server(e).

14.2.2 Statistiker

HAProxy gør det også muligt for os at se statistikker for vores to node servere som har vores NodeJS app installeret.

Frontend og Backend er ikke hjemmesiden, det er HAProxy's egen moduler.



HAProxy version 1.4.24, released 2013/06/17

Statistics Report for pid [redacted]

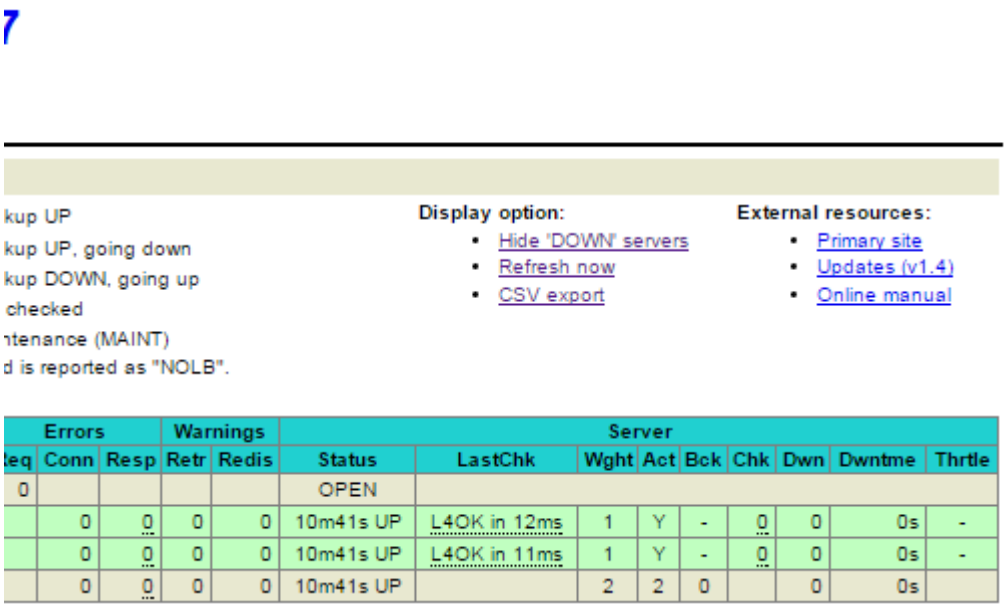
> General process information

pid = [redacted] (process #1, nbproc = 1)
uptime = 0d 0h10m41s
system limits: memmax = unlimited; ulimit-n = 53
maxsock = 53; maxconn = 20; maxpipes = 0
current conns = 1; current pipes = 0/0
Running tasks: 1/3

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for mai
Note: UP with load-balancing disable

Servers

	Queue			Session rate			Sessions					Bytes		Denied		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	R
Frontend				1	2	-	1	2	2 000	14		5 039	75 750	0	0	
node1	0	0	-	1	1		0	1	-	3	3	441	3 648		0	
node2	0	0	-	0	1		0	1	-	2	2	294	2 432		0	
Backend	0	0		1	1		0	1	2 000	5	5	5 039	75 750	0	0	



7

kup UP
kup UP, going down
kup DOWN, going up
checked
ntenance (MAINT)
d is reported as "NOLB".

Display option:
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.4)
- Online manual

Errors			Warnings		Server								
eq	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
0					OPEN								
	0	0	0	0	10m41s UP	L4OK in 12ms	1	Y	-	0	0	0s	-
	0	0	0	0	10m41s UP	L4OK in 11ms	1	Y	-	0	0	0s	-
	0	0	0	0	10m41s UP		2	2	0		0	0s	

Vi vil ikke gå i dybden med hvad de mange forskellige kolonner betyder men nogle få udpluk som vi bruger allerede nu er.

- Bytes(In/Out) - Fortæller os hvor meget data som går ind til og ud af den pågældende server.
- Status - Fortæller os statussen af serveren, vi kan ovenover se en farvepalette med passende statuskoder.
- LastChk - Hvornår sidst den pågældende server blev tjekket. HAProxy tjekker om serverne er online ved at prøve at oprette en TCP forbindelse til serveren.

15 Konklusion

15.1 Spørgsmål 1

- Hvordan er det muligt at lave en skalérbar server arkitektur, uden et Single Point of Failure, der selv kan distribuere arbejdet ud på servere med mindre kapacitet?

For at opnå en skalerbar server arkitektur overvejede vi et par løsninger, men nåede frem til en løsning hvor vi skalerer ud via en HAProxy load balancer og Round-robin algoritmen, og skalerer op via PM2 modulet til NodeJS.

HAProxy load balanceren sørger for at sprede requests ud til flere servere, og eftersom vi gør brug af Round-robin algoritmen bliver de spredt ud sekventielt. HAProxy sørger også for at udføre health checks, for at se om serverne kører.

PM2 gør at vi kan skalere op på selve serveren ved at uddele arbejdet på flere instanser af vores NodeJS applikation. PM2 har sin egen load balancer der også gør brug af Round-robin algoritmen, og på den måde uddeler arbejdet sekventielt på NodeJS instanserne.

15.2 Spørgsmål 2

- Hvordan kan vi udvikle et generelt bookingsystem der kan håndtere forskellige typer af bookinger?

For at vores bookingsystem er så generelt som muligt, og kan håndtere forskellige typer af bookinger, har det været vigtigt at bruge den rette database. Vi startede med at designe og implementere en MySQL database, men nåede frem til at bruge en NoSQL database i stedet.

MySQL databasen blev alt for begrænset og krævede at der blev tilføjet nye tabeller hvis der kom nye bookinger, og samtidig blev backenden også meget statisk da den skulle passe til databasen. I backenden skulle der også tilføjes nye metoder for hver type booking.

Med NoSQL behøver vi ikke at skulle tilføje nye collections til nye booking typer, og derfor behøver vi heller ikke nye metoder for hver booking type. Vi kan i princippet tilføje hvad som helst i databasen, men for at holde en vis standard sørger vi for at backenden bestemmer hvad der må tilføjes i databasen ved at validere indkomne bookinger og holde dem op imod et prædefineret JSON skema.

15.3 Spørgsmål 3

- Hvordan kan vi have kommunikation mellem en hjemmeside og et eksternt system?

For at kommunikere mellem vores frontend og backend gør vi brug af HTTP requests. Fra frontenden sender vi fx et HTTP Post request hvor vi i headeren specificerer hvilken type booking det handler om, og i bodyen er selve JSON dataen der skal tilføjes til databasen.

Når backenden så er færdig med at tilføje, ændre, slette eller hvad den nu skal, kan vi melde tilbage til frontenden, via et HTTP response, med en fejlbesked eller en besked om at handlingen er udført.

16 Perspektivering/Refleksion

16.1 Systemudvikling

Når vi ser tilbage på starten af vores sprints hvor vi skulle vælge hvor mange user stories vi kunne håndtere for et givent sprint, så kan vi godt se nu at det ville have været nemmere hvis vi havde estimeret vores velocity undervejs for at finde ud af hvor meget vi kan nå.

Det ville dog ikke været helt muligt for os, at kunne lave en relativ præcis estimering af vores velocity fordi vi kun havde så lidt tid til projektet, samt i vores tilfælde også få sprints.

Men i et længere projekt synes vi det er en god ide at estimere velocity fordi så ville vi nemmere kunne se hvor meget arbejde vi kan nå at udføre og derfor mere præcist kunne vælge hvor mange user stories vi kan nå at arbejde med i et sprint.

16.2 Forretningsanalyse

Vi synes vi har lavet en god forretningsanalyse dog er der nogle få mangler for at have en solid forretningsanalyse.

Vi kunne eksempelvis have været ude i felten og spørge frisørsaloner hvilke funktioner de synes ville kunne hjælpe dem.

Vi kunne også have allieret os med en eller to frisørsaloner til at teste vores produkt regelmæssigt og have en dialog med en kunde som skal bruge produktet mange gange dagligt.

16.3 Arkitektur

Inden vi påbegyndte denne udviklingsproces havde vi begge begrænset forståelse af at distribuere arbejdet mellem flere servere, samt også hvordan man sikre reliability, high-availability og sikre sig imod SPOF.

Når vi ser tilbage på de valg vi har taget undervejs synes vi at de har været fornuftige baseret på vores krav og hvordan vi har læst os frem til at det kan gøres i et produktionsmiljø.

Vi kan selvfølgelig altid forbedre arkitekturen, eksempelvis er vores load balancer et SPOF lige nu, ligeledes er websiden den eneste indgangsvinkel til vores motor dvs. hvis websiden gik ned ville vores service ikke være til rådighed. Vores database er også et SPOF, hvis den går ned har vi ikke adgang til de nødvendige data.

Man kan også gå til yderligheder at hvad-nu-hvis det datacenter vi gør brug af mistede strømmen, elektriciteten eller anden force majeure.

For at løse de problemer kunne vi implementere en kontrol software på alle serverne som kontrollerede om load balanceren kørte og starte en ny hvis den ikke kørte. Vi kunne også lave en tro kopi af hele vores arkitektur hos vores nuværende datacenter og have den kørende i standby tilstand hos et andet datacenter og have en load balancer foran de to datacenter, den load balancer kunne være en tredjepart som leverede hardware baserede load balancing løsninger.

Vi har på den tid vi har haft udarbejdet en solid arkitektur men vi kunne sagtens gøre løsningen bedre hvis vi havde mere tid.

16.4 Database

Hvis vi ser tilbage på perioden inden vi havde valgt hvilken database vi ville gøre brug af, ville det have været smartere hvis vi havde haft en bedre forståelse af den data som vi skulle gemme og derfor have haft et bedre grundlag for valg af database.

Det problem vi stødte på var at ret sent i udviklingsprocessen måtte vi skifte fra MySQL til MongoDB fordi vi havde generiske booking typer som er nemmere at håndtere i en dokumentbaseret database.

At skifte fra MySQL til MongoDB krævede længere tid end forventet, det inkluderede både opsætning af serveren samt konfigurering men også forståelse for hvordan MongoDB virker og hvordan vi kunne designe databasen.

For at undgå samme fejl fremover ved vi nu af erfaring at det er vigtigt at have en god forståelse af den data man vil gemme så man kan lave en bedre vurdering af den type database man skal bruge.

17 Litteraturliste

https://leantesting.com/resources/test-driven-development/
Forfatter: Simon Hill - Overskrift: The pros and cons of Test-Driven Development
- Dato: 23-02-2015

http://www.amzur.com/pros-and-cons-of-using-test-driven-development-for-web-applications-built-with-ruby-on-rails-ror/
Overskrift: Pros and cons of using Test Driven Development for web applications built with Ruby on Rails (RoR)

https://www.mysql.com/
Forfatter: MySQL

www.vierhairtools.dk
Forfatter: Hairtools

www.timecenter.com
Forfatter: TimeCenter

www.admind.dk
Forfatter: Admind

http://www.vierhairtools.dk/hvem-er-vi/
Forfatter: Hairtools - Overskrift: Hvem er vi?

https://almsaeedstudio.com/themes/AdminLTE/index.html
Forfatter: Abdullah Almsaeed - Overskrift: AdminLTE

https://hadoop.apache.org/
Forfatter: The Apache Software Foundation - Overskrift: Welcome to Apache™ Hadoop®!
- Dato: 18-12-2015

https://nodejs.org/en/about/
Forfatter: NodeJS - Overskrift: About Node.js®

https://www.npmjs.com/package/jsonschema
Forfatter: tdegrunt - Overskrift: jsonschema

http://php.net/manual/en/intro.curl.php
Forfatter: The PHP Group - Overskrift: Introduction

http://curl.haxx.se/
Forfatter: curl.haxx.se

http://curl.haxx.se/libcurl/
Forfatter: curl.haxx.se - Overskrift: libcurl - the multiprotocol file transfer library

https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
Forfatter: git-scm - Overskrift: 8.3 Customizing Git - Git Hooks

https://travis-ci.com/
Forfatter: Travis-CI

https://mochajs.org/
Forfatter: MochaJS

https://semaphoreci.com/community/tutorials/getting-started-with-node-js-and-mocha
Forfatter: Igor Šarčević - Overskrift: Getting Started with Node.js and Mocha

https://docs.mongodb.org/manual/reference/method/db.eval/
Forfatter: MongoDB - Overskrift: db.eval()

http://stackoverflow.com/a/14797071
Forfatter: user568109 - Overskrift: How the single threaded non blocking IO model works in Node.js - Dato: 10-02-2013

https://docs.mongodb.org/manual/reference/database-references/#dbref-explanation
Forfatter: MongoDB - Overskrift: DBRefs

http://www.haproxy.org/
Forfatter: HAProxy

http://www.haproxy.org/#docs
Forfatter: HAProxy - Overskrift: Documentation

https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts
Forfatter: Mitchell Anicas - Overskrift: An Introduction to HAProxy and Load Balancing Concepts - Dato: 13-05-2014

https://www.digitalocean.com/community/tutorials/how-to-use-haproxy-to-set-up-http-load-balancing-on-an-ubuntu-vps
Forfatter: Jesin A - Overskrift: How To Use HAProxy to Set Up HTTP Load Balancing on an Ubuntu VPS - Dato: 26-09-2013