

Fog Carports

Navn	Studie-mail	GitHub
Lasse Timm Hansen	cph-lh546@cphbusiness.dk	Maximitas
Johan Nikolaj Poulsen	cph-jp471@cphbusiness.dk	ash2213
Asim Kadir Kilic	cph-ak576@cphbusiness.dk	AerrowV
Zana Baran	cph-zb40@cphbusiness.dk	ShadowB0X

Projektinformation

Klasse: A: Gruppe G

Projektfase: 12. nov 2024 – 19. dec 2024

Udarbejdelse af Rapport: 9. dec 2024 – 18. dec 2024

Links

GitHub Repository: https://github.com/AerrowV/fog_semesterprojekt

Video Demonstration: <https://youtu.be/0XNPuSVgBVk>

Hjemmeside: <https://fog.yumiya.dk/>

Demo Adminbruger:

Email: admin@1

Adgangskode: admin

Indledning	4
Baggrund	4
Virksomheden/Forretningsforståelse	5
Interessentanalyse	5
Risikoanalyse	6
Teknologivalg	6
Krav	7
Martins krav og ønsker	7
Grundlaget for user stories	8
Aktivitetsdiagram	9
As-Is	9
To-Be	10
User stories	11
User Story 1	11
User Story 2	11
User Story 3	11
User Story 4	12
User Story 5	12
User Story 6	12
User Story 7	12
User Story 8	13
User Story 9	13
User Story 10	13
User Story 11	13
Domæne model	14
Første iteration af domænemodel	14
Sidste iteration af domænemodel	15
ERD	16
Første iteration af ERD	16
Ændring af tabel navnet "item" til "material"	17
Tilføjelse af en tabel til specifikke materialer ("material_spec")	17
Opdeling af adresseoplysninger fra user	17
Opdeling af "zip_code" fra "address"	17
Normalformer og designvalg	19
Flere veje til rom (data)	19
Brug af auto genereret nøgler	19
Constraints og fremmednøgler	20
1. "user.address_id", som sikrer, at hver bruger kun kan have en gyldig adresse fra tabellen "address".	20
2. "address.zip_code", som sikrer, at alle adresser er knyttet til et gyldigt postnummer fra tabellen "zip_code".	20
3. "order.carport_id", som sikrer, at alle ordrer refererer til en eksisterende carport i tabellen "carport_spec".	20
Generelle designovervejelser	20
Navigationsdiagram & Mockups	21

Valg af arkitektur	22
Særlige forhold	25
Sessions og request variabler	25
Exceptions	26
Brugerininput validering	26
Login sikkerhed	27
Roller i JDBC	27
Mailsystem	27
Kodeeksempler	28
SVG	28
Html meta refresh	29
Status på implementering	29
Test der fejler	30
Kvalitetssikring	30
1. Brug af annotations (@BeforeAll, @BeforeEach, @AfterEach)	31
2. Test Strukturen	31
3. Integrationstest-specifik struktur	32
4. Modularitet og genbrug	32
5. Fordele ved denne struktur	32
Implementering af Mockito	33
1. Simulering af eksterne afhængigheder	33
2. Forenkling af testen	33
3. Isolering af testobjektet	33
4. Håndtering af scenarier, der er svære at replikere	33
5. Verifikation af interaktioner	33
Konklusion	34
User Acceptance tests	34
Proces	37
Arbejdsprocessen faktuel	37
Idefase	37
Udviklingsfase	37
Rapport fase	38
KanBan Mester	38
Vejledningsmøder	38
Kommunikation i teamet	38
Arbejdsprocessen reflekteret	39
Rytme og Proces	39
GitHub	39
Bilag	41

Indledning

Dette projekt omhandler udviklingen af et byg-selv carport-system, for virksomheden Fog. Formålet med projektet er at skabe en fullstack-applikation, der inkluderer en database, backend, front end og deployment til en droplet server. Fog står med udfordringer i deres nuværende system, der gør det besværligt for medarbejdere at håndtere specialdesignede carporte. Vores løsning sørger for at optimere og modernisere deres proces ved at levere et brugervenligt program der samler funktionerne i en samlet platform. Gennem dette projekt har vi haft fokus på at bygge et robust system, der løser de nuværende problemer.

Baggrund

Fog er en dansk virksomhed med 9 lokationer fordelt på Sjælland – i Hørsholm, Fredensborg, Kvistgård, Helsingør, Lyngby, Ølstykke, Herlev, Farum og Vordingborg. I Fog er de eksperter i byggematerialer og giver kvalificeret rådgivning. På Fogs hjemmeside kan man købe byggematerialer, værktøjer, bolig og design, el og belysning og meget mere. De er meget engageret i at give den bedste kundeoplevelse og være med hele vejen, om det så er at købe en lampe eller skulle få opsat en carport, så holder de en i hånden og guider en.

Fog som kunde har nogle mangler bag facaden med deres database og hvordan man kan gøre hele deres system mere enkelt og nemt. Her er fokuset på kunder som bestiller en carport med selvvalgt bredde og længde. Fog ville gerne have et system der automatisk sætter kundens ønsker ind i et program og udregner en stykliste med pris og tegning. Det er også meget vigtigt for dem at styklisten ikke bliver sendt til kunden før der er betalt for produktet.

De ønsker også at kunne være i stand til at ændre på deres database med materialerne i. Såsom at ændre på beskrivelserne på priserne af materialet og hvor meget af de gældende materialer der stadig er på lager. Det er også meget vigtigt for dem at have kontakt til kunden og have en dialog, så kunden opnår den bedste mulige oplevelse og tilfredshed, så et system til at kommunikere med kunden og udregne rabatter og prisændringer er også et ønske.

Virksomheden/Forretningsforståelse

Interessentanalyse

Medvirken på projektet



Indflydelse på projektet

Vores interessentanalyse giver et indblik i, hvilke personer der har interesse i produktet, i hvilken grad, og om de har indflydelse eller ej. Her fremgår det, at salgslederen, som også er den person, der har hyret os til at udvikle programmet, spiller en central rolle i udarbejdelsen af produktet.

Hans ønsker har højeste prioritet, og under normale omstændigheder ville det være ham, vi kommunikerer med. Samtidig spiller udviklerne – altså os selv – en vigtig rolle, da vi både har indsigt i, hvad der er muligt, og er dem, der står for at udarbejde projektet.

Ledelsen/bestyrelsen har også stor indflydelse, men deres direkte involvering er ikke nødvendig for gennemførelsen af produktet, da vi primært kommunikerer med salgslederen.

Vi kan konkludere ud fra denne interessentanalyse at vi ikke har så stor interesse i hvad Fogs kunde ønsker, men mere hvad Fog ønsker fra os, da vi skal forenkle deres arbejdsproces. Vi udarbejder et program, der skal hjælpe Fogs sælgere, som ikke bliver set af Fogs kunder. De har stadig lidt indflydelse, eftersom at en glad kunde hos Fog er en glad kunde hos os.

En vigtig indsigt, vi har opnået, er, hvilken holdning der bør vægtes højest. En enkelt sælgers unikke ønske prioriteres ikke lige så højt som salgslederens ønsker.

Risikoanalyse

	Sandsynlighed	Konsekvenser	Risikograd
Sygdom I gruppen	5	3	15
Fog går konkurs	1	5	5
Martin bliver syg	2	4	8
Dødsfald	1	5	5
Hacket	3	4	12
Martin siger op	1	3	3
Internet er nede	1	4	4
Kommer til skade	4	2	8
Interne konflikter	3	2	6
Hardware problemer	3	5	15
Fravær	3	4	12
Manglende struktur på projektet	4	4	16 *
For stor ambition	2	2	4 *
Dårlig kommunikation	3	4	12
Versions problemer	5	1	5
Github problemer	3	3	9
Github wipeout	1	5	5
Dårlig prioritering	3	4	12
Ujevne bidrag	3	3	9
Server problemer	2	4	8

Vores risikoanalyse har givet os et indblik i hvor der kan forekomme komplikationer for udarbejdelsen af produktet. For eksempel er sygdom i gruppen en kæmpe risiko. Der er en høj sandsynlighed for dette, da vi tidligere har arbejdet i grupper og oplevet udfordringer som sygdom. Konsekvensen vurderes til 3/5, da vi har metoder til at fortsætte arbejdet på projektet, selv hvis udfordringen skulle opstå.

Risikoanalysen har været en stor hjælp til at strukturere projektet bedre, hvilket har givet os et større overblik og gjort os mere produktive med vores tid. Den har også hjulpet os med at forebygge potentielle risici for at minimere deres indvirkning. For eksempel har vi fra starten fået struktureret projektet langt bedre.

Teknologivalg

- JDBC
- PostgreSQL ver. 16.2
- Javalin ver. 6.1.3
- IntelliJ IDEA 2024.2.4 ultimate edition
- Java 17.0.13 Amazon Corretto

- Docker Desktop
- pgAdmin 4
- HTML 5.0
- CSS
- Thymeleaf
- PlantUML
- Mailgun 1.1.3
- DigitalOcean
- GitHub
- Miro
- Mockito 5.4.0
- Figma
- Ubuntu
- Powershell
- Zoom
- Discord

Krav

Under vores gennemgang af videointerviewet med Martin fra Fog blev det tydeligt, at han har specifikke ønsker og krav til forbedringer af deres nuværende system. Disse krav har vi valgt at fokusere på som en del af projektets udviklingsgrundlag.

Martins krav og ønsker

1. **Betalingsafhængig stykliste**

En central del af Martins forretningsmodel er, at styklstens detaljer aldrig må sendes til kunden, før betalingen er modtaget. Styklstens information fungerer som en vejledning til, hvordan man bygger en carport, og hvis kunden modtager denne uden betaling, risikerer Martin, at kunden køber materialerne andre steder og bygger carporten udenom Fog. I stedet skal kunden modtage en ordrebekræftelse, der indeholder detaljer om ordren, såsom længde, bredde og andre relevante oplysninger.

2. **Begrænsede justeringsmuligheder i det eksisterende system**

Martin pointerede, at systemet i sin nuværende form kun tillader prisen på materialer at blive ændret. Beskrivelser, navne, detaljer og lignende kan ikke justeres, og det er heller ikke muligt at tilføje nye materialer eller fjerne eksisterende. Dette begrænser fleksibiliteten og skaber udfordringer, når der er behov for tilpasninger.

3. **Automatisk generering af dynamiske tegninger**

Martin ønskede et system, der automatisk kunne generere en dynamisk tegning baseret på kundens specifikationer. Dette ville ikke kun effektivisere arbejdsgangen, men også gøre det lettere for kunden at visualisere det endelige produkt.

4. Valgmuligheder for bræddebeklædning

Det blev nævnt, at det ville være en fordel, hvis systemet gav kunden mulighed for at vælge mellem forskellige typer bræddebeklædning. Dette ville øge fleksibiliteten og forbedre kundens oplevelse.

5. Integration og arbejdsproces

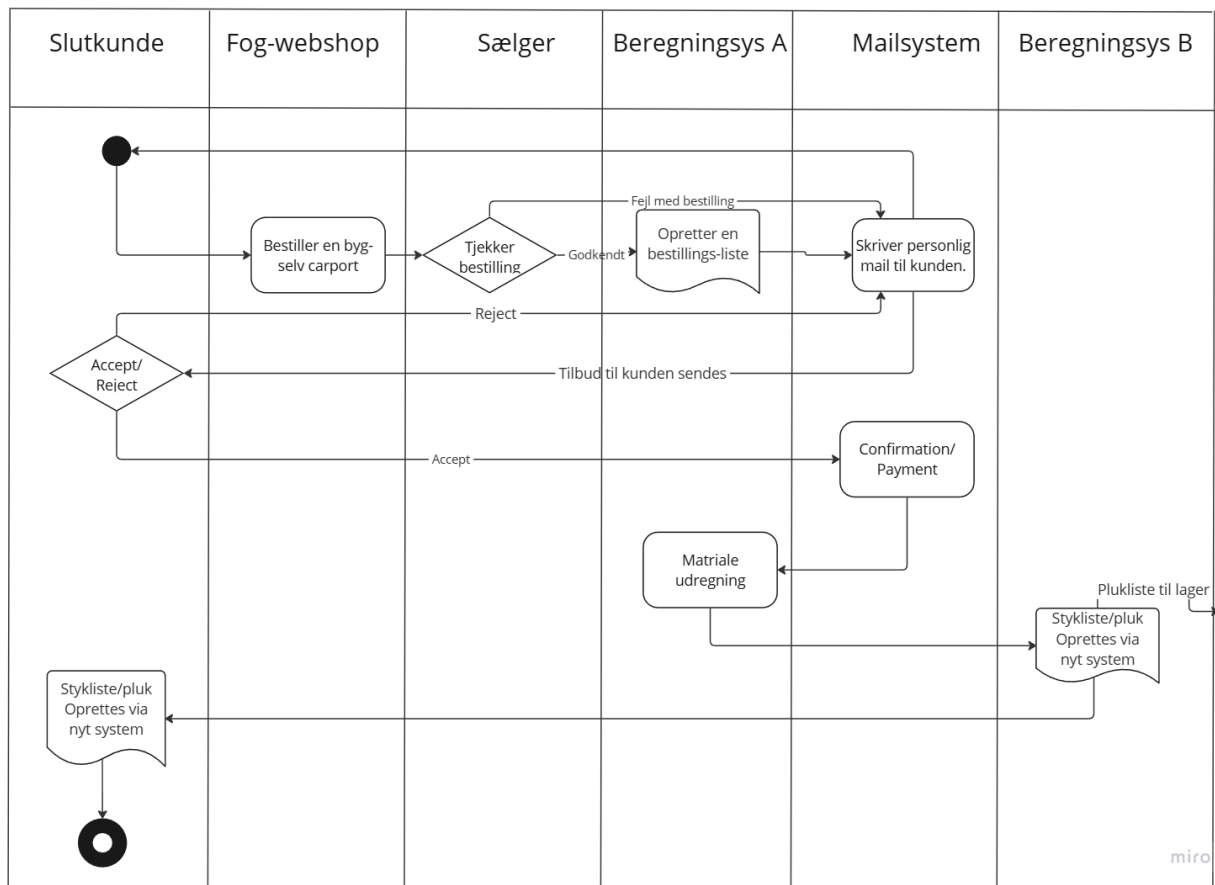
Martin fremhævede en udfordring ved at skulle arbejde i to forskellige programmer for at håndtere kundernes ordrer. Derudover modtog han kundens specifikationer via e-mail og skulle manuelt indtaste dem i systemet, hvilket var tidskrævende og øgede risikoen for fejl.

Grundlaget for user stories

Disse ønsker og udfordringer har dannet grundlaget for udviklingen af konkrete user stories, som vi har brugt til at strukturere og prioritere vores arbejde. Med Martins input har vi kunnet fokusere på at skabe en mere streamline og brugervenlig løsning, der ikke kun opfylder hans krav, men også effektiviserer arbejdsprocesserne hos Fog.

Aktivitetsdiagram

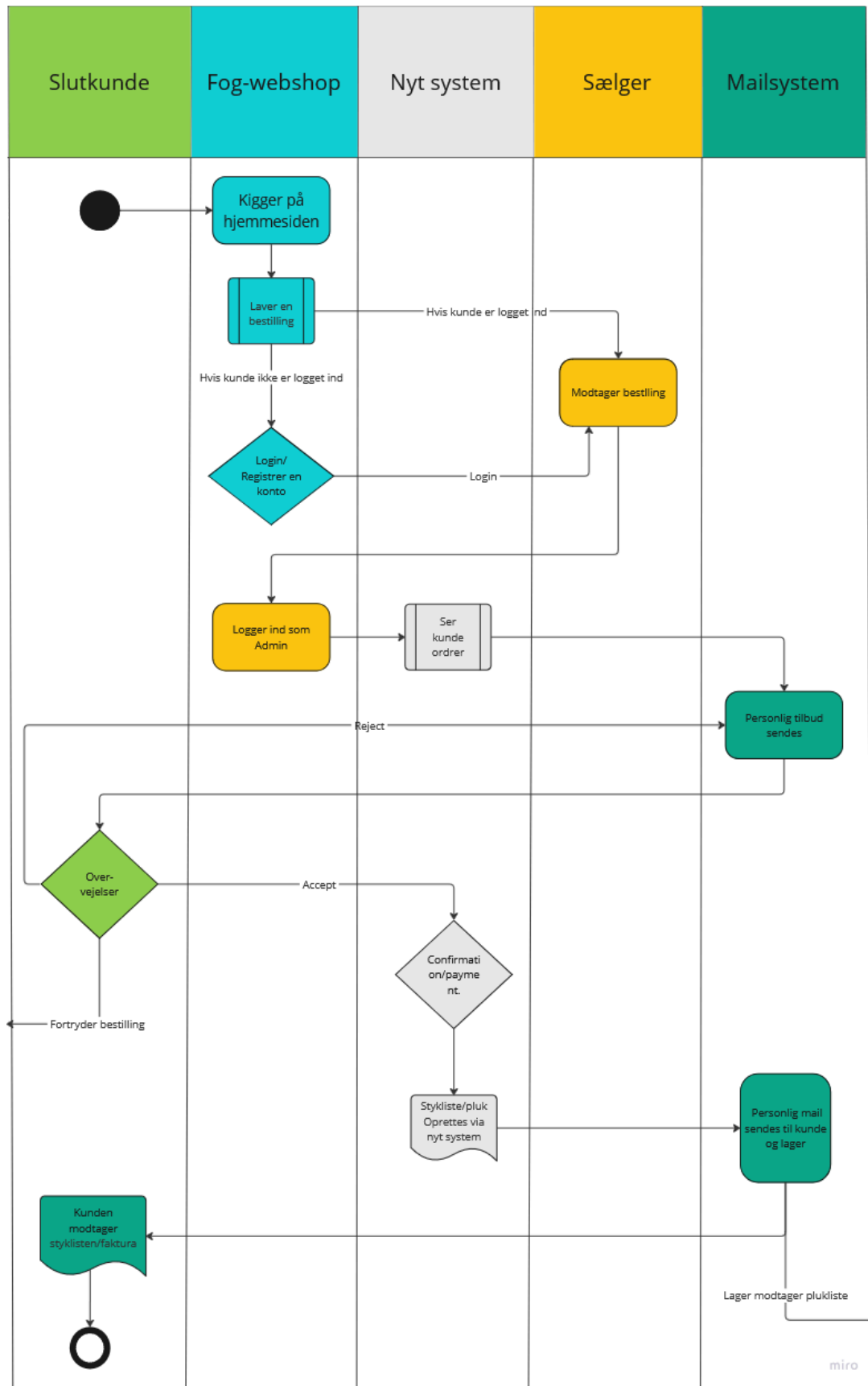
As-Is



Vi har udarbejdet et aktivitetsdiagram, der giver et hurtigt og overskueligt overblik over, hvordan Fogs nuværende hjemmeside og systemer arbejder sammen, samt hvordan aktiviteterne hænger sammen. Diagrammet består af seks svømmebaner, hvor det tydeligt fremgår, at der benyttes to beregningssystemer og et mailsystem. Dette skaber en del frem og tilbage kommunikation mellem de forskellige programmer, før kunden kan gennemføre et køb, og materialerne kan sendes.

Processen starter med, at kunden besøger hjemmesiden og bestiller en carport. Herefter tager en sælger bestillingen og indtaster den i sit beregningssystem. Sælgeren benytter derefter e-mail til at kommunikere med kunden om pris og detaljer. Når betalingen for carporten er modtaget, skal sælgeren skifte til endnu et system for at udarbejde en stykliste til kunden og en plukliste til lageret.

To-Be



Med vores to-be-aktivitetsdiagram har vi udtænkt metoder til at gøre hele processen nemmere og minimere behovet for at hoppe mellem forskellige programmer. Vi har kombineret de to tidligere svømmebaner fra beregningssystem A og B og sammensat dem til en samlet svømmebane, som vi kalder "Nyt System". Planen var, at hele processen skulle foregå i det nye system, så man undgår at skifte mellem programmer.

Aktivitetsdiagrammet To-Be blev udarbejdet, før vi begyndte at kode, og derfor er der nogle forskelle mellem diagrammet og det endelige produkt. I vores program bliver de dog oprettet med det samme, så sælgeren kan få et overblik over omkostningerne og bedre kunne præsentere det for kunden. Styklisten bliver stadig sendt til kunden, når betalingen er modtaget.

Diagrammet viser desuden, at man kan vælge specifikationer for carporten og derefter bliver sendt til login, hvis man ikke allerede er logget ind. Vi har justeret dette i vores program, så brugeren bliver bedt om at logge ind, inden de kan tilgå fanen for valg af carport-specifikationer. Dette forhindrer, at brugeren bruger tid på at vælge specifikationer og overvejelser, for så at blive afbrudt af et log-in midt i processen. På denne måde sikrer vi en mere smidig brugeroplevelse.

User stories

User Story 1:

Som admin vil jeg gerne kunne modtage kundens ordre beskrivelse direkte i beregningsprogrammet, så jeg undgår at skulle indskrive alle data manuelt.

Acceptance Criteria:

Givet en kunde har afgivet en ordre på fogs custom carport page,
Når admin modtager ordren,
Så lægges dataen direkte over i beregningsprogrammet.

User Story 2:

Som admin kan jeg sende beskeder til kunden, så kunden kan vurdere prisen og ordren.

Acceptance criteria:

Givet en kunde har afgivet en ordre,
Når admin logger ind på administrationssiden,
Så skal han kunne sende en personlig mail til kunden.

User Story 3:

Som admin vil jeg have et system der beregner pris på materialer til den carport kunden har valgt, så jeg nemt og præcist kan give kunden en prisoversigt.

Acceptance criteria:

Givet at kunden har lavet en bestilling

Når bestillingen er modtaget i systemet.

Så kan admin se den beregnede pris for kundens bestilling inklusiv fortjeneste.

User Story 4:

Som admin vil jeg gerne kunne tilføje og ændre i databasen, så at databasen kan opdateres løbende med nye materialer/priser.

Acceptance criteria:

Givet at der kommer nye materialer til brug på carporte, eller ændringer af nuværende.

Når admin logger ind på siden,

Så skal han kunne ændre/opdatere med nye materialer/priser.

User Story 5:

Som admin skal styklisten sendes til kunden efter der er betalt, så det sikres at Fogs custom carport concept ikke invalideres.

Acceptance criteria:

Givet en admin har sendt et personligt tilbud til kunden,

Når kunden har accepteret tilbuddet,

Så sendes styklisten til kunden.

User Story 6:

Som admin vil jeg gerne kunne tjekke og godkende en kundes bestilling, så jeg sikrer, at ordren er korrekt og kan sendes videre til lageret.

Acceptance criteria:

Givet en kunde har afgivet en ordre,

Når admin ser ordren i systemet,

Så skal de kunne godkende eller afvise ordren baseret på korrekthed og tilgængelighed af materialer.

User Story 7:

Som admin kan jeg bruge de data som kunden har angivet på sin bestilling fra Fogs custom carport page i et beregningssystem, så kunden kan få et tilbud på en ordre.

Acceptance criteria:

Givet at kunden har lavet en bestilling

Når admin kigger på bestillingen og beregner ordren.

Så kan der gives rabat på ordren.

User Story 8:

Som bruger vil jeg gerne kunne bestille en tilpasset carport via Fogs custom carport hjemmeside, så jeg kan få en carport, der passer til mine behov.

Acceptance Criteria:

Givet jeg er på Fog's custom carport page,
Når jeg indtaster oplysningerne til en tilpasset carport og tilføjer den til min kurv,
Så skal jeg kunne gennemføre bestillingen og få en bekræftelse på min ordre.

User Story 9:

Som admin vil jeg gerne have at lagerstatus for materialer opdateres automatisk, så jeg kan sikre, at kunden ikke bestiller noget, der ikke er på lager.

Acceptance criteria:

Givet et materiale er udsolgt
Når kunden forsøger at bestille det,
Så skal kunden få en besked om, at varen ikke er tilgængelig.

Givet en kunde har afgivet en ordre,
Når ordren indeholder et materiale, der er begrænset på lager,
Så skal lagerstatus automatisk reduceres med de bestilte mængder

User Story 10:

Som admin skal pluklisten sendes til lageret efter en kunde har betalt for en custom carport, så lageret kan udplukke kundens ordrer.

Acceptance criteria:

Givet en admin har sendt et personligt tilbud til kunden,
Når kunden har accepteret tilbuddet,
Så sendes pluklisten til lageret.

User Story 11:

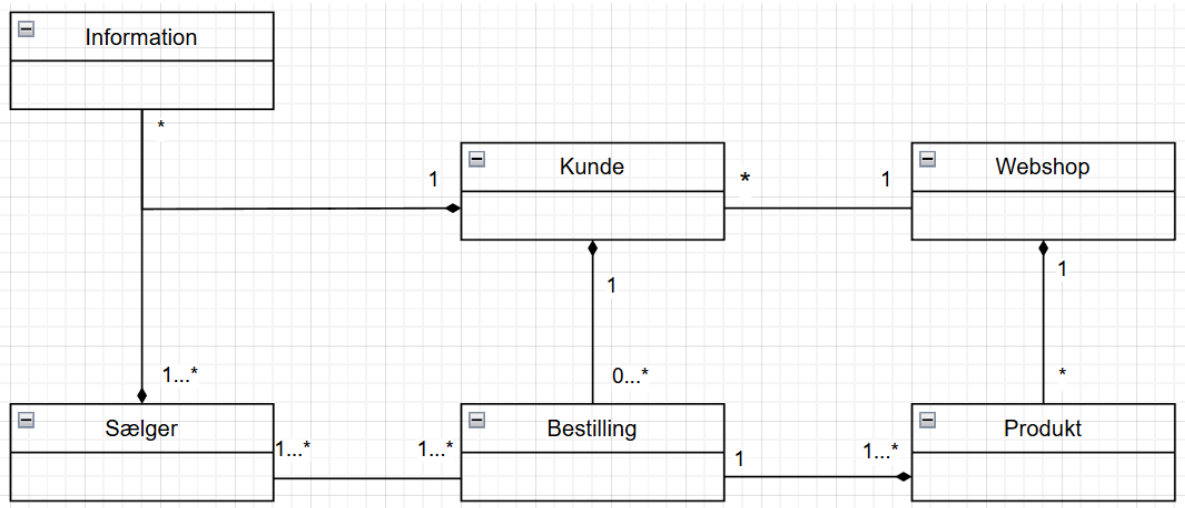
Som kunde vil jeg gerne se en tegning af den custom carport jeg har valgt, så jeg får et overblik over hvordan den ser ud.

Acceptance criteria:

Givet at kunden vælger en custom carport
Når at en kunde har færdiggjort sit valg
Så bliver der vist en tegning af carporten

Domæne model

Første iteration af domænenemodel



Dette var vores første udkast til modellen, som var designet til at understøtte en webshop-platform, hvor kunder kunne bestille produkter og kommunikere med sælgere. Modellen fokuserer på de centrale funktioner i webshoppen og består af følgende klasser:

Kunde: repræsenterer de kunder, der bruger webshoppen til at handle og kommunikere med sælgere.

Sælger: repræsenterer de sælgere, der tilbyder produkter og kommunikerer med kunderne.

Bestilling: repræsenterer kundernes ordrer. En bestilling er knyttet til en kunde og kan indeholde flere produkter.

Produkt: repræsenterer de varer, der sælges på webshoppen. Et produkt kan være en del af flere bestillinger.

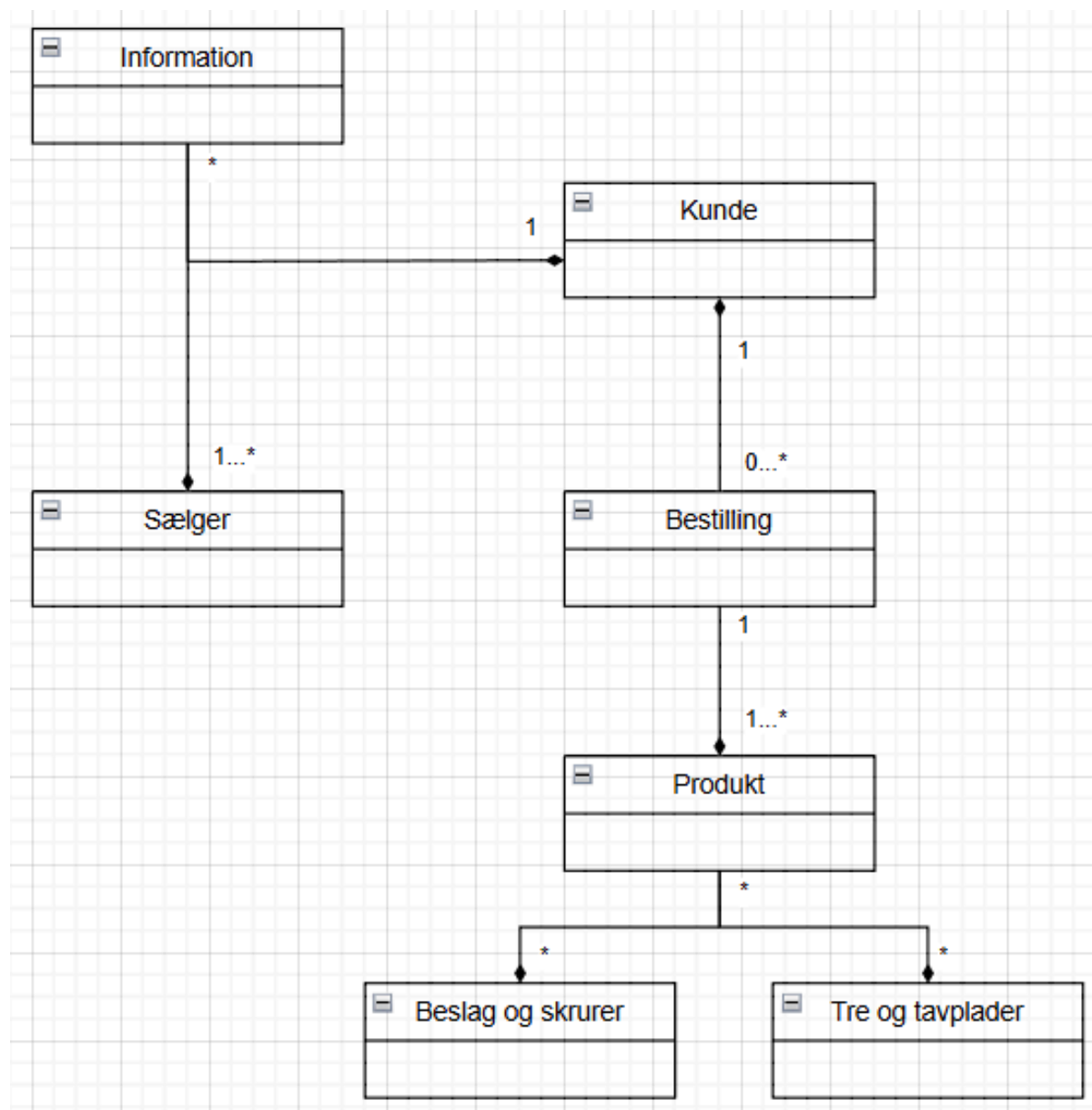
Information: repræsenterer kommunikation mellem kunder og sælgere, f.eks. beskeder eller forespørgsler.

Vores overvejelser for domænenemodellen var lige til. Information bruges til at facilitere kommunikationen mellem kunder og sælgere som nævnt tidligere. For eksempel kan en kunde stille spørgsmål om et produkt eller en sælger kan besvare en forespørgsel. Relationen mellem kunde og sælger er mange-til-mange, da en kunde kan kommunikere med flere sælgere og sælgere kan have kontakt med flere kunder. Vi skabte en tabel, der forbinder både kunder og sælger. Navnet **information** kan man dog føle er lidt tvetydigt, vi fik at vide af en vejleder at det ville være en normal måde at beskrive klassen på.

Derimod tænkte vi også at en kunde kunne have flere bestillinger, men hver bestilling er knyttet til en kunde, hvilket gør det muligt at holde styr på hvem der har afgivet hvilke ordrer.

En bestilling kan også bestå af flere produkter. Vi havde også tilføjet en webshop klasse, men det viste sig ikke at være det rette valg. Dette kommer vi ind på med den sidste version af domænemodellen.

Sidste iteration af domænemodel



I den sidste version af domænemodellen er webshoppen ikke repræsenteret som en separat klasse, fordi hele modellen og dens relationer allerede repræsenterer websystemet. Dette valg er bevidst baseret på at webshoppen fungerer som et

organisatorisk og teknologisk fundament for alle andre klasser i modellen. Vi havde overset, at det vi designede domænemodellen ud fra var vores webshop system.

En separat webshop-klasse ville være overflødig, da den blot ville fungere som en kontekst, der allerede er implicit i de eksisterende relationer. For eksempel er det implicit at alle kunder og sælgere tilhører webshoppen, og at alle produkter og bestillinger administreres af webshoppen.

Dernæst har vi også tilføjet to underklasser af beslag og skruer, og tre og tagplader. Opdelingen giver en klarere struktur og gør det muligt at skelne mellem forskellige typer produkter. Dette understøtter forretningslogikken og gør det lettere at håndtere forskellige produktkategorier i databasen og systemet. Det vil så betyde, at der er mere fleksibilitet og skalerbarhed i systemet.

ERD

Første iteration af ERD



Vores første udkast til databasen indeholdt nogle udfordringer som vi har arbejdet på at løse for at forbedre dens struktur og funktionalitet.

Ændring af tabel navnet “item” til “material”

I vores første design brugte vi tabellen “item” til at repræsentere materialer. Efter nærmere overvejelse besluttede vi at omdøbe tabellen til “material”, da dette bedre beskriver dens formål og gør navnet mere intuitivt.

Tilføjelse af en tabel til specifikke materialer (“material_spec”)

En vigtig opdagelse under designprocessen var behovet for at samle de materialer, der er knyttet til en specifik carport. Dette førte til oprettelsen af tabellen “material_spec”, som fungerer som et mellemlid mellem “material” og “carport_spec”. Denne tabel gør det muligt at tildele materialer til en specifik carport og angive mængden af hvert “material”, der skal bruges. Uden denne tabel ville det være vanskeligt at håndtere relationen mellem lageret og de specifikke materialer, der anvendes til en given carport. Tilføjelsen af “material_spec” var essentiel for at sikre, at databasen kunne understøtte programmets kernefunktion. Altså skabelsen af byg-selv carporte og opfylde 3. normalform.

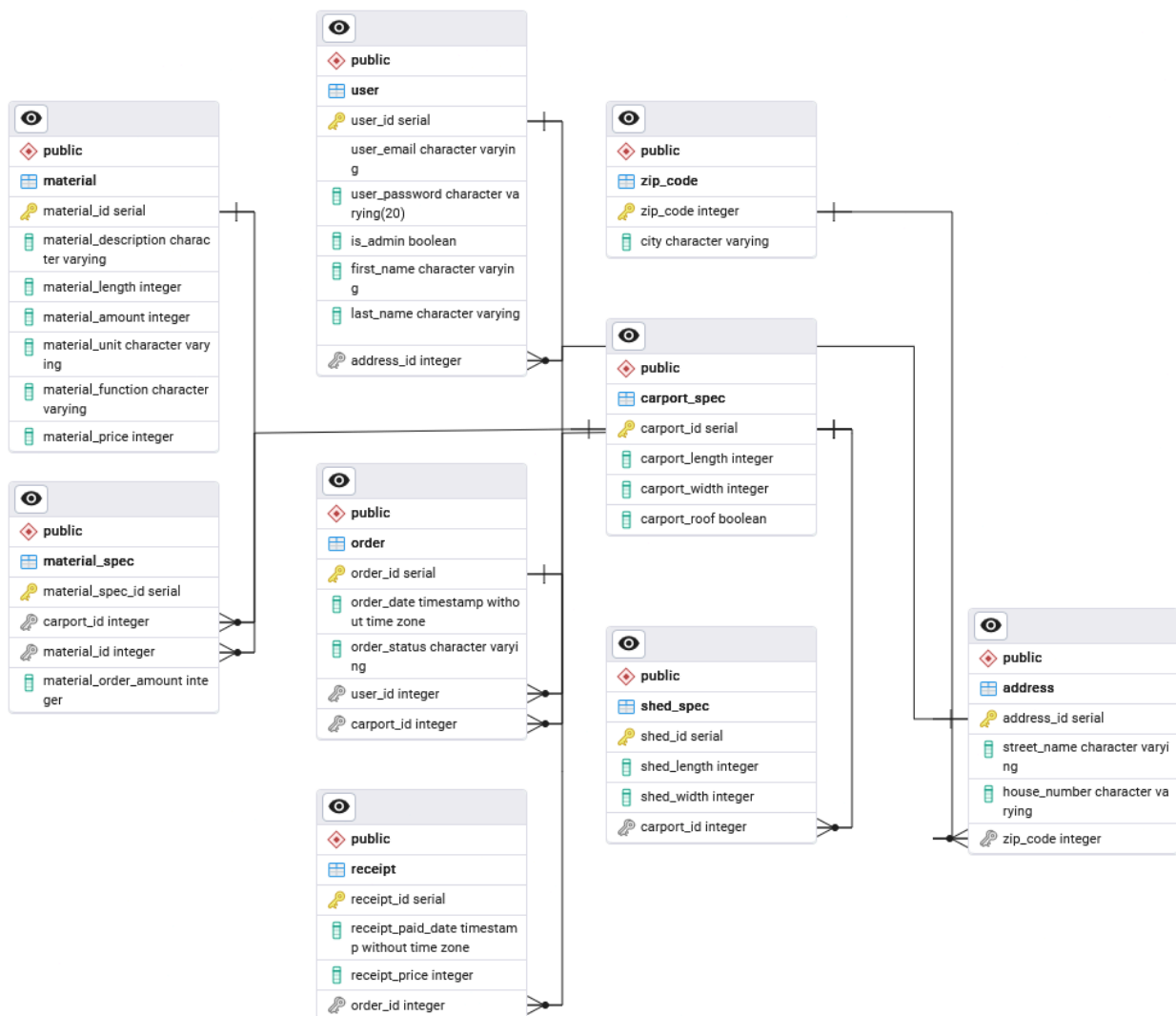
Opdeling af adresseoplysninger fra user

I det oprindelige design blev adresseoplysningerne gemt direkte i user-tabellen. For at undgå redundans besluttede vi at opdele disse oplysninger i en separat tabel kaldet “address”. Denne tabel indeholder felterne “street_name”, “house_number” og en reference til tabellen “zip_code”, som gemmer postnumre og bynavne. Denne opdeling gør det muligt at genbruge adresser på tværs af flere brugere, hvilket fjerner redundans og sikrer bedre datahåndtering.

Opdeling af “zip_code” fra “address”

For yderligere at fjerne redundans blev postnumre og bynavne flyttet til en separat tabel kaldet “zip_code”. Denne tabel sikrer, at hver kombination af postnummer og by kun optræder én gang i databasen, hvilket gør det lettere at opdatere eller vedligeholde disse oplysninger, hvis der sker ændringer. Dette var også for at opretholde 3. normalform.

Sidste iteration af ERD



Databasen er designet til at håndtere data for en specialbygget carport:

“user”: Indeholder oplysninger om brugere af systemet, herunder e-mail, adgangskode og navn. Hver bruger er koblet til en adresse gennem “address_id”.

“address”: Indeholder adresser, vejnavn, husnummer og et tilhørende postnummer (zip_code). Flere brugere kan have samme adresse.

“zip_code”: Indeholder postnumre og tilhørende bynavn. Denne tabel sikrer, at adresser kun kan bruge gyldige postnumre.

“carport_spec”: Indeholder specifikationerne for de speciallavede carporte, som f.eks. længde, bredde og om carporten skal have et tag. Denne tabel definerer de valg, en kunde har taget for deres carport.

“material”: Fungerer som lageret, hvor vi opbevarer oplysninger om alle tilgængelige materialer, herunder beskrivelser, længde, pris, mængde, enhed og funktion.

“material_spec”: Indeholder de specifikke materialer, der er blevet udvalgt fra

lageret ("material") til en bestemt carport ("carport_spec"). Denne tabel specificerer også mængden af materialer, der skal bruges til en given carport.

"shed_spec": Indeholder specifikationer for eventuelle skure, der er tilknyttet en carport. Skure er altid koblet til en specifik carport via "carport_id".

"order": Registrerer kundeordrer, herunder dato, status og hvilke bruger- og carport-specifikationer ordren er knyttet til.

"receipt": Indeholder oplysninger om kvitteringer for betalinger, herunder beløb og betalingsdato. Hver kvittering er koblet til en specifik ordre.

I forbindelse med designet af vores database har vi gjort en række overvejelser for at sikre et robust, fleksibelt og konsistent datamodel.

Normalformer og designvalg

Alle tabeller i databasen er designet til at være på 3. normalform. Dette betyder, at:

1. Hver tabel har en entydig primærnøgle
2. Alle attributter afhænger kun af primærnøglen, og der er ingen transitive afhængigheder.
3. Redundans er elimineret, så data ikke gentages unødigt.

Et eksempel er tabellen "user", hvor attributterne "user_email", "first_name" og "last_name" kun afhænger af "user_id". På samme måde er tabellen "address" designet sådan, at "street_name" og "house_number" kun afhænger af "address_id".

Flere veje til rom (data)

Der er flere veje til at tilgå bestemte data i databasen. F.eks. kan en brugers adresse findes via "user.address_id", som refererer til en postadresse i tabellen "address", der igen er knyttet til tabellen "zip_code". På denne måde kan man både finde detaljer om brugers adresse og bynavn fra forskellige perspektiver. Dette designvalg giver fleksibilitet, men kræver også, at relationen mellem tabellerne opretholdes nøje. Vi har valgt denne løsning for at lette dataudtræk og sikre, at hver tabel indeholder sin specifikke type data uden redundans.

Brug af auto genereret nøgler

De fleste tabeller anvender automatisk genererede primærnøgler (serial) for at sikre entydige identifikatorer. En undtagelse er tabellen "zip_code", som bruger "zip_code" som en naturlig primærnøgle. Dette er valgt, fordi postnumre allerede er unikke og ofte bruges som identifikatorer i virkeligheden. Det hjælper med at gøre databasen enklere og eliminere unødigt gentagelse af data.

Constraints og fremmednøgler

Fremmednøgler bruges gennem hele databasen for at opretholde referentiel integritet. Eksempler inkluderer:

1. **“user.address_id”**, som sikrer, at hver bruger kun kan have en gyldig adresse fra tabellen **“address”**.
2. **“address.zip_code”**, som sikrer, at alle adresser er knyttet til et gyldigt postnummer fra tabellen **“zip_code”**.
3. **“order.carport_id”**, som sikrer, at alle ordrer refererer til en eksisterende carport i tabellen **“carport_spec”**.

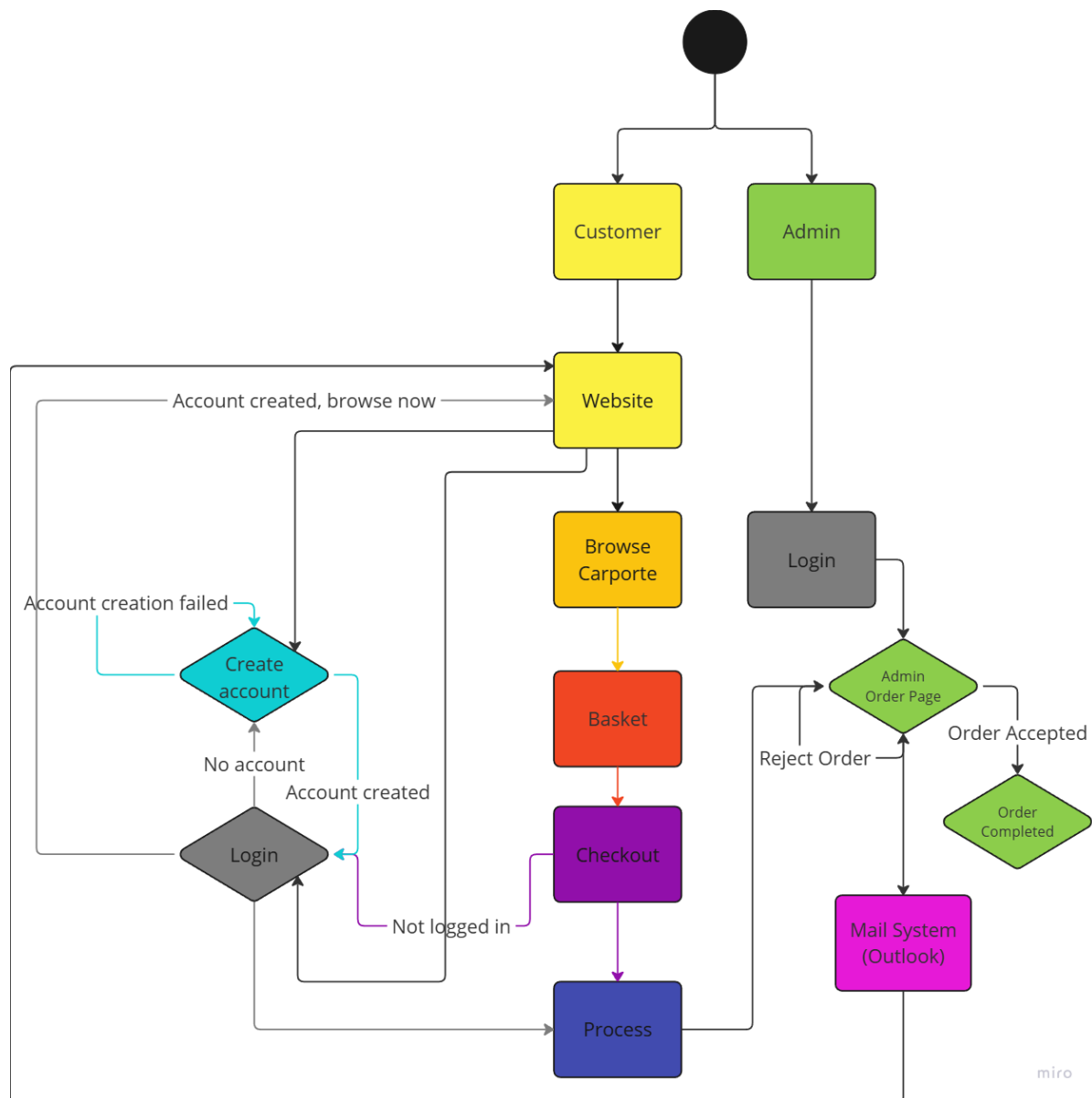
Vi har desuden valgt at implementere CASCADE på visse fremmednøgler, som f.eks. mellem **“order”** og **“receipt”**. Dette sikrer at tilknyttede kvitteringer automatisk bliver slettet, hvis en ordre slettes.

Generelle designovervejelser

Redskabsrum og carport: Tabellen **“shed_spec”** er knyttet til **“carport_spec”**, da et skur altid er knyttet til en bestemt carport. Dette sikrer at der ikke kan oprettes redskabsrum uden en tilhørende carport.

Unikke værdier: **“user_email”** er markeret som unik for at sikre, at der ikke kan oprettes to brugere med dens samme e-mailadresse, hvilket er vigtigt for autenticitet og login-funktionalitet.

Navigationsdiagram & Mockups



I vores navigationsdiagram har vi sat det op så man nemt kan se hvad man kan gøre på vores hjemmeside. Her kan man se at der er to veje fra starten af, en for admin og en for customer (kunde). Det er ikke faner på hjemmesiden, men personer. Dette er for at vise at, når admin logger ind, kommer de til admin siden, og når customer logger ind, navigerer de til forsiden. Både kunder og admin starter på forsiden, men ændringerne sker på login.

For eksempel når der bliver logget ind, tjekker vi ved hjælp af en boolean. Hvis boolean "isAdmin" er true så bliver du redirected til /admin, ellers er det til "/" som er index, altså "website".

Når en admin er logget ind, kan de ikke komme ind på "website" (index). Dette er gjort, da det er en arbejdskonto og vi ønsker adskillelse. En arbejdskonto skal ikke kunne bestille en carport.

I vores endelige kode er der ikke nogen "basket" (kurv). På vores hjemmeside kan man kun bestille en byg-selv carport, så der er ikke brug for en kurv.

"Browse carporte" er blevet ændret til "choose-carport". Vi har gjort dette, eftersom at man ikke browser i færdigbygget carporte, men at man selv vælger specifikationer til en custom carport.

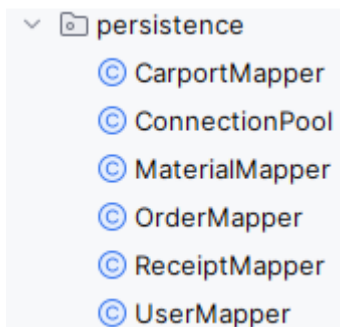
Når man trykker på quick-byg, altså "checkout", tjekker den, om du er logget ind eller ej. Hvis du ikke er logget ind, kommer du til login siden.

Admin har en admin order page hvor de kan se alle de forskellige ordrer af kunderne og kan så interagere med dem.

Valg af arkitektur

Arkitekturen i dette projekt er MVC, Model View Controller. Dette er en arkitektur, der har gjort det mere overskueligt at udarbejde koden for os.

Første del af MVC er **model**, model er alle vores "Mappers", der kommunikerer med vores database.



Her er vores persistence package. Det er nemt og overskueligt at finde frem til hvad og hvor der kommunikeres i databasen. Alle metoder i denne persistence package bliver kun brugt til at kommunikere med databasen via SQL.

```
public static int getCarportId(double length, double width, boolean
hasRoof, ConnectionPool connectionPool) throws DatabaseException {
    String sql = "SELECT carport_id FROM carport_spec WHERE
carport_length = ? AND carport_width = ? AND carport_roof = ?";
    int carportId = 0;

    try (Connection connection = connectionPool.getConnection();
        PreparedStatement ps = connection.prepareStatement(sql)) {
        ps.setDouble(1, length);
        ps.setDouble(2, width);
```

```

        ps.setBoolean(3, hasRoof);

        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                carportId = rs.getInt("carport_id");
            } else {
                throw new SQLException("No carport found with the given
specifications.");
            }
        }
    } catch (SQLException e) {
        throw new DatabaseException("Database error: " + e.getMessage());
    }
    return carportId;
}

```

I kode eksemplet kan man se en metode inde i "CarportMapperen". Denne hjælpeklasse er til for at få fat på et ID fra et table kaldet "carport_spec". Hvis man ville finde en metode, f.eks. "getCarportId", så ville man først lede efter den korrekte mapper og derefter finde metoden "getCarportId".

V'et i MVC står for **view** og er måden vi præsenterer vores dataer til en bruger.

Alt vores frontend har vi sorteret i en resource package, hvor vi har navngivet dem efter hvad kunden eller admin gør på den fane. Hvis vi hopper ind i "login.html" ved vi at det er her at kunden vil indskrive data for at logge ind på sin konto.

Her kan der ses at vi bruger Thymeleaf til at få fat i kundens data og sammenligner det med den data vi har på databasen. Nedenfor har vi en Thymeleaf form, der spørger efter noget data fra kunden. Når kunden indtaster og trykker log på, sender thymeleaf et post-forespørgsel til /login.

```

<form th:action="@{/login}" method="post">
    <label for="email">Mailadresse</label>
    <input type="email" id="email" name="email" placeholder="Mailadresse"
required>
    <label for="password">Adgangskode</label>
    <input type="password" id="password" name="password"
placeholder="Adgangskode" required>
    <button type="submit" class="login-button">Log på</button>
    <a th:href="@{/register}" class="create-account1">Create Account</a>
    <p class="errorMessage" th:text="${#ctx.message}"></p>
</form>

```

```
app.post("/login", ctx -> UserController.login(ctx, connectionPool));
```

Den bliver samlet op på denne linje i vores main, der så fører den videre til vores controllers.

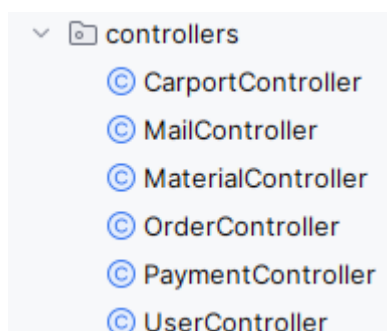
Controllerne er C'et i MVC. Controllers håndtere brugernes input og interaktioner med systemet. Controller er mellemmand for model og view. Vi har en package som hedder controllers med alle vores klasser der tager imod "user" data og behandler det, for så at sende det til "Mappers" som fører det videre til databasen.

```
public static void login(Context ctx, ConnectionPool connectionPool) {
    String email = ctx.formParam("email");
    String password = ctx.formParam("password");

    try {
        User user = UserMapper.login(email, password, connectionPool);
        ctx.sessionAttribute("currentUser", user);
        ctx.sessionAttribute("isLoggedIn", true);

        if (user.getIsAdmin()) {
            ctx.redirect("/admin");
        } else {
            ctx.sessionAttribute("user_id", user.getUserId());
            ctx.redirect("/");
        }
    } catch (DatabaseException e) {
        ctx.attribute("message", e.getMessage());
        ctx.render("login.html");
    }
}
```

I eksemplet har vi at gøre med metoden der får fat i "user's" input fra html formen. Det er vigtigt at nøglen er ens med ID'et på html linjen hvor inputtet bliver tastet. Det kommer ind som en Context og bliver oversat til Strings. Den opretter derefter en "user" hvis e-mail findes i databasen, metoden "login()" i "UserMapper" gør dette ved hjælp af connectionPool.



Her har vi alle vores controllers i en package så det igen er nemt og overskueligt.

MVC har gjort det dejligt nemt at finde rundt i koden og holde det overskueligt med dataen der kommer fra brugeren og skal behandles i databasen.

Særlige forhold

Sessions og request variabler

Vi har valgt at bruge session-variabler, fordi de gør det muligt at gemme brugerens tilstand på tværs af HTTP-forespørgelser. Sessionen bruges f.eks. til at håndtere brugerens "isLoggedIn" og roller:

```
public static void showOrders(Context ctx, ConnectionPool
connectionPool) {
    try {
        Integer userId = ctx.sessionAttribute("user_id");
```

"ctx.sessionAttribute("user_id")" i metoden "showOrders" gemmer brugerens unikke ID efter login, så systemet kan hente de ordrer, der tilhører brugeren.

Dernæst har vi request parametre. I metoden "updateOrderStatus()" er det admin, der ændrer ordrestatus ved hjælp af en dropdown-menu. Dropdown-menuen sender data til databasen via en POST-request, og metoden læser disse data fra formularfelter med "ctx.formParam("order_id")" og "ctx.formParam("new_status")".

```
public static void updateOrderStatus(Context ctx, ConnectionPool
connectionPool) {
    try {
        int orderId = Integer.parseInt(ctx.formParam("order_id"));
        String newStatus = ctx.formParam("new_status");

        OrderMapper.updateOrderStatus(orderId, newStatus,
connectionPool);

        ctx.redirect("/admin/orders");
    } catch (DatabaseException | NumberFormatException e) {
        ctx.attribute("message", "Failed to update order status: " +
e.getMessage());
        ctx.render("admin-order-list.html");
    }
}
```

Denne tilgang gør det muligt for administratoren at ændre status for en specifik ordre, dynamisk og sikkert. For eksempel kan status ændres fra "Approved" eller "Completed," afhængigt af hvad administratoren vælger i dropdown-menuen.

Exceptions

Vi har håndteret exceptions med try-catch blokke for at sikre at systemet ikke bryder sammen ved fejl, og at brugeren får en klar fejlmeddelelse. Det giver samtidig mulighed for at logge eller vise relevante beskeder til brugeren.

I “showOrders()” håndteres fejl såsom en Database Exception, her bliver brugeren informeret om at ordrer ikke kunne indlæses:

```
    } catch (DatabaseException) {  
        ctx.attribute("Message", "Failed to load orders: " +  
e.getMessage())  
        ctx.render("order.html");  
    }
```

Brugerininput validering

Vi har valgt at implementere input validering for at sikre, at kun gyldige data behandles i systemet, dette reducerer risikoen for fejl og mulige sikkerhedsproblemer som SQL injection. Der er flere steder i koden hvor vi benytter input validering men et punkt kunne være i “createUser()”.

I “createUser()” valideres adgangskoder ved at sammenligne “password1” og “password2”. Hvis de ikke matcher, får brugeren en besked.

```
public static void createUser(Context ctx, ConnectionPool  
connectionPool) {  
  
    String email = ctx.formParam("email");  
    String password1 = ctx.formParam("password1");  
    String password2 = ctx.formParam("password2");  
  
    if (password1.equals(password2)) {  
        try {  
            UserMapper.createUser(email, password1, connectionPool);  
            ctx.attribute("message", "Account created");  
            ctx.render("login.html");  
        } catch (DatabaseException e) {  
            ctx.attribute("message", e.getMessage());  
            ctx.render("register.html");  
        }  
    } else {  
        ctx.attribute("message", "Passwords do not match");  
        ctx.render("register.html");  
    }  
}
```

Login sikkerhed

Login og sessionsstyring er implementeret for at beskytte systemet mod uautoriseret adgang. Session-attributten "is_admin" bruges til at adskille almindelige brugere fra administratorer.

I "login()" f.eks. hvis brugeren er admin bliver de dirigeret til "/admin".

```
if (user.getIsAdmin()) {  
    ctx.redirect("/admin");  
}
```

Når en bruger så logger ud i logout(), ændres session-attributter som "currentUser" og isLoggedIn, så systemet ikke længere kan genkende brugeren:

```
public static void logout(Context ctx, ConnectionPool connectionPool) {  
    ctx.sessionAttribute("currentUser", null);  
    ctx.sessionAttribute("isLoggedIn", false);  
    ctx.sessionAttribute("user_id", null);  
  
    ctx.redirect("/");  
}
```

Det sørger for at sikre at kun autoriserede brugere kan tilgå deres data og at administratorer har adgang til de rigtige værktøjer.

Roller i JDBC

Der er defineret to roller i systemet:

Almindelige brugere: Kan oprette ordrer, se deres egne ordrer og bestille en carport.

Admin: Har adgang til at se og opdatere alle ordrer, opdatere priser og tilføje/fjerne/opdatere materialer til databasen.

Vi har valgt denne opdeling for at holde systemet simpelt og let at vedligeholde. Dette sikrer også at brugere kun har adgang til relevante data.

"is_admin" er en boolean vi har tilføjet på database niveau for at definere om man er en køber eller sælger. Man kan kun designere, om man er admin direkte inde i databasen.

Mailsystem

Derimod har vi implementeret et mailsystem i projektet ved at bruge Mailgun. Dette er en tredjeparts e-mail tjeneste, som gør det nemt at sende e-mails på en sikker og

pålidelig måde. Systemet bruges til at sende forskellige typer e-mails, såsom bruger forespørgsler, ordrebekræftelser og admin beskeder.

Vi har specifikt valgt Mailgun af den grund at de understøtter e-mails med html indhold hvilket giver os mulighed for at sende professionelle beskeder.

Vi har så valgt at gemme følsomme oplysninger som API-nøgler, domæne, og afsender email i "Environment Variables" ("System.getenv()"), i stedet for at hardcode dem i koden:

```
private static final String API_KEY = System.getenv("MAILGUN_API_KEY");
private static final String DOMAIN = System.getenv("MAILGUN_DOMAIN");
private static final String SENDER_EMAIL =
System.getenv("MAILGUN_SENDER_EMAIL");
private static final String BASE_URL = "https://api.eu.mailgun.net/";
```

Kodeeksempler

SVG

Scalable Vector Graphics eller SVG for kort, er et værktøj til at tegne billeder i HTML. SVG har vi gjort brug af for at tegne en skitse af kundernes carport. Dette er gjort ved nedenstående klasser og metoder.

Klassen SVG, starter med en standard SVG-ramme, som er defineret med attributter "x", "y", "viewBox" og "width". Dette gør det nemt at specificere basale parametre for SVG-filen.

"addRectangle()", "addLine()", "addArrow()" og "addText()": Disse metoder tilføjer SVG-elementer ved at formatere dataene med skabeloner og sætter dem sammen til stringbuilderen.

```
public void addRectangle(double x, double y, double height, double
width, String style) {
    svg.append(String.format(SVG_RECT_TEMPLATE, x, y, height, width,
style));
}
```

I vores "toString()" sikrer vi os at SVG-strukturen altid afsluttes korrekt, når den konverteres til en streng. Vores ide med dette var at metoden tjekker om SVG-elementet er lukket ved at bruge "isClosed". Hvis det så ikke er lukket, så tilføjer den slutmærket, som er "</svg>" til Stringbuilder-objektet og sætter isClosed til true. Så undgår vi at den generede SVG-streng er syntaktisk korrekt. Det er afgørende for kvaliteten af SVG-grafik.

```

public String toString() {
    if (!isClosed) {
        svg.append("</svg>");
        isClosed = true;
    }
    return svg.toString();
}

```

Meningen med koden er at gøre det nemmere at generere dynamiske SVG-indhold, hvilket er nyttigt i webapplikationer eller værktøjer, som kræver interaktiv grafik. Koden bruges til at bygge grafiske visualiseringer.

Html meta refresh

I metoden “saveUserData()” i “PaymentControlleren” benytter vi os af HTML meta refresh. Dette forklares nedenfor.

```

<meta http-equiv="refresh" content="5;url=/">

```

I denne del af HTML koden bruger vi `http-equiv="refresh"` til at angive at siden skal refreshes. Derefter bruger vi `content="5"` til at indikere, hvor mange sekunder der skal gå og `url=/"` til at fortælle, hvilken side der skal refreshes til. I denne del af koden bliver det index, der refreshes til. Meta refresh er simpelt og fungerer godt i situationer, hvor server-side eller JavaScript-baserede omdirigeringer ikke er nødvendige.

Status på implementering

Vi fik implementeret alle planlagte websider i navigations diagrammet, udover “Basket”. Vi kom frem til, at der ikke skulle lægges noget i databasen før at der blev oprettet en bestilling.

Derefter har vi fået stylet alle vores faner på hjemmesiden. Vi har gjort brug af samme color scheme og skrifttype som vores Figma tegninger for at give brugeren en bedre user experience. Disse tegninger kan ses i bilag.

Vi havde forventninger i starten af projektet om at give kunden muligheden for at få skur med samt at vælge en hældning på sit tag. Derfor eksisterer det i vores database, men der er ikke blevet implementeret i koden.

Det har heller ikke været muligt at lave SVG tegning af sideprofilen for carporten. Dernæst opdagede vi en fejl, hvor SVG tegningen ikke vises på browseren Safari.

Test der fejler

I forbindelse med vores test har vi nogle tests der fejler, dette gælder "createOrder()" og "updatePaidDate()"

"createOrder()" fejler, når man først kører "CarportMapperIntegrationTest" og derefter "createOrder()". Vi har en formodning om, at fejlen opstår, fordi ordren ikke bliver slettet fra databasen efter testen. Hvis "createOrder()" køres alene, uden at "CarportMapperIntegrationTest" er blevet kørt først, opstår fejlen ikke.

"updatePaidDate()" fejler, fordi der anvendes et timestamp med høj præcision, hvor alle decimaler for millisekunder inkluderes. Dette medfører en forskel i de sidste cifre af tidsværdien, hvilket får testen til at fejle, før den reelt kan validere værdien.

Kvalitetssikring

(Integrationstest)

Klassenavn	Metode(r) testet	Beskrivelse	Dækningsgrad
CarportController	saveCustomerSpecifications()	Validerer logikken for oprettelse af carport specifikationer og ordrer, inkl. input validering og omdirigering.	Dækker alle input scenarier: gyldigt, ugyldigt og ikke-godkendt bruger.
UserMapper	createUser(), checkEmail(), login(), getUserByIdWithAddress()	Tester oprettelse af bruger, login og hentning af brugerdata med adresse integration.	Dækker gyldige/ugyldige scenarier for hver metode.
ReceiptMapper	getReceiptByOrderId(), updatePaidDate(), saveReceiptPrice(), getReceiptPaidDate()	Sikrer, at kvitterings håndtering fungerer korrekt, inkl. prisopdateringer og håndtering af betalingsdato.	Dækker gyldige/ugyldige tilfælde for kvitteringer.
OrderMapper	createOrder(), getOrderById(), getAllOrders(), getOrderByUserId(), updateOrderStatus()	Tester oprettelse, hentning og statusopdatering af ordrer.	Dækker gyldige scenarier og fejlscenarier for brugerrelaterede ordrer.

MaterialMapper	getAllMaterials(), getMaterialById(), getMaterialSpecsByCarportId()	Bekræfter hentning af materialer, inkl. materiale specifikationer for en given carport ID.	Dækker gyldige og ugyldige forespørgsler relateret til materialer.
CarportMapper	saveCarportSpecs(), getCarportId(), saveMaterialSpec(), getCarportSpecsById()	Tester lagring af carport specifikationer, ID-hentning og materiale tilknytning.	Dækker gyldige scenarier og kanttilfælde for carportspecifikationer.

Ud fra tabellen ovenfor blev vi enige om i gruppen at teste alle vores "Mappers", samt vores "CarportController", da det er her vores beregningsmetode ligger. Herunder forklares det hvordan vi har benyttet os af tests og hvorfor.

1. Brug af annotations (@BeforeAll, @BeforeEach, @AfterEach)

- @BeforeAll:
 - Bruges til at konfigurere engangs initialisering, såsom oprettelse af tabeller i databasen.
 - Dette sikrer, at databasen er korrekt opsat, inden nogle tests udføres.
 - Køres kun én gang, hvilket reducerer redundans og forbedrer effektiviteten.
- @BeforeEach:
 - Bruges til at indsætte testdata før hver test.
 - Gør hver test uafhængig, da den starter med en kendt og konsistent database-tilstand.
 - Forhindrer, at tidligere tests påvirker resultaterne af senere tests.
- @AfterEach:
 - Bruges til at rydde op i databasen efter hver test.
 - Sikrer, at tests ikke efterlader uønskede data, hvilket ellers kunne påvirke efterfølgende tests.

2. Test Strukturen

- Navngivning af tests:
 - Testmetodenavne beskriver klart, hvad testen verificerer (f.eks. testGetMaterialByIdSuccess).
 - Dette forbedrer læsbarheden og gør det nemt at forstå formålet med hver test.
- Assertions:

- Brug af assertEquals, assertNotNull, og assertFalse sikrer, at testens resultater matcher forventningerne.
- Dette hjælper med at validere forretningslogik og korrekt database interaktion.

3. Integrationstest-specifik struktur

- Direkte database interaktion:
 - Integrationstest verificerer funktionaliteten af mapperne i forbindelse med databasen.
 - Indsættelse og sletning af data i en rigtig testdatabase sikrer, at mapperne fungerer korrekt under realistiske forhold.
- Forberedelse af testdata:
 - Testdata indsættes i testdatabasen (testdb) med kendte værdier i @BeforeEach.
 - Dette muliggør forudsigelige resultater, som testene kan validere mod.

4. Modularitet og genbrug

- Centraliseret opsætning:
 - Brug af en fælles ConnectionPool sikrer, at alle tests bruger samme forbindelse, hvilket reducerer kompleksiteten.
 - Dette gør det nemt at ændre database konfigurationen ét sted, hvis det er nødvendigt.
- Testdata som konstanter:
 - Testdata (f.eks. testMaterialId, testDescription) er defineret som konstanter.
 - Dette gør det nemt at genbruge og opdatere testdata på tværs af tests.

5. Fordele ved denne struktur

- Pålidelighed:
 - Ved at starte med en kendt database-tilstand og rydde op bagefter, sikres det, at testresultater er pålidelige og ikke påvirkes af andre tests.
- Genanvendelighed:
 - Strukturen gør det muligt at tilføje nye tests uden at påvirke eksisterende tests.
 - Opsætningen (oprettelse af tabeller, indsættelse af data) kan nemt genbruges i andre mapper-tests.
- Læsevenlighed:
 - Klare metoder, navne og veldokumenteret opsætning gør det nemt for andre udviklere at forstå og vedligeholde testene.

Udover denne form for struktur har vi også brugt Mockito til integrationstest af vores CarportController af følgende grunde.

Implementering af Mockito

1. Simulering af eksterne afhængigheder

- Mocking af mapper klasser (f.eks. CarportMapper, MaterialMapper):
 - Mapper Klasser interagerer direkte med databasen. Ved at mocke dem kan vi undgå at lave databasekald i testen.
 - Dette sikrer, at testen fokuserer på logikken i CarportController uden at afhænge af en fungerende database.
- Mocking af Context:
 - Ved at mocke Context (ctx) kan vi simulere input (som form-parametre) og verificere, hvordan controlleren reagerer på dem uden at have en webserver og database.

2. Forenkling af testen

- Ved at bruge mocks kan vi kontrollere specifikke scenarier ved at simulere forskellige responser fra mapperne:
 - CarportMapper.saveCarportSpecs(): Returnerer en fast ID-værdi for at simulere succesfuld lagring.
 - OrderMapper.createOrder(): Returnerer en fast ordre-ID.
 - ReceiptMapper.saveReceiptPrice(): Simulerer en void-metode uden at udføre noget reelt.
- Dette forenkler testen ved at gøre den forudbestemt og uafhængig af eksterne faktorer som database- eller netværksproblemer.

3. Isolering af testobjektet

- Formålet med integrationstesten er at verificere CarportControllers funktionalitet.
- Ved at mocke mapper klasserne isoleres controlleren fra resten af systemet.
- Dette sikrer, at testen kun validerer controllerens logik og ikke afhænger af korrekt funktion af mapperne eller databasen.

4. Håndtering af scenarier, der er svære at replikere

- Unikke scenarier:
 - For eksempel kan vi simulere fejltilstande (f.eks. DatabaseException eller ugyldigt input) uden at skulle genskabe dem i en database.

5. Verifikation af interaktioner

- Mockitos verify-metoder:
 - Bruges til at sikre, at metoder på mock-objekter kaldes med de forventede parametre.

For eksempel verificeres det i testen, at:

```
receiptMapperMock.when(() -> ReceiptMapper.saveReceiptPrice(anyInt(),
anyDouble(), any()))
    .thenReturn(invocation -> null);
```

Bliver kaldt præcis én gang. Dette giver ekstra sikkerhed for, at controlleren korrekt bruger Mapper klasserne.

Konklusion

Mockito bruges til at:

1. Isolere CarportController fra eksterne afhængigheder.
2. Simulere forskellige scenarier uden at skulle udføre reelle databasekald eller HTTP-forespørgsler.
3. Forenkle og accelerere testprocessen.
4. Gøre testen mere pålidelig ved at eliminere afhængigheder af eksterne systemer.

Dette sikrer, at testen fokuserer på controllerens logik og ikke påvirkes af implementerings detaljer i "Mapperne" eller fejl i databasen.

I forbindelse med testningen af vores "Mappers" stødte vi på mange problemer med constraints og foreign keys. Da databasen ikke ville have man lagde data ind på specifikke pladser uden at have det på plads. Dette kunne have været løst meget bedre ved at lave alle tests i Mockito og dermed bare simulere en database og være sikker på at ens logik var korrekt opskrevet.

User Acceptance tests

Vores user-stories er blevet testet af en 21-årig testperson med gode IT-færdigheder og erfaring i brugen af digitale værktøjer, hvilket sikrer en pålidelig evaluering af funktionalitet og brugervenlighed.

User-stories	Godkendt	Ikke godkendt
User Story 1: Som admin vil jeg gerne kunne modtage kundens ordre beskrivelse direkte i beregningsprogrammet, så jeg undgår at skulle	Alle data bliver gemt korrekt og ligger klar til sælger med det samme når han kigger på ordren. Det var nemt at finde dataen.	

indskrive alle data manuelt.		
User Story 2: Som admin kan jeg sende beskeder til kunden, så kunden kan vurdere prisen og ordren.		Det var ikke intuitivt at man skulle hoppe ud af fanen med alle ordrer for så at hoppe ind et andet sted og skrive beskeder. Samtidig var der ikke noget sted i programmet til at finde e-mailerne til kunderne. Man kan kun se deres ID og deres ordre dato. Så der er ikke nogen måde at skrive til dem på uden at gå ind i databasen.
User Story 3: Som admin vil jeg have et system der beregner pris på materialer til den carport kunden har valgt, så jeg nemt og præcist kan give kunden en prisoversigt.	Prisen bliver beregnet automatisk og står klar til sælger. Prisen var nem at finde og tilknyttet til de enkelte ordrer.	
User Story 4: Som admin vil jeg gerne kunne tilføje og ændre i databasen, så at databasen kan opdateres løbende med nye materialer/priser.	Det var meget nemt at opdatere databasen da der allerede stod ting og man bare skulle rette dem. Såsom at øge længden fra 360 til 460. Derimod hvis man skulle tilføje et helt nyt produkt, var man lidt lost på hvad 'unit' f.eks var. Siden dette er testet på en person som ikke har styr på denne type materialer og ikke er en Fog salgsperson som programmet er designet	

	til.	
User Story 5: Som admin skal styklisten sendes til kunden efter der er betalt, så det sikres at Fogs custom carport concept ikke invalideres.		Styklisten bliver ikke vist til kunden efter de har betalt og får deres svg tegning. Den bliver dog sendt til deres e-mail, men dette var ikke intuitivt.
User Story 6: Som admin vil jeg gerne kunne tjekke og godkende en kundes bestilling, så jeg sikrer, at ordren er korrekt og kan sendes videre til lageret.	Det var nemt at finde frem til ordene. Han fandt hurtigt den ordre han selv havde lavet, eftersom de stod efter dato.	
User Story 7: Som admin kan jeg bruge de data som kunden har angivet på sin bestilling fra Fogs custom carport page i et beregningssystem, så kunden kan få et tilbud på en ordre.	Det fandt han selv frem til. Han fik selv givet sig en rabat på 15% og opdaterede den så han kunne se det på prisen.	
User Story 8: Som bruger vil jeg gerne kunne bestille en tilpasset carport via Fogs custom carport hjemmeside, så jeg kan få en carport, der passer til mine behov.	Det opnåede han uden problemer.	
User Story 9: Som admin vil jeg gerne have at lagerstatus for materialer opdateres automatisk, så jeg kan sikre, at kunden ikke bestiller noget, der ikke er på lager.	Det gik stærkt at finde ud af hvor materialerne var, det tog lidt tid at finde ud af om der var trukket materialer fra. Det skyldes at de alle lå i bunden efter opdatering, så man skulle rulle helt ned i bunden af siden.	

User Story 10: Som admin skal pluklisten sendes til lageret efter en kunde har betalt for en custom carport, så lageret kan udplukke kundens ordrer.		Dette var ikke til at finde noget sted. Det var ikke til at vide om dette var sket eller ej.
User Story 11: Som kunde vil jeg gerne se en tegning af den custom carport jeg har valgt, så jeg får et overblik over hvordan den ser ud.	Tegningen kom frem med det samme, da der var blevet indtastet betalingsoplysninger og de var godkendt.	

Proces

Arbejdsprocessen faktisk

Idefase

I idefasen startede vi ud med at analysere Fogs interview video. Det var en svær proces at både kunne tyde og formulere nogle konkrete user-stories. Til gengæld blev

vi hjulpet godt på vej i første uge (forretning og forståelse) med Kim Melkane, hvor vi både fik lavet As-Is, To-Be og interessentanalyse. Med disse værktøjer fik vi udviklet de user-stories nævnt i rapporten. Herefter fik vi designet et udkast til vores domæne model samt database.

Desto mere vi kom i dybden med hvordan opgaven skulle udføres, ift. vores diagrammer stod det klart, at vores første udgave af databasen skulle laves om. Som nævnt tidligere i rapporten opnåede vi dette resultat i slutningen af første uge af projektfasen.

Udviklingsfase

I udviklingsfasen begyndte vi at kode. Dette var omtrent 1 uge inde i den officielle projektfase. Vi benyttede os af de user stories vi udarbejdede i idefasen og sikrede os at vi udviklede det relevante ved hjælp af vores user-stories. Vi opdelt arbejdsbyrden i user-stories, så alle havde noget at tage hånd om. Til selve hjemmesiden, så lavede vi HTML testsider til at kontrollere de metoder vi skabte fungerede. Til sidst, da vi var langt i arbejdet med user-stories og havde perioder

med downtime, blev der arbejdet med styling. Denne fase var ikke uden komplikationer, hvilket vi vil uddybe senere i rapporten. Fasen strakte sig omtrent 2 uger.

Rapport fase

I rapport fasen begyndte vi på rapportskrivning. Dette viste sig at være forholdsvis gnidningsfrit, da vi både havde en masse materiale som vi har benyttet os af igennem projektet. Vi havde en masse diagrammer at skrive om og logbog for at huske vores tidligere refleksioner.

Denne fase har heller ikke været uden komplikationer som bliver nævnt nedenfor.

KanBan Mester

Principielt havde vi ikke en KanBan Mester. Når det kommer til KanBan af vores projekt, implementerede vi det først efter en af de senere vejledninger. Det var godt at få med, da det gav et overblik over hvor langt vi var med vores user-stories, og hvor langt user-storiesene var med at blive implementeret.

Vi alle arbejdede ligeledes med KanBan i projektet og valgte hver især en user-story at arbejde med. Så vi benyttede os ikke af en KM til vores projekt. Dog var KanBan med til at gøre vores projekt og ligeledes os selv mere struktureret.

Vejledningssmøder

Inden hver vejledningssmøde havde vi planlagt en liste af spørgsmål ift. vores projekt/rapport som vi var usikre om på det tidspunkt. Hvis der var ekstra tid, fremviste vi det vi havde lavet siden sidst og fik feedback på det. F.eks. til vejledningssmødet 13/12 kl. 10:30 havde vi spørgsmål til, hvor meget af koden vi skulle få dækket med test. Vi kom frem til at vi skulle dække vores "Mappers" og "CarportController" med test.

Kommunikation i teamet

Kommunikationen i teamet har alt i alt været relativt fint, men ikke uden sine udfordringer. I de første par uger af projektforsløbet var vi relativt dygtige til at møde ind på skolen og til en vis grad møde til den aftalte tid. Men vi kunne ikke undgå sygdom, rehabiliteringsproces efter operation og den sædvanlige, at man sov over sig. Dette var mere prevalent i løbet af de sidste par uger af projektforsløbet, og der opstod en vis form for dovenskab i vores gruppe. Det har så betydet at vi arbejdede en stor del af tiden hjemmefra og kommunikerede over applikationen Discord ved brug af beskeder, opkald og skærmdeling.

Arbejdsprocessen reflekteret

I vores udviklingsfase fik vi aldrig rigtig nedbrudt vores user-stories til mindre opgaver og derfor endte vi også med, at der var mange overlappende user-stories. Dette endte med, at en person på et tidspunkt sad og arbejdede med tre user-stories på en gang.

Hvis vi havde lavet test på dem i form af hardcoded objekter/metoder, så man kunne arbejde med dem uden at have data eller lignende fra andre user-stories, så kunne de nemmere deles op.

I vores udviklingsfase betød det at vi blev nødt til at være flere der arbejdede på samme user-story. Da vi først fik implementeret KanBan processen, blev vi mere klar over dette problem, men dette gjorde vi desværre først en uge inde i udviklingsfasen, efter vores vejledningsmøde om fredagen. Med kanban fik vi mere struktur over projektet, men vores arbejdsproces forblev den samme, da vi ikke havde mere tid til at bearbejde problemet.

Fra starten af projektet var vi meget enige om, at en uge var tilstrækkelig tid til idefasen. Dernæst ville udviklingsfasen tage hvad svarer omtrent til to uger, hvor vi til sidst ville sætte hvad svarer til to uger til rapporten.

Vi fik stort set opfyldt de estimeringer vi havde skabt for hele forløbet, men det var ikke helt spot-on. Der var nogle dages forskel ift. vores mål, hvilket skyldtes f.eks. problemer ift. vores ERD under designfasen. Vi kom frem til, at nogle af modellerne måske ikke helt var på plads, og at der var uforudsigelig problemer, som opstod i koden.

Rytme og Proces

Igennem projektet havde vi som sagt de forskellige faser, men vi blev først rigtig produktive, da vi implementerede KanBan og fik et overblik over vores user-stories. Alt i alt var vi meget produktive fra starten af, men produktiviteten faldt, da vi nåede til rapportskrivningen. Dette skyldtes nok at vi alle sammen brænder for at kode, men da vi nåede til rapportskrivningen, var vi brændt ud.

GitHub

Vi oplevede ikke problemer med at dele koden på GitHub, eftersom vi arbejdede på vores egen branch, hvor vi pushed det vi lavede op til en fælles "dev" branch. Vi benyttede os ikke af pull-requests da vi altid havde et overblik over hvad folk lavede og kiggede hvert individs kode igennem på Intellij, inden det blev hentet ned. Derefter blev der så merged ind til "dev" branchen, generelt uden nogen mærkværdige komplikationer.

Vi testede også vores egne versioner af *dev*-branchen, som der blev pushet internt, før den blev delt med resten af teamet. Det havde været mere effektivt og fejlsikkert at pushe ændringerne som et pull request. Dette ville have givet os bedre indsigt i vores arbejde og mulighed for at give feedback, inden ændringerne blev integreret i *dev*-branchen.

Bilag

30 dages returret

Klik og hent inden for 3 timer*

1-3 dages levering*

Lån en trailer gratis

Fog®

Bolig & design

Byggematerialer

El & belysning

Have & fritid

Værktøj

Mærker

Tilbud

Fog Pro

Søgning

Fog Konto

Find Fog


Kurv

Johannesfog.dk

Browse our shop

Snevarslet er her


Bliv vinterklar med det rette udstyr



Fog®

Byg selv-carporte

Fog leverer byg selv-carporte baseret på kvalitetsbyggematerialer og kan tilbyde individuelle løsninger tilpasset dine behov og ønsker. Vi leverer altid til døren med egen kranbil eller i samarbejde med vores leverandører.



Kundeservice

Kontakt Fog

Otte stillede spørgsmål

Købs- og leveringsvilkår online

Fortrydelse og returnering

Reklamation og klage

Frugt

Prismatch

Lån en trailer

Aktuelt

Mærker

Aviser og kataloger

Tilmeld nyhedsbrev

Inspiration & guides

LinkedIn

Instagram

Facebook

Om Fog

Åbningstider

Om Fog

Fogs Fond

Karriere i Fog

Bliv kontaktkunde i Fog

Persondatapolitik

Cookies

Med omtanke

CSR-rapporter

Certificering

Whistleblowerordning

Whistleblowerpolitik

Fog®

Johannes Fog A/S

Firskovvej 20

2800 Lyngby

CVR-nr. 16314439

41



Forretning: Johannes Fog A/S

Beløb: 24998,00 (DKK)

Ordrenummer: 89987

Vælg betalingsmetode og tryk "Næste".

- ☒ Visa
- ☐ Mastercard
- ☐ Diners Club
- ☐ Maestro
- ☐ Dankort
- ☐ Gavekort
- ☐ MobilePay

Ved betaling med internationale betalingskort reserveres købsbeløbet på din konto.

NÆSTE >

AFBRYD