Webhooks

Forfatter: Asmus Skafte Böttcher
Uddannelse: Datamatiker (Computer Science)
Uddannelsesinstitution: Erhvervsakademi Zealand Roskilde
Vejleder: Jakob Nørager Christensen
Projektperiode: 19/10/2022 – 09/01/2023
Antal tegn: 51.132

FORFATTER: ASMUS SKAFTE BÖTTCHER

UDDANNELSE: DATAMATIKER (COMPUTER SCIENCE)

UDDANNELSESINSTITUTION: ERHVERVSAKADEMI ZEALAND ROSKILDE

VEJLEDER: JAKOB NØRAGER CHRISTENSEN

PROJEKTPERIODE: 19/10/2022 – 09/01/2023

Abstract

After my Internship at Monthio I asked if they had a project I could work on for my final semester. The project they found for me was the development of an API that can keep their clients informed whenever there are changes in their system. The way Monthio's current system handles this is by sending out webhooks to their clients when changes happen. This is logic is tied into multiple places in their software. My task was to create a centralized place where webhooks are configured and fired. To do this I had to figure out the answer to these questions:

- How can webhooks be implemented so that they are retried in case of failure?
- How can webhooks automatically be fired when there are changes in the system?
- How can webhook data be stored?
- How can Unit Tests be used to ensure quality?

To answer these questions I used an agile approach to software development. This was done by following Extreme Programming practices.

Forord

Denne rapport handler om hvordan jeg arbejdede med udviklingen af et produkt for Monthio. Meningen med projektet er at lave et centralt sted hvor Monthio's webhooks kan konfigureres og sendes. Dette vil gøre det muligt for Monthio's personale at håndtere ting som langsigtede genforsøg, analyse af sendte webhooks og andre infrastrukturelle ting med bedre resultater. En fordel vil også være at en separat applikation der håndterer webhooks vil være mere fleksibel, eftersom det eneste den indeholder er logik vedrørende webhooks. Dette vil gøre det meget lettere for kunder at konfigurere og have overblik over deres webhooks i enten UI eller API.

Indholdsfortegnelse

1. Indledning

1.1 Virksomhedsbeskrivelse

Monthio blev grundlagt i 2018 og er en fintech virksomhed. De tilbyder et b2b SaaS produkt som virksomheder kan bruge til at gøre vurderingen af deres kunders kreditværdighed lettere, ved bl.a. at lade dem udfylde et flow med information om deres finansielle situation. Og efter udfyldelsen af det flow findes der automatisk ud af hvilke lån de kan tilbyde.

Der er 33 medarbejdere, som hovedsageligt er opdelt i to afdelinger. Den ene afdelings fokus ligger på customer-succes, altså at gøre kundens oplevelse så værdifuld som muligt. Dette indebærer bl.a. arbejde med data analyse, salg af produktet, kundeservice og snakke med kunder om hvilke features de kunne ønske sig for at gøre oplevelsen af produktet bedre. Den anden afdeling består hovedsageligt af software udviklere. De arbejder med udvikling, design og vedligeholdelse af it-programmer og it-systemer. Afdelingen er delt op i tre hold med 3-5 udviklere på hvert hold, et par enkeltstående udviklere, samt en produkt manager og en produkt designer. Stort set alle udviklere er full-stack developers og arbejder med både Backend og Frontend.

1.2 Produkt introduktion

For at få en bedre forståelse for hvad projektet handler om, kræver det noget viden om hvordan selve Monthio's produkt fungerer i praksis.

B2b SaaS står for business-to-business software-as-a-service. Det vil sige at Monthio sælger deres produkt til andre firmaer. Et eksempel på et firma der kan være bruger af produktet kunne være en bank. I det eksempel vil banken havde adgang til Monthio's produkt og implementere det i deres praksis. Hvis en kunde kontakter banken for at optage et lån, vil processen finde sted i Monthio's automerede system. Her opretter en medarbejder i banken en sag, og dermed et link der sendes til kunden. Det her link skal kunden åbne for at få adgang til et flow hvor de kan udfylde information om deres finansielle situation. Medarbejderen er selvfølgelig i gang med at hjælpe flere kunder med denne proces, og har derfor et sted hvor de kan se en liste over alle sager. Til hver sag kan de se en status på om for eksempel kunden har uploadet et dokument, gennemført flowet eller andre ting der kræver at banken skal tage handling. Hvis medarbejderen har en masse kunder på en gang, har de brug for en bedre måde end at skulle scrolle igennem sager og tjekke statusser, for at se hvilke der er gennemført. Til det bruger Monthio webhooks som er en måde at sende automatiske beskeder og information til andre applikationer via en URL. Så vil man for eksempel kunne opsætte en webhook for en sag, hvor webhooken bliver afsendt når en sags status bliver ændret. På den måde kan medarbejdere i banken modtage en besked på slack, en email eller andet, der informerer dem om at sagen er blevet opdateret.

1.3 Problemformulering

Monthio's nuværende system udsender webhooks til deres kunder når der er statusændringer vedrørende deres data. Disse webhooks bliver udsendt fra forskellige steder i deres software. Monthio vil gerne have ét centralt sted hvor webhooks kan konfigureres og sendes. Dette vil gøre det muligt for Monthio's personale at håndtere ting som langsigtede genforsøg, analyse af sendte webhooks og andre infrastrukturelle ting med bedre resultater. En fordel vil også være at en separat applikation der håndterer webhooks vil være mere fleksibel, eftersom det eneste den indeholder er logik vedrørende webhooks. Dette vil gøre det meget lettere for kunder at konfigurere og have overblik over deres webhooks i enten UI eller API. Som systemet er lige nu, hvis Monthio's kunder bruger en webhook til forskellige use cases og gerne vil opdatere en webhook's destinations URL, kan de blive nødt til at ændre konfigurationen flere steder.

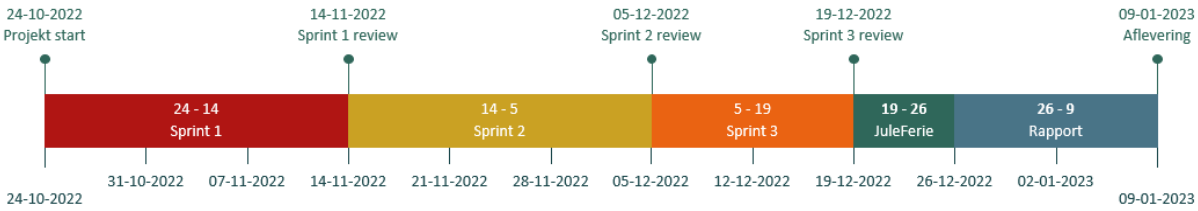
Hvordan kan Monthio bruge Webhooks som en service til at holde deres kunder informeret?

- Hvordan kan afsendelsen af webhooks implementeres så der ikke bliver opgivet med det samme i tilfælde af fejl?
- Hvordan kan webhooks automatisk afsendes når der sker statusændringer?
- Hvilken måde kan webhook data opbevares?
- Hvordan kan Unit Tests bruges til at sikre kvalitet?

2. Metode

Inden jeg gik i gang med projektet fik jeg udleveret en projektbeskrivelse og en liste af acceptance kriterier af Monthio. Herefter har jeg snakket med min kontaktperson hos Monthio, om prioriteten af de forskellige acceptance kriterier og hvilken rækkefølge det kunne være smart at udvikle produktet i. Så begyndte jeg at opdele kriterierne i user stories, som jeg hver gav en prioritet og et story point. Disse user stories delte jeg op i sprints for at gøre arbejdet mere overskueligt, hvilket jeg vil uddybe nærmere under punkt 3.1 'Product Backlog'.

Inden jeg startede, lavede jeg en tidsplan for hvordan jeg ville fordele mit arbejde over hele perioden. I alt havde jeg 11 uger til projektet, som jeg valgte at dele op så jeg ville bruge de første otte uger til hovedsageligt produktudvikling, med en smule opgaveskrivning ved siden af. Efter det havde jeg sat en uge af til juleferie da jeg skulle på skiferie. Derefter med to uger tilbage til deadline, ville jeg stoppe helt med at arbejde på produktet og udelukkende fokuserer på rapportskrivning.



For at kunne svare på spørgsmålene i min problemformulering skal jeg bruge meget tid på at undersøge hvilke metoder der vil være mest passende at bruge til at lavet mit projekt. Det kræver at jeg forholder mig kritisk over for det læsestof jeg finder, og sørger for at vælge metoder der er baseret på forskning og data.

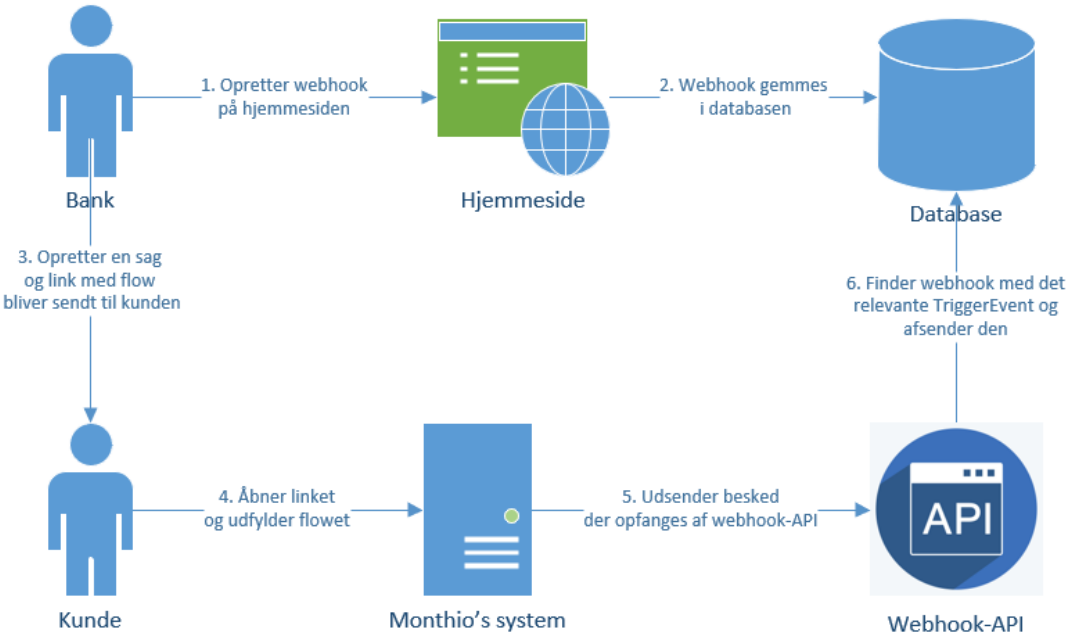
2.1 Inception Deck

For at klargøre kernen i projektet, vil jeg bruge en lille del af Inception Deck til at vise hvad løsningen går ud på.

2.1.1 Show the solution

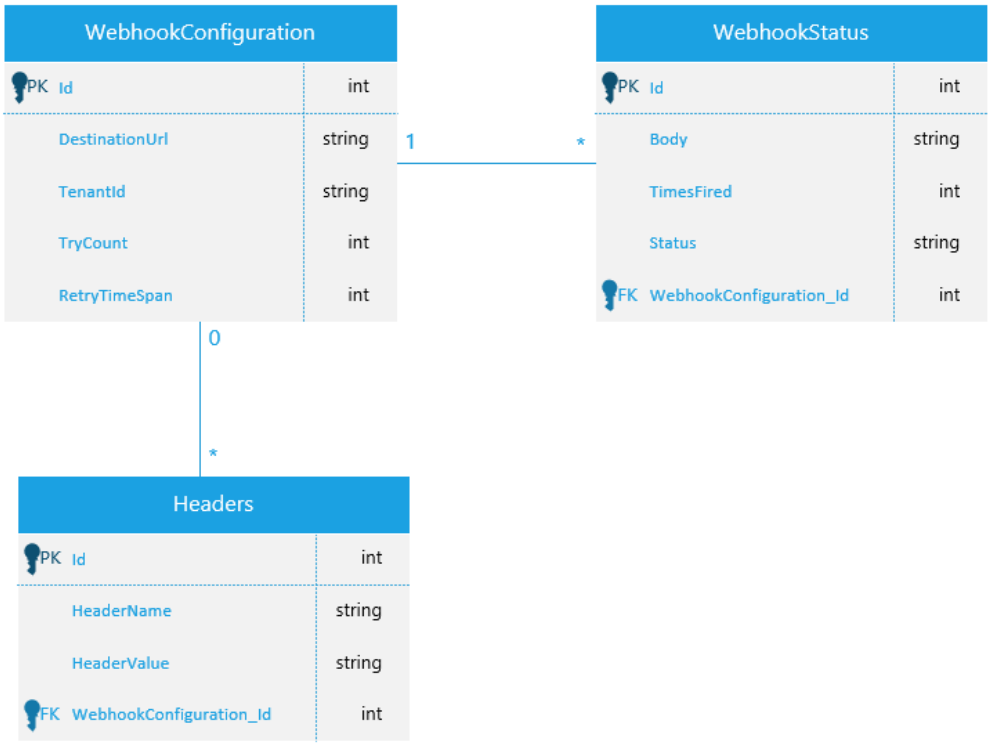
2.1.2 Teknisk arkitektur

For at give et bedre overblik over hvordan processen ser ud, når banken opretter en sag med en tilknyttet webhook, har jeg lavet dette diagram:



1.1.1 ER-Diagram

Som basis for den database jeg skal oprette senere, har jeg lavet et ER-Diagram.



2.2 Extreme Programming (XP)

Jeg vil bruge et udsnit af The New XP metodikkerne under mit arbejde med dette projekt, til at hjælpe mig til at arbejde agilt. Ikke alle XP praktikkerne vil være relevante for mig af forskellige årsager, så jeg vælger at bruge dem der er aktuelle for mit projektforløb.

2.2.1 Stories

Funktionaliteten af systemet skal beskrives med User Stories, som er korte beskrivelser af krav til systemet.

Jeg vil bruge denne praktik, fordi user stories kan hjælpe med at skabe et overblik over det arbejde man har foran sig. Samtidig vil de gøre det lettere at kommunikere funktionaliteten af produktet til andre.

Jeg har gået efter at opfylde de seks INVEST[1] kriterier i mit arbejde med user stories. INVEST omfatter seks koncepter der danner en god user story.

Independant:
User stories skal være så uafhængige af hinanden som muligt. Dette var ikke altid muligt for mig, da jeg har skulle bygge et projekt fra bunden. Derfor har det være nødvendigt for nogle user stories at være gennemført før en anden kunne begyndes. Men der hvor det har været muligt har jeg arbejdet på at lave dem uafhængige af hinanden ved at opdele hver user story i mindre underopgaver på mit Kanban Board.

Negotiable:
User stories må gerne ændres undervejs. Det har jeg været åben for og jeg har ændret dem efter mit eget behov og efter feedback fra min kontaktperson hos Monthio.

Valueable:
Alle user stories skal have værdi. Jeg vil mene at alle mine user stories har værdi da de er udarbejdet direkte ud fra de acceptance kriterier jeg er blevet givet Monthio. Jeg også givet user stories en business value(prioritet), og sørget for at fordele de user stories med højeste prioritet i de tidlige sprints.

Estimable:
For at have en idé om hvor lang tid en user story tager at gennemføre skal man estimere den. Det har jeg valgt at gøre ved at tildele hver user et Story Point. På den måde kan jeg bedre fordele mine user stories i sprints, så mængden af arbejde passer til længden på sprintet.

Small:
User stories skal helst bestå af små stykker arbejde. Det her kriterie opfyldte jeg ved først at sørge for at de var uafhængige af hinanden, hvorefter jeg gik ind og lavede uddybende underopgaver til de user stories hvor jeg syntes det var nødvendigt.

[1] New to agile? INVEST in good user stories – [Agile for All]

Testable:
Hver user story skal kunne testes før den er ”done”. For at gøre det let at teste at mine user stories lever op til kravene sat af acceptance kriterierne, har jeg skrevet acceptance tests i form af Given-When-Then(GWT) til hver af dem. GWT er en måde at sætte et testscenarie op på, hvor man først angiver de krav der er og den situation man skal være i inden man tester. Derefter angiver man hvad der skal gøres og trykkes på for at sætte testen i gang. Til sidst definerer man resultatet der forventes for at testen er succesfuld.

2.2.2 Slack

For at undgå at love noget man ikke kan holde, er det en god idé at inkludere nogle opgaver der kan droppes, hvis man senere finder ud af at tiden ikke er til det.

Jeg har valgt at bruge denne praktik fordi jeg har en deadline på projektet, og jeg skal nå at både kode og skrive rapport. Derfor vil jeg gerne have muligheden for at have nogle opgaver at give af, hvis jeg kan se at det bliver svært at nå det hele.

I mit projekt er et af acceptance kriterierne at bygge en React JS Frontend til min API. Dette kriterie fik jeg givet som ”ekstra”, og har derfor valgt at sætte user story nr. 5 ind som Slack.

2.2.3 Informative Workspace

Informative Workspace handler om at have diverse ting, som post it notes, på sin arbejdsplads til at informere om det projekt man er i gang med. Det handler også om at arbejde et sted der tilpasser sig dine menneskelige behov. Det kan både være et sted at være social eller et sted at være alene, afhængig af hvad man føler der er nødvendigt. Det er altså vigtigt at det sted man arbejder skal være det sted hvor man kan yde sit bedste.

Jeg har valgt at bruge denne praktik for at holde motivationen oppe igennem hele forløbet. Fordi jeg er alene om projektet, ville det være meget let at blive træt af bare at sidde hjemme hver dag. Derfor vil det være vigtigt for mig at sørge for at have andre muligheder, hvilket Informative Workspace kan hjælpe med. Samtidig vil praktikken også kunne hjælpe mig med at have bedre overblik over hvilke opgaver jeg er i gang med, og hvilke jeg skal arbejde på bagefter.

En måde jeg har sørget for at have et Informative Workspace har været ved at lave et Kanban board på Trello[2]. Det er den måde jeg har holdt styr på hvor langt jeg har været med mine user stories og de tilhørende opgaver. En anden måde jeg har brugt denne praktik på, er ved altid at have mulighed for at tage ud på Monthio’s kontor og arbejde derfra. Her har jeg kunne snakke med medarbejdere og deltage i sociale arrangementer og andet. På den måde har jeg ikke bare siddet alene med opgaven i hele perioden, men arbejdet der hvor jeg har kunnet yde mit bedste.

[2] https://trello.com/

2.2.4 Energized Work

Energized Work går ud på at man skal være frisk og veloplagt, så man kan fokusere på sine arbejdsopgaver og være produktiv. Af den grund skal man undgå overarbejde, så der er tid til fritidsaktiviteter og privatliv osv. Hvis man overarbejder for meget, kan det føre til at man er træt, hvilket kan resultere i at der bliver taget smutveje og lavet fejl.

Jeg bruger denne praktik fordi den vil hjælpe mig med at sprede arbejdsmængden over hele projektperioden. Hvis jeg sørger for at arbejde hver hverdag fra starten, undgår jeg at sidde med en kæmpe mængde opgaver til sidst, og være tvunget til overarbejde i en periode.

Jeg har fulgt denne praktik ved at gå tidligt i gang med projektet, og have nogenlunde faste arbejdstider hver hverdag fra cirka 9-15. Fordi jeg har arbejdet alene har jeg kunne være meget fleksibel, og derfor kunnet rykke arbejdstiderne rundt fra uge til uge som det bedst passede mig. Nogle dage har jeg arbejdet mindre og nogle har jeg arbejdet mere, afhængig af hvor frisk og veloplagt jeg har været.

2.2.5 Incremental Design

XP er imod at gå efter at producere et færdiggjort design af et produkt fra starten af. I stedet for bliver der foreslået at man designer trinvist imens der kodes, på den måde undgår man lettere at gentage sig i koden. Hvis man producerer koden hurtigt, kan man hurtigere få feedback og derefter lave forbedringer. Denne tilgang fungerer oftest bedre, fordi et up-front design ikke tager hensyn til de fejl der bliver opdaget senere.

Jeg vil bruge denne praktik af flere grunde. For det første kan det være svært at have et overordnet overblik over hvordan produktet skal ende med at se ud, inden man er startet med at kode. Derfor vil det være lettere at trinvist skabe sig et overblik over det næste stykke kode der skal produceres, inden man begynder på det. Når den første del er færdig, vil jeg begynde på næste del og så videre. For det andet vil denne trinvise proces gøre så jeg ikke kommer til at skabe et forkert design, og så kode det hele før jeg opdager at det var forkert. Hvis dette skulle ske, vil det trinvise design gøre at jeg kun kan komme til at kode en lille del forkert, inden jeg kommer tilbage på rette spor.

Jeg har brugt Incremental Design ved først at kode noget der virker, hvorefter jeg har snakket med min kontaktperson og fået feedback til hvordan det kunne forbedres. Dette har for eksempel været ved sprint reviews, men også bare hvis jeg i midten af et sprint har villet sikre mig at jeg var på rette spor, for at undgå at kode videre på noget der ikke var nødvendigt.

3. Acceptance kriterier

Jeg er blevet givet disse acceptance kriterier af virksomheden:

1. Application should be written in dotnet 6.0 as a web api

2. Application itself should be stateless and all state should be persisted in a database layer. SQL database preferably with entity framework as Object-Relational-Mapping

3. All business logic and API's should be covered by automated tests (Preferably xUnit framework)

4. There should be a management API (CRUD) to create a webhook definition.

5. Webhook definition entity should include fields like: tenant id(ClientId), destination url, http header name + value (optional + allow multiple), client certificate (optional)

6. If http header name + values are defined, those should be attached to the http calls when webhooks are fired

7. If a client certificate is defined, it should be attached to the http calls when webhooks are fired

8. Webhook engine should manage sending and updating statuses of fired webhooks. If a webhook fails, simple retry logic should be defined. Preferably this retry logic is configurable.

9. There should be a status API to read the status(pending, successful, retrying, failed, stopped) of existing fired webhooks. + Endpoint that allows resending failed webhook or stopping pending and retrying webhooks.

10. **Extra**: Build React JS frontend for the API's

3.1 Product Backlog

For at strukturere mit arbejde vil jeg starte med at lave mine egne user stories der dækker alle acceptance kriterierne. Jeg vil bruge Trello til at lave et Kanban Board, hvor jeg let kan holde styr på mine user stories. Her vil jeg lave en tjekliste for hver user story med de opgaver det vil kræve at gennemføre dem. Jeg vil tildele hver user story både en prioritet og et story point, for at hjælpe mig med bedre at kunne fordele dem i sprints. Prioritet skal klargøre hvor vigtig for virksomheden gennemførelsen af en user story er. Og story point repræsenterer mængden af tid det vil tage at færdiggøre en user story. Jeg har valgt at begge ting kan tildeles enten 1, 3 eller 5, hvor 5 er den største prioritet og længste opgave. Grunden til jeg har valgt at skalaen kun består af tre tal, er fordi jeg syntes det gør det lettere at estimere user stories. En for stor skala kan gøre det svært at bedømme især sværhedsgraden, og dermed mængden af tid der skal bruges på en user story. Derfor ville en skala bestående af for eksempel 10 tal kunne gøre estimeringen mere besværlig, da forskellen på for eksempel syv og otte er svær at vurdere.

Udover at bruge Given-When-Then til hver enkelt user story, som beskrevet under XP praktikken ”Testable”, vil jeg også bruge unit tests til at teste individuelle stykker kode i programmet for at sikre mig alt virker som det skal.

1.	**Som bruger vil jeg kunne oprette og en Webhook definition, så jeg ud fra den kan oprette webhooks der kan afsendes når en bestemt status i systemet opdateres. (Prio: 5, SP: 1)**
Given	Når jeg er på hjemmesiden
When	Og opretter en webhook definition
Then	Gemmes den til senere brug

2.	**Som bruger vil jeg kunne oprette 1 eller flere Webhooks tilhørende en webhook definition, så jeg kan konfigurere flere webhooks til samme endpoint. (Prio: 5, SP: 3)**
Given	Når jeg er i gang med at oprette en webhook definition
When	Og vil tilføje en webhook til definitionen
Then	Kan jeg udfylde information vedrørende den webhook

3.	**Som bruger vil jeg kunne oprette 0 til mange http headers tilhørende en webhook definition, så jeg kan kontrollere hvilke headers jeg vil medsende i en webhook. (Prio: 5, SP: 3)**
Given	Når jeg er i gang med at oprette en webhook definition
When	Og vil tilføje en header til definitionen
Then	Kan jeg udfylde information til Headeren

4.	**Som virksomhed vil Monthio have at alt data skal gemmes i en database med entity framework som Object-Relational-Mapping, så webhookapplikationen er stateless. (Prio: 5 , SP: 5)**
Given	Når en bruger opretter en webhook definition
When	Og trykker gem
Then	Bliver den gemt i en database

5.	**Som bruger vil jeg have et status API hvor jeg kan se status på mine webhooks, så jeg kan se om mine webhooks virker som de skal. (Prio: 3, SP: 5)**
Given	Når jeg er på hjemmesiden
When	Og trykker ind på siden for webhook status
Then	Kan jeg se status på mine webhooks

6.	**Som virksomhed vil Monthio have at API'en og resterende kode er dækket af automatiske unit tests, så fejl hurtigere bliver opdaget. (Prio: 3, SP: 3)**
Given	Når en udvikler har skrevet ny kode til projektet
When	Og vil oprette et pull request
Then	Bliver alle tests kørt igennem og skal være succesfulde før merging kan lade sig gøre

7.	**Som virksomhed vil Monthio have at Webhook applikationen skal kunne sende og opdatere webhooks når en status i systemet ændres, så deres kunder automatisk bliver notificeret om ændringerne. (Prio: 5, SP: 5)**
Given	Når en kunde har oprettet en webhook i systemet
When	Og den relevante status ændres
Then	Bliver webhooken afsendt til det specificerede endpoint

8.	**Som virksomhed vil Monthio have at metoden for afsendelse af webhooks indeholder konfigurerbar "retry logic", så der ikke bare bliver givet op efter 1 forsøg og webhooken aldrig bliver sendt.** **(Prio: 5, SP: 3)**
Given	Når en webhook i systemet bliver sendt
When	Og den af en eller anden grund ikke bliver afsendt succesfuldt
Then	Bliver afsendelsen forsøgt igen senere

4. Sprint

Som skrevet i metodeafsnittet har jeg sat otte uger af til produktudvikling. Og på de otte uger har jeg fordelt mine user stories i tre sprints. Det første sprint skal vare tre uger, mest fordi jeg havde planlagt at bruge meget tid på research i dette sprint. Det andet skal også vare tre uger, og her skal der bruges lidt mindre tid på research mens user stories er sværere. Og det sidste skal vare 2 uger da der ikke er meget tid tilbage, og den ene user story i dette sprint er også sat som Slack, og kan derfor droppes hvis jeg løber tør for tid. I slutningen af hvert sprint holder jeg et sprint review med min kontaktperson som Product Owner. Her gennemgår vi hvor jeg langt jeg nåede med det planlagte arbejde, hvad der er færdigt, og hvad der mangler og skal rykkes til senere sprint.

4.1 Sprint 1

Det første sprint består af fire user stories og 10 story points. Det er de user stories med højest prioritet og laveste story points. På den måde kan jeg fokusere på hurtigt at få udviklet det vigtigste først. Sprintet består af user story 1, 2, 3 og 8.

User Story 1: Som bruger vil jeg kunne oprette og en Webhook definition, så jeg ud fra den kan oprette webhooks der kan afsendes når en status i systemet opdateres.
(Prio: 5, SP: 1)

1. Brugeren skal kunne angive DestinationUrl
2. Brugeren skal kunne angive TenantId
3. Brugeren skal kunne angive RetryCount
4. Brugeren skal kunne angive RetryTimeSpan
5. Brugeren skal kunne angive Client Certificate

User Story 2: Som bruger vil jeg kunne oprette 1 eller flere Webhooks tilhørende en webhook definition, så jeg kan konfigurere flere webhooks til samme endpoint.
(Prio: 5, SP: 3)

1. Brugeren skal kunne skrive en besked der følger med en webhook
2. Brugeren skal kunne bestemme hvilket event der afsender en webhook
3. Webhook skal indeholde antal gange den er blevet afsendt
4. Webhook skal indeholde status

User Story 3: Som bruger vil jeg kunne oprette 0 til mange http headers tilhørende en webhook definition, så jeg kan kontrollere hvilke headers jeg vil medsende i en webhook.
(Prio: 5, SP: 3)

1. Brugeren skal kunne angive HeaderName
2. Brugeren skal kunne angive HeaderValue

User Story 8: Som virksomhed vil Monthio have at metoden for afsendelse af webhooks indeholder konfigurerbar "retry logic", så der ikke bare bliver givet op efter 1 forsøg og webhooken aldrig bliver sendt.
(Prio: 5, SP: 3)

1. Der skal være en metode for afsendelse af webhooks der indeholder konfigurerbar "retry logic"

Tre af de her user stories er rimelig lige til, derfor har jeg valgt kun at uddybe mit arbejde med user story 8, da det udover selve researchen var den der tog det meste af min tid. Målet med denne user story er at opfylde den del af acceptance kriterie nr. 8, der går ud på at definere konfigurerbar "retry logic".

Efter en del research og overvejelser om hvordan jeg skulle implementere logik til automatiske genforsøg, kom jeg frem til at bruge et meget populært framework der hedder Polly[3]. Jeg valgte at bruge Polly fordi man med meget få linjer kode kan få fejlede http requests til at lave genforsøg. Man kan også meget let konfigurere hvornår og hvor mange gange der skal prøves igen. For eksempel hvis grunden til at et request fejler er noget der ikke vil virke lige meget hvor meget man genforsøger, er der ingen grund til overhovedet at prøve igen, i hvert fald ikke lige med det samme. Det kunne være at der simpelthen var lavet en skrive fejl i URL'en i en webhook, i det tilfælde giver det ingen mening at forsøge at sende det igen. I det tilfælde ville det give mere mening at stoppe og tilføje webhooken til en liste med webhooks der ikke virkede. I andre tilfælde kan grunden til en fejl være en forbigående fejl på netværket eller infrastrukturen du afhænger af. Her ville det give god mening at forsøge afsendelsen af webhooken igen, da systemet højst sandsynligt vil være oppe og køre igen kort tid efter.

For at vide hvordan jeg har implementeret det i min applikation er her først et billede af en metode der sender en webhook til den tilhørende DestinationUrl og Body, og returnerer en HTTPResponseMessage.

[3] Home · App-vNext/Polly Wiki (github.com)

```
public async Task<HttpResponseMessage> SendWebhook(WebhookStatus webhookStatus)
{
    if (webhookStatus.Config.Headers != null)
    {
        foreach (var h :Header in webhookStatus.Config.Headers)
        {
            _client.DefaultRequestHeaders.Add(h.HeaderName, h.HeaderValue);
        }
    }

    var content = new StringContent(webhookStatus.Body, Encoding.UTF8, mediaType: "application/json");
    HttpResponseMessage response;
    try
    {
        response = await _client.PostAsync(webhookStatus.Config.DestinationUrl, content);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        throw;
    }

    return response;
}
```

Da den metode var lavet, kunne jeg lave en anden metode, hvor selve konfigurationen af min logik for genforsøg fandt sted. Til at starte med kræver det at man installerer de nødvendige NuGet Packages i sin solution. Når det er gjort kan man oprette sine ønskede Politikker. På nedenstående billede opretter jeg en Politik der hedder waitAndRetryPolicy, hvorefter jeg bruger Handle<RequestException> til at angive hvilke exceptions den skal forsøge igen på. Så bruger jeg OrTransientHttpError som er en måde at konfigurere politikken til at forsøge igen på forbigående fejl

```
public async Task<WebhookStatus> ExecuteWebhookWithPollyRetry(WebhookStatus webhookStatus)
{
    var retryDelay :TimeSpan = TimeSpan.FromSeconds(webhookStatus.Config.RetryTimeSpan);
    var retryCount :int = webhookStatus.Config.TryCount;

    IAsyncPolicy<HttpResponseMessage> waitAndRetryPolicy = Policy<HttpResponseMessage>
        .Handle<HttpRequestException>()
        .OrTransientHttpError() // PolicyBuilder<HttpResponseMessage>
        .WaitAndRetryAsync( sleepDurations: Backoff.DecorrelatedJitterBackoffV2(retryDelay, retryCount));

    var response = await waitAndRetryPolicy.ExecuteAsync(() => SendWebhook(webhookStatus));
    _db.SaveWebhookStatusAndHistory(webhookStatus, response);

    return webhookStatus;
}
```

Herefter bruger jeg WaitAndRetryAsync(Backoff.DecorrelatedJitterBackoffV2(retryDelay,retryCount)) til at angive et start delay på hvor lang tid der skal gå imellem de resterende forsøg, samt hvor mange gange der skal forsøges igen. En ligetil retry strategi kunne for eksempel være simpel eksponentiel backoff, hvor den starter ud med at prøve hurtigt efter det tidligere forsøg, hvorefter der går længere og længere tid imellem. Det kunne se sådan her ud: 2, 4, 8, 16, 32 osv. Dette kan være et problem i scenarier hvor der er mange webhooks der skal sendes, fordi hvis alle sammen fejler på samme tid, bliver de alle sammen forsøgt igen på præcis samme tid igen, hvilket kan belaste systemet. I stedet for anbefales det af udviklerne bag Polly at man bruger en algoritme der implementerer jitter i sin

retry strategi[4]. Jitter går ud på at tilføje tilfældighed i ventetiden mellem genforsøg, hvilket resulterer i at hvert request bliver affyret med små mellemrum, så systemet ikke bliver belastet så meget. Den anbefalede formel er derfor DecorrelatedJitterBackoffV2, som sørger for en god fordeling af genforsøg.

Til sidst kalder jeg den tidligere beskrevede metode der sender webhooks igennem min waitAndRetryPolicy. På den måde kører SendWebhook igennem min politik og forsøger det antal gange der er angivet, før den enten giver op eller får en succes statuskode returneret. Herefter bruger jeg SaveWebhookStatusAndHistory(webhookStatus, response) til at gemme webhooken i de relevante tabeller i databasen, hvilket selvfølgelig afhænger af den HTTPResponseMessage der er returneret af requestet.

Som man kan se er slutresultatet forholdsvist simpelt med kun en retrypolitik. Tidligt i arbejdet med de her politikker var jeg lidt for opsat på at få rigtig meget af systemet til at køre igennem politikker. Det resulterede i at jeg brugte en del tid på at implementere diverse politikker som jeg kunne køre min SendWebhook metode igennem. Efter en snak med min kontaktperson fandt vi frem til at det nok kun gjorde opgaven sværere end den behøvede at være. Derfor endte jeg med at tage et par skridt tilbage og kun bruge WaitAndRetryAsync, og kode resten af retry logikken uden andre politikker.

En af grundene til at jeg fandt frem til det, var fordi jeg havde tænkt mig bl.a. at implementere en politik der hedder FallBack Policy[5]. Politikken går ud på at definere hvad der skal ske når noget fejler. Derfor var min idé var at få den her politik til at forsøge at sende en failed webhook for eksempel to timer senere. Men min kontaktperson brugte en analogi der hedder ”Pets vs Cattle” til at forklare hvorfor det ikke nødvendigvis var en god idé. Monthio har måske 100 applikationer der kører hele tiden, det er vores Cattle. Det vil sige at vi vil have muligheden for at dræbe dem når som helst for at starte nye programmer. Og når de dræbes skal de ikke være midt i et to timers forsøg på at gensende et http request, som så ender med at forsvinde. I det her tilfælde er det en SQL server der er vores Pet. Det skal vi passe på fordi det er her alt vores data gemmes. Derfor endte jeg med at gemme de webhooks der ikke virkede i databasen, og så på et senere tidspunkt tjekke igennem databasen for webhooks der skal forsøges igen.

4.1.1 Sprint 1 review

I dette sprint planlagde jeg at lave de 4 user stories med højest prioritet med sammenlagt 10 story points.

Under gennemgangen af hvilke opgaver der var færdiggjort blev user story 1,2 og 3 godkendt. Der var dog en ting der manglede. Det var den sidste opgave under user story 1; ”Brugeren skal kunne angive Client Certificate”. PO foreslog at jeg kunne udelade den, så jeg kunne komme videre med de andre user stories som havde højere prioritet. Hvis det viste sig at blive vigtigt, vil det altid kunne blive tilføjet som en separat user story i et fremtidigt sprint. Derfor valgte vi at godkende user story 1.

[4] Retry with jitter · App-vNext/Polly Wiki (github.com)
[5] Fallback · App-vNext/Polly Wiki (github.com)

Den anden ting var user story 8 med at implementere ”retry logic”. Denne user story var ikke helt færdig med endnu, fordi en del af min ”retry logic” skulle foregå via en database der ikke var oprettet endnu. Derfor gav det god mening at sætte user story 4, som går ud på at sætte en database op, i sprint 2. På den måde kunne jeg rykke user story 8 til slutningen af sprint to hvor jeg kunne arbejde videre med den når opsætningen af database var færdiggjort. Det vil sige jeg fik lavet syv ud af de 10 planlagte story points dette sprint, og derfor er lidt bagud efter planen. Dette afspejler min burn-down chart for sprint 1, og den kan findes i bilag under punkt 8.1.

4.1.2 Sprint 1 retrospective

Jeg vil sige at første sprint forløb efter planen. Efter jeg havde brugt en del tid på research, nåede jeg alligevel næsten alt det planlagte arbejde for sprintet. Og de opgaver jeg ikke nåede, var opgaver der ville tage længere tid end jeg lige havde regnet med, eller endte med at låse sig bag en anden user story. Og ved sprint review var PO også enig i at de opgaver skulle rykkes til et senere sprint. Jeg har prøvet at lave alle mine user stories uafhængige af hinanden. Men jeg vidste godt at jeg ikke nødvendigvis havde udfyldt det krav for user story 8, da jeg under udførelsen af den ikke vidste alt hvad den indebar. Jeg vidste at det ville kræve meget research for at finde ud af hvordan jeg skulle implementere den. Og under arbejdet med første sprint viste det sig at user story 8 med ’retry logic’ var afhængig af user story 4 med databasen.

Jeg mener ikke det var noget jeg kunne have forudset, da det først gik op for mig da jeg stødte ind i et problem med koden. Alligevel vil jeg til de efterfølgende sprint sætte mere fokus på uafhængigheden af mine user stories, fordi det kan være besværligt at rykke rundt på for mange opgaver imellem sprints.

4.2 Sprint 2

Det andet sprint består af to user stories og otte story points. Det er igen de user stories med højest prioritet, men med lidt højere story points end første sprint. Derudover består det også til dels af user story 8 som skal færdiggøres her. Derfor består sprintet af user story 4, 7 og til dels 8.

User Story 4: Som virksomhed vil Monthio have at alt data skal gemmes i en database med entity framework som Object-Relational-Mapping, så webhookapplikationen er stateless.
(Prio: 5, SP: 3)

User Story 8: Som virksomhed vil Monthio have at metoden for afsendelse af webhooks indeholder konfigurerbar ”retry logic”, så der ikke bare bliver givet op efter 1 forsøg og webhooken aldrig bliver sendt.
(Prio: 5, SP: 3)
1. Der skal være en metode for afsendelse af webhooks der indeholder konfigurerbar ”retry logic”

User Story 7: Som virksomhed vil Monthio have at Webhook applikationen skal kunne sende og opdatere webhooks når en status systemet ændres, så deres kunder automatisk bliver notificeret om ændringerne.
(Prio: 5, SP: 5)

1. Der skal være en metode der automatisk kan afsende de relevante webhooks når en status i systemet ændres

Jeg vil starte med at uddybe mit arbejde med user story 4, der går ud på at opfylde acceptance kriterie nr. 2. Derfor skulle jeg også finde ud af hvordan jeg kunne bruge et entity framework som Object-Relational-Mapping.

Det første jeg skulle overveje var om jeg ville have bruge en code-first eller database-first tilgang. Jeg valgte at bruge code-first da jeg allerede havde mine modelklasser, og ved at ændre lidt i dem ville jeg automatisk kunne danne en database, samt migrations dertil. Under arbejdet med udviklingen af produktet har jeg brugt min egen database. Derfor vil denne tilgang også gøre det let for Monthio at oprette disse tabeller i deres egen database senere hen, kun ved at ændre en connection string.

Herefter skulle jeg ændre i mine modelklasser så de autogenererede tabeller ville komme til at opfylde første, anden og tredje normalform. Samtidig med at nogle tabeller ikke må indeholde fortrolige detaljer. Jeg har her et par eksempler på hvordan jeg fik lavet mine model-klasser så de både opfyldte normalformene samtidig med mine user stories.

Eksempel på hvilke værdier jeg ville have i min database, og hvordan jeg opsatte dem:

Den første tabel her er et eksempel på de værdier jeg ville have i min database, men at der skulle ændres til for at opfylde første normalform.

WebhookConfiguration:

Id (PK)	Headers	DestinationUrl	TenantId	Trycount	RetryTimespan	Body	TimesFired	Status
1	{Name1, Value1}, {Name2, Value2}	www.google.dk	ABC-123	3	1	Webhook message	5	Sending

For at tilfredsstille 1. Normalform hvor der ikke må være mere end én værdi i en kolonne, måtte jeg flytte Headers til sin egen modelklasse og dermed også sin egen tabel med en foreign key der tilknytter hver header til en WebhookConfiguration:

WebhookConfiguration:

Id (PK)	DestinationUrl	TenantId	Trycount	RetryTimespan	Body	TimesFired	Status
1	www.google.dk	ABC-123	3	1	Webhook message	5	Sending

Headers:

Id (PK)	Webhook_Id (FK)	HeaderName	HeaderValue
1	1	Name1	Value1
2	1	Name2	Value2

Derefter skulle WebhookConfiguration opdeles i to tabeller, da det for det første skulle være muligt at oprette flere Webhooks til en WebhookConfiguration. For det andet skal en webhook ikke indeholde fortrolige ting som TenantId og DestinationUrl, men kun det man gerne vil have vist i et UI der viser status på webhooks.

WebhookConfiguration:

Id (PK)	DestinationUrl	TenantId	Trycount	RetryTimespan
1	www.google.dk	ABC-123	3	1

Headers:

Id(PK)	Webhook_Id (FK)	HeaderName	HeaderValue
1	1	Name1	Value1
2	1	Name2	Value2

WebhookStatus:

Id(PK)	Webhook_Id (FK)	Body	TimesFired	Status
1	1	Webhook message1	3	success
2	1	Webhook message2	5	failed
3	1	Webhook message3	0	Sending

Efter disse rettelser skulle jeg tilføje en primary key(id) til mine klasser, samt angive at både Headers og WebhookStatus skulle have en foreign key der svarede til den tilhørende WebhookConfigurations Id(PK). Så skulle der laves en klasse hvor jeg vælger hvilke klasser jeg gerne vil have lavet til tabeller i min database:

```
public class WebhookDBContext : DbContext
{
    0 references
    public WebhookDBContext(DbContextOptions options) : base(options)
    {
    }

    2 references
    public DbSet<Header> Headers { get; set; }
    5 references
    public DbSet<WebhookConfiguration> WebhookConfigurations { get; set; }
    4 references
    public DbSet<WebhookStatus> WebhookStatus { get; set; }
    3 references
    public DbSet<WebhookHistory> WebhookHistory { get; set; }
}
```

Og min DbContext klasse skal registreres i Program.cs, og den skal bruge en connection string til den database hvor tabellerne skal oprettes, som her hentes i mine appsettings.json

```
builder.Services.AddDbContext<WebhookDBContext>(x => x.UseSqlServer(builder.Configuration.GetConnectionString("ConStr")));
```

Når alt dette var gjort var det sidste der skulle til for at generere databasen at køre to kommandoer i Package Manager Console:

”add-migration” og ”update-database”.

Efter at have lavet mine modelklasser om, for at autogenerere tabeller i databasen stødte jeg ind i et problem med min API. Grunden til det var at jeg ikke længere kunne bruge min modelklasser til at oprette objekter da id'en var en autogeneret værdi i databasen. Derfor havde jeg brug for en måde at bruge de gamle modelklasser. De skal bruges til min API hvor jeg skal kunne bruge det gamle json format, uden primary keys og foreign keys, til at oprette objekter. Det gør jeg ved at have 2 versioner af den samme modelklasse, en til databasen og en til API'en. Og de skal forbindes via mapping, der går ud på at konvertere en modelklasse til en anden. Jeg var derfor nødt til at lave en metode til mapping for hver af de 3 klasser der bruges i API'en. Det endte ud i den her metode:

```
public WebhookConfiguration Map(WebhookConfigurationApi source)
{
    List<Header> headerList = new List<Header>();
    if (source.Headers != null)
    {
        foreach (var h :HeaderApi in source.Headers)
        {
            var header = _headerMapper.Map(h);
            headerList.Add(header);
        }
    }

    List<WebhookStatus> statusList = new List<WebhookStatus>();
    foreach (var w :WebhookStatusApi in source.Webhooks)
    {
        var webhook = _statusMapper.Map(w);
        statusList.Add(webhook);
    }

    return new WebhookConfiguration()
    {
        Headers = headerList,
        DestinationUrl = source.DestinationUrl,
        TenantId = source.TenantId,
        TryCount = source.TryCount,
        RetryTimeSpan = source.RetryTimeSpan,
        Webhooks = statusList
    };
}
```

Det er en metode der tager en WebhookConfigurationApi som parameter og returnerer en WebhookConfiguration som kan tilføjes til databasen. Metoden starter med at mappe WebhookStatusAPI og HeadersAPI over til deres database version så de til sidst kan tilføjes til WebhookConfiguration objektet nedenunder, hvor de resterende værdier bare bliver konverteret direkte over til det det objekt der skal returneres.

Nedenfor vises der hvordan jeg har brugt _configMapper.Map(webhookConfigurationApi) i min CreateWebhookConfiguration metode for at konvertere værdierne.

```
public async Task<WebhookConfiguration> CreateWebhookConfiguration(WebhookConfigurationApi webhookConfigurationApi)
{
    WebhookConfiguration webhookConfig = _configMapper.Map(webhookConfigurationApi);

    _db.AddConfiguration(webhookConfig);
    return webhookConfig;
}
```

Som nævnt i mit review af sprint 1 rykkede jeg det videre arbejde med user story 8 og "retry logic" til slutningen af sprint 2, når arbejdet med databasen var færdiggjort. Så det arbejde vil jeg beskrive her.

```
public async Task RetryAllWebhooks()
{
    foreach (var wh :WebhookStatus in _db.GetAllWebhookStatuses())
    {
        if (wh.CurrentFailedAttempts is >= 1 and <= 3)
        {
            Console.WriteLine("trying to fire webhook: " + wh.Id);
            await ExecuteWebhookWithPollyRetry(wh);
        }
    }
}
```

Måden jeg kom frem til at køre igennem de fejlede webhooks, var ved at implementere en Background Task[6] i min API. Det gjorde jeg ved først at lave en metode der hed RetryAllWebhooks, der går igennem tabellen med webhooks, og finder dem med en CurrentFailedAttempts værdi imellem 1-3. Når en webhook fejler I den tidligere beskrevede metode ExecuteWebhookWithPollyRetry bliver den gemt i databasen først med en CurrentFailedAttempts værdi på 1, så jeg kan finde den senere i RetryAllWebhooks og genforsøge den. Hvis det lykkedes at sende den her, vil den blive opdateret i databasen med en CurrentFailedAttempts værdi på 0. Hvis den fejler igen bliver værdien plusset med et, indtil den er over tre hvorefter den ikke vil blive genforsøgt mere, og også blive gemt i en anden tabel over fejlede webhooks med den relevante information om fejlen.

Herefter fik jeg RetryAllWebhooks til at køre på et konfigurerbart tidsinterval, ved at implementere den som background task som en hosted service. En hosted service er en klasse med background task logic der implementerer IHostedService interfacen. Derfor starter jeg med at lave min klasse sådan her:

```
public class WebhookBackgroundService : IHostedService, IDisposable
```

Hvorefter den skal registreres i Program.cs således:

```
builder.Services.AddHostedService<WebhookBackgroundService>();
```

[6] Background tasks with hosted services in ASP.NET Core | Microsoft Learn

Når det er gjort kan jeg tilføje kode til klassen for at få den til at køre RetryAllWebhooks som background task.

```
public Task StartAsync(CancellationToken stoppingToken)
{
    var backgroundServiceInterval :TimeSpan = TimeSpan.FromHours(2);
    _logger.LogInformation( message: "Webhook Background Service running.");
    _timer = new Timer(RetryAllWebhooks, state: null, TimeSpan.Zero, period: backgroundServiceInterval);
    return Task.CompletedTask;
}
```

Ovenstående er den metode der kommer til at køre i baggrunden af min API. Først angiver jeg et tidsinterval, i det her eksempel kører den en gang hver anden time. Hvorefter jeg logger at metoden er i gang. Så kører jeg metoden RetryAllWebhooks på en timer hvor jeg bl.a. angiver tidsintervallet.

4.2.1 Sprint 2 review

I dette sprint var den originale plan at lave user story 4, 7 og færdiggøre user story 8. I alt ville det være 11 story points i stedet for de planlagte otte.

Under gennemgangen med PO af hvilke opgaver der var færdiggjort, blev user story 4 og 8 godkendt, hvilket betød at jeg havde lavet seks story points dette sprint. Jeg henviser til min burn-down chart for sprint 2 i bilag under punkt 8.2, der viser at jeg ikke nåede de otte originalt planlagte story points.

Jeg var så småt begyndt på user story 7, men havde lavet langt fra nok til at den kunne blive godkendt af PO. Derfor blev vi enige om at rykke den til sprint tre.

4.2.2 Sprint 2 retrospective

I mit andet sprint viste det sig jeg havde lavet en fejlbedømmelse på hvor meget jeg kunne nå. Grunden til det var user story 4 der tog længere tid en planlagt. Jeg havde tildelt den tre story points, men når jeg ser tilbage på hvor lang tid den tog burde jeg have givet den fem. Det viste sig også at være en udfordring at lave det jeg manglede fra user story 8. Fordi begge disse user stories tog så lang tid som de gjorde, nåede jeg kun kort at kigge på user story 7.

Jeg vidste godt i forvejen at det kan være svært at bedømme hvor lang tid en user story vil tage. Det var derfor jeg valgte kun at give dem værdierne 1-3-5. Alligevel endte jeg med at ramme ved siden af på disse to user stories. Som sagt tror jeg det ville passe fint at give user story 4 fem story points. Men user story 8 havde allerede fem story points som var maksimum, hvilket får mig til at tro en skala med måske fem tal i stedet for tre kunne have fungeret bedre for mig.

4.3 Sprint 3

Det tredje sprint består af to user stories og otte story points. Det er ligeså mange story points som sprint to, men begge user stories har en lavere prioritet og er derfor placeret i mit sidste sprint. Og som nævnt i sprint reviewet for sprint to er user story 7 blevet rykket hertil. Så nu består sprintet af user story 5, 6 og 7.

User Story 7: Som virksomhed vil Monthio have at Webhook applikationen skal kunne sende og opdatere webhooks når en status systemet ændres, så deres kunder automatisk bliver notificeret om ændringerne.
(Prio: 5, SP: 5)

1. Der skal være en metode der automatisk kan afsende de relevante webhooks når et event i systemet sker

User Story 5: Som bruger vil jeg have et status API hvor jeg kan se status på mine wehooks, så jeg kan se om mine webhooks virker som de skal.
(Prio: 3, SP: 5)

User Story 6: Som virksomhed vil Monthio have at API'en og resterende kode er dækket af automatiske unit tests, så fejl hurtigere bliver opdaget.
(Prio: 3, SP: 3)

I dette sprint startede jeg med user story 7 som originalt var planlagt til mit andet sprint. Målet med den er at opfylde den del af acceptance kriterie nr. 8, hvor opgaven var at implementere en måde hvorpå jeg automatisk kan afsende webhooks med den relevante trigger når et event i systemet sker. Før jeg kunne gå i gang med det var jeg nødt til at lave en ændring i min database og tilføje ”TriggerEvent” til mit Webhook objekt. Mens jeg alligevel var i gang med at ændre i databasen, tilføjede jeg også en tabel kaldet ’WebhookHistory’. Det havde jeg snakket med PO om at tilføje, for at bedre kunne holde styr på hvor mange webhooks der var succesfulde og hvor mange der fejlede og hvorfor. Dette medførte også ændringer i mit ER-Diagram. Det endelige ER-Diagram kan ses i bilag under punkt 8.5.

Til det valgte jeg at bruge et publisher/subscriber mønster. Det er en måde at sende beskeder på, hvor afsenderen, kaldet publishers, ikke sender beskederne til nogen bestemt modtager. I stedet for giver publishers et navn(topic) til deres beskeder, som bliver sendt til en slags mellemmand, hvor beskederne bliver håndteret. Så kan modtagere, kaldet subscribers, selv opsætte topics på de beskeder de gerne vil modtage, hvorefter mellemmanden sørger for at sende beskederne ud til de rette subscribers. Grunden til jeg valgte denne måde at gøre det på, er fordi Monthio allerede har steder i deres sofware hvor de bruger dette mønster. Det vil sige at der allerede bliver published de beskeder som jeg skal have min API til at subscribe til.

Jeg skulle finde en måde at implementere automatisk afsendelse af webhooks som en anden background task end RetryAllWebhooks, der bliver kørt en gang hver 2. time, hvilket jeg ikke kunne bruge til denne opgave. Derfor fandt jeg frem til at gøre det med Azure Functions[7], som er en anden måde at automere background tasks på.

Selvom denne del af mit produkt godt kunne bruge lidt videre arbejde, vil jeg give et eksempel på hvordan jeg implementerede en af mine Azure Functions, der afsender webhooks når et dokument bliver uploadet til en case. Det gjorde jeg ved at tilføje et nyt projekt til min solution kaldet "Azure Functions", og tilføje en reference til mit webhook-api projekt, hvor jeg oprettede den her metode:

```
public async Task SendAllWebhooksWhereTriggerDocumentUploaded()
{
    foreach (var wh :WebhookStatus in _db.GetAllWebhookStatuses())
    {
        if (wh.TriggerEvent == "document-uploaded")
        {
            await ExecuteWebhookWithPollyRetry(wh);
        }
    }
}
```

Dens formål er at køre ExecuteWebhookWithPollyRetry metoden på alle de webhooks den finder med TriggerEventet "document-uploaded". Da det var gjort, skulle den udføres i en Azure Function der bliver aktiveret når der sker et event, altså når en subscriber modtager en besked.

```
[FunctionName("SendAllWebhooksForTriggerDocumentUploaded")]
0 references
public static async Task Run([
    ServiceBusTrigger
    (topicName: "webhook", subscriptionName: "document-uploaded", Connection = "WebhookDataConnection")] ILogger logger)
{
    await _webhookService.SendAllWebhooksWhereTriggerDocumentUploaded();
    logger.LogInformation(message: "WEBHOOKS SENT FOR UPLOADED DOCUMENTS");
}
```

Den ovenstående Azure Function har en ServiceBusTrigger, som jeg bruger som min mellemmand mellem publishers og subscribers. I den ServiceBusTrigger angiver jeg topicName, subscriptionName, for at sørge for at min funktion bliver aktiveret af det korrekte event. Herefter kører jeg metoden SendAllWebhooksWhereTriggerDocumentUploaded.

Selvom jeg fik fundet en måde til automatisk afsendelse af webhooks, var der et par ting jeg gerne ville have implementeret, men var nødt til at lade ligge for at nå videre med næste user story. Jeg ville gerne have haft en måde hvorpå min azure function tjekkede op på et ID for hver enkelt sag, så der specifikt bliver afsendt webhooks for de relevante sager.

[7] Azure Functions Overview | Microsoft Learn

Til at slutte projektet af med arbejdede jeg på user story 6, der gik ud på at opfylde acceptance kriterie nr. 3. Jeg valgte at bruge xUnit framework, da det er det Monthio foretrækker. Under mit arbejde med unit tests isolerede jeg enkelte dele af koden for at teste dem individuelt. Og jeg vil her vise hvordan jeg gjorde dette for min Post metode CreateWebhook. Når metoden udføres bliver der kørt et par metoder i baggrunden for at tilføje objektet til databasen. Men det eneste jeg vil teste i denne metode, er at den returnerer det forventede resultat. Det vil sige at jeg skulle finde en måde at teste den uden at ramme databasen. Til det fandt jeg frem til at bruge Moq[8] som er et framework der kan bruges til at lave "mock objects", der er simulerede objekter som efterligner rigtige objekters opførsel. Det brugte jeg til at lave et "mock object" af min IWebhookService.

Derudover brugte jeg Autofixture[9] som er et open-source library til .NET, der er designet til at minimere 'Arrange' fasen i unit tests.. Det vil jeg bruge til at lave test data, da ehvilken data jeg tester ikke er vigtig, men bare at det er den rigtige type objekt der bliver returneret.

På nedenstående billede viser jeg hvordan jeg har brugt Moq til at bruge den simulerede service i testen af controlleren.

```
private readonly IFixture _fixture;
private readonly Mock<IWebhookService> _serviceMock;
private readonly WebhookController _controller;

0 references
public WebhookControllerTests()
{
    _fixture = new Fixture();
    _serviceMock = new Mock<IWebhookService>();
    _controller = new WebhookController(_serviceMock.Object);
}
```

Til mine tests brugte jeg Arrange-Act-Assert som er en god måde at strukturere unit tests på. I 'Arrange', sætter man alle sine værdier og initialisere vores objekter. Så her opretter jeg et nyt testobjekt af typen WebhookConfigurationApi. I 'Act' udfører man selve testen, så her kører jeg CreateWebhook metoden. Til sidst i 'Assert' tjekker jeg om jeg får det forventede resultat, altså at det returnerde objekt er af typen WebhookConfiguration. Dette kan ses på følgende billede:

```
[Fact]
| 0 references
public async Task CreateWebhook_ShouldReturnOkResponse_WhenWebhookCreated()
{
    //Arrange
    var whConfigApi = _fixture.Create<WebhookConfigurationApi>();

    //Act
    var result = await _controller.CreateWebhook(whConfigApi);

    //Assert
    result.Should().BeAssignableTo<ActionResult<WebhookConfiguration>>();
}
```

[8] Testing with a mocking framework - EF6 | Microsoft Learn
[9] UnitTest With AutoFixture In .NET 6.0 (c-sharpcorner.com)

Idéen med at lave unit tests til min kode er at sikre kvaliteten af pull requests. Til det gør Monthio brug af Github Actions[10], som de har opsat så alle unit tests køres igennem, og skal være succesfulde, før et pull requests kan merges. På den måde kan gode unit tests afsløre om der er en fejl i koden inden et pull request kan accepteres. Sådan kan man undgå at sætte fejlkode i produktion, hvilket kan have katastrofale følger for et firma.

4.3.1 Sprint 3 review

I dette sprint var planen at lave user story 5, 6 og 7, hvilket i alt er 13 story points istedet for de originalt planlagte otte for sprint 3.

Under sprint reviewet blev user story 6 med unit tests godkendt. Som jeg nævnte i beskrivelsen af mit arbejde med user story 7, var der et par ting jeg gerne ville have bygget ovenpå. På trods af det mente PO at den skulle godkendes, og det der manglede ville kunne tilføjes som en separat user story i et fremtidigt sprint. User story 5, som var placeret under Slack, havde jeg simpelthen ikke tid til at begynde på. Det betyder at jeg fik godkendt user stories med i alt otte story points hvilket svarer til det originalt planlagte antal. Dette kan også ses ud fra min burn-down chart for sprint 3 i bilag under punkt 8.3.

4.3.2 Sprint 3 retrospective

I mit tredje sprint endte jeg med at have alt for mange story points, eftersom jeg var kommet lidt bagud efter de to første sprints. Dog endte det med at jeg fik lavet otte story points som var det jeg orignalt havde planlagt. Og den eneste user story der manglede var den med lavest prioritet og flest story points som jeg allerede havde placeret som Slack.

Som min burndown-chart for sprint 3 også viser, så havde jeg lavet en god estimering af user stories til dette sprint og jeg fik lavet de planlagte story points.

[10] GitHub Actions Documentation - GitHub Docs

5. Refleksion

Jeg er overordnet godt tilfreds med min rapport, og de agile arbejdsmetoder jeg brugte. De XP praktikker jeg valgte at bruge var en stor hjælp igennem hele forløbet. 'Informative Workspace' og 'Energized Work' hjalp mig meget med at holde motivationen og produktiviteten oppe igennem hele forløbet. 'Stories', 'Slack' og 'Incremental Design' var en stor hjælp da det kom til at strukturere arbejdsopgaver. Alle disse praktikker var med til at sørge for at det lykkedes at holde mig til den planlagte tidsplan.

I starten af perioden da jeg estimerede og planlagde sprints, syntes jeg det var meget svært at bedømme hvor mange af acceptance kriterierne jeg ville kunne nå at opfylde. Monthio havde opstillet gode muligheder for at jeg så vidt muligt selv kunne planlægge mit arbejde ud fra hvad jeg mente jeg kunne nå. Jeg endte med at få lavet 21 ud af 26 story points, som kan ses på min burn-down chart i bilag under punkt 8.4. Taget i betragtning at webhooks ikke er noget jeg havde nogen tidligere erfaring med, og jeg derfor ikke vidste hvad jeg skulle forvente, syntes jeg det var en udmærket estimering.

6. Konklusion

Under arbejdet med projektet har jeg prøvet at svare på de 4 underspørgsmål i min problemformulering. Det har været en lærerig proces hvor jeg har lært om mange nye ting.

Spørgsmål 1

- **Hvordan kan afsendelsen af webhooks implementeres så der ikke bliver opgivet med det samme i tilfælde af fejl?**

Som jeg fandt ud af i sprint 1 under arbejdet med user story 8, var Polly et meget populært framework til at implementere automatiske genforsøg af http requests. Med Polly kan man med meget få linjer beskrive hvordan man vil have et http request til at agere når det fejler. Derudover er det også let at konfigurere denne opførsel i fremtiden, hvis man har brug for at ændre hvornår og hvor mange gange den skal prøve at afsende et request igen.

Jeg kunne dog ikke svare på spørgsmålet udelukkende ved at bruge Polly. Derfor fandt jeg i sprint 2, under mit videre arbejde med user story 8, frem til at bruge Polly i samspil med en background task i min API. Denne background task kører på et konfigurerbart tidsinterval og kigger igennem databasen, og finder de webhooks der tidligere har fejlet og derfor har brug for at bliver genforsøgt. De webhooks bliver så afsendt igen med Polly så den ikke bare giver op med det samme i tilfælde af fejl. Hvis de i sidste ende fejler alligevel, bliver de lagt tilbage i databasen og kan genforsøges på et senere tidspunkt igen.

Spørgsmål 2

- **Hvordan kan webhooks automatisk afsendes når der sker statusændringer?**

Det spørgsmål forsøgte jeg at svare på i sprint 3 i arbejdet med user story 7. Monthio's system udsender allerede webhooks til deres kunder når der er statusændringer. Det gør de ved at bruge et publisher/subscriber mønster. Det vil sige at der allerede bliver published beskeder ved statusændringer. Derfor var det min opgave at finde en måde at få min API til at subscribe til disse beskeder. Jeg fandt frem til at gøre det med Azure Functions, som er nogle funktioner der ligger i baggrunden af API'en og venter på at blive aktiveret af beskeder de er subscribed til. Derefter ligger arbejdet i at få disse funktioner til at køre metoder, der går igennem databasen og finder de relevante webhooks og afsender dem.

Spørgsmål 3

- **Hvilken måde kan webhook data opbevares?**

I acceptance kriterie nr. 2, kan man se at Monthio helst ville have at data bliver opbevaret i en SQL database med entity framework som Object-Relational-Mapping. Derfor var det oplagt at opbevare data om webhooks sådan.

Under arbejdet med user story 4 i sprint 2 fandt jeg frem til at man kan bruge en code-first tilgang. Det går ud på at autogenerere en database samt migrations dertil, ud fra modelklasser hvor man klargører blandt andet primary-og foreign keys. Derefter kan man med en connection string til en database autogenerere sine modelklasser som tabeller i databasen.

Spørgsmål 4

- **Hvordan kan Unit Tests bruges til at sikre kvalitet?**

Det er meningen at en unit test skal teste en lille del af koden, derfor lavede jeg unit tests der hver testede isolerede dele af koden. I sprint 3 under arbejdet med user story 6, viser jeg hvordan man kan bruge Moq framework til at lave simulerede objekter der efterligner rigtige objekters opførsel, så man på den måde kan undgå at ramme for eksempel databasen med sin unit tests. Når ens unit tests er færdige, kan de bruges til at sikre kvalitet ved hjælp af Github Actions. Github Actions kan bruges til at sørge for at alle unit tests køres igennem og er succesfulde inden et pull request kan accepteres. På den måde sikres kvaliteten og der undgås at sætte noget defekt kode i produktion.

7. Litteraturliste

Fodnoter:

1. INVEST:
New to agile? INVEST in good user stories – [Agile for All]

2. Trello:
https://trello.com/

3. Polly wiki:
Home · App-vNext/Polly Wiki (github.com)

4. Polly jitter algorithms recommended for production:
Retry with jitter · App-vNext/Polly Wiki (github.com)

5. Fallback Policy:
Fallback · App-vNext/Polly Wiki (github.com)

6. Background Task:
Background tasks with hosted services in ASP.NET Core | Microsoft Learn

7. Azure function:
Azure Functions Overview | Microsoft Learn

8. Moq:
Testing with a mocking framework - EF6 | Microsoft Learn

9. AutoFixture:
UnitTest With AutoFixture In .NET 6.0 (c-sharpcorner.com)

10. Github Actions:
GitHub Actions Documentation - GitHub Docs

Andre links:

Webhooks:
ASP.NET WebHooks Overview | Microsoft Learn

Polly policyWrap:
PolicyWrap · App-vNext/Polly Wiki (github.com)

Polly backoff reasons:
Exploring the Polly.Contrib.WaitAndRetry helpers (hyr.mn)

Transient fault handling:
Transient fault handling and proactive resilience engineering · App-vNext/Polly Wiki (github.com)

Introduktion til Polly:
What is Polly? (pluralsight.com)

EF Code-first:
What is Code-First? (entityframeworktutorial.net)

EF DB-first:
Entity Framework 6 (entityframeworktutorial.net)

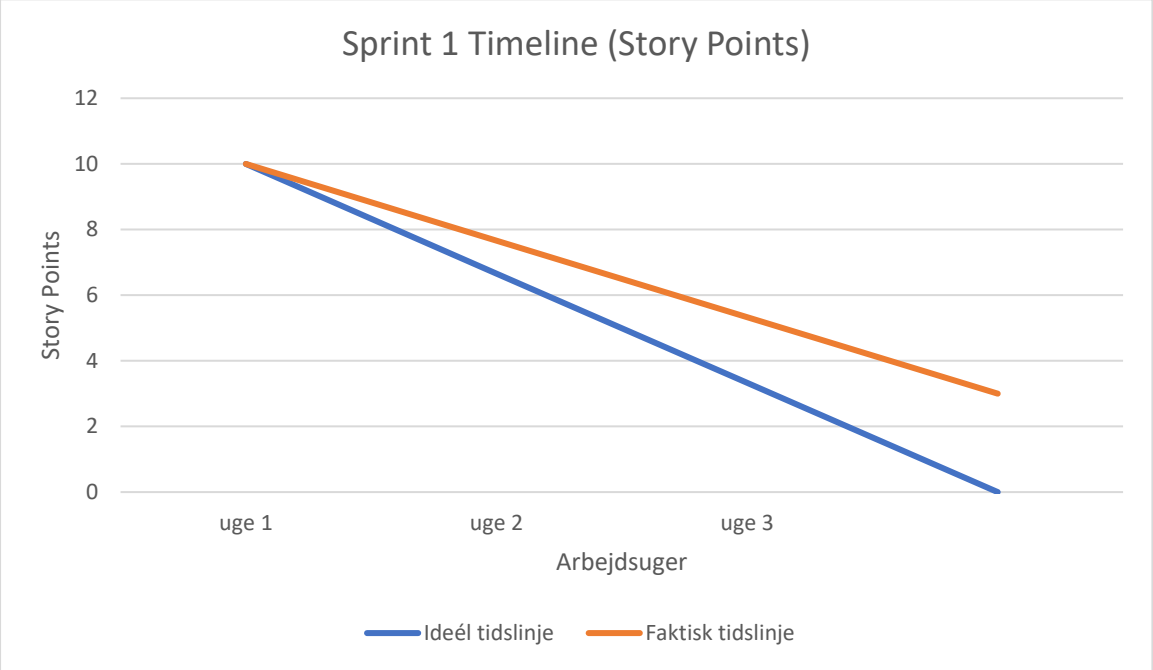
Pub/sub pattern:
Publisher Subscriber (Pub-Sub) Design Pattern (enjoyalgorithms.com)

Pub/sub pattern:
Publisher-Subscriber pattern - Azure Architecture Center | Microsoft Learn

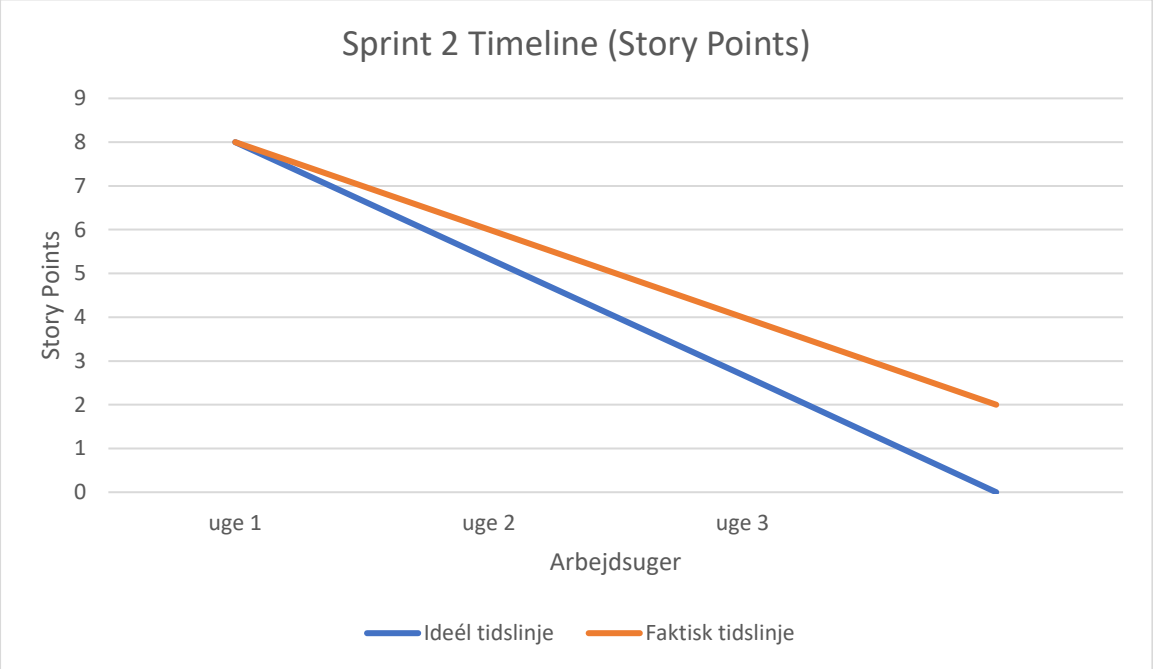
XP:
Articolo su nuova XP (piapetersen.dk)

8. Bilag

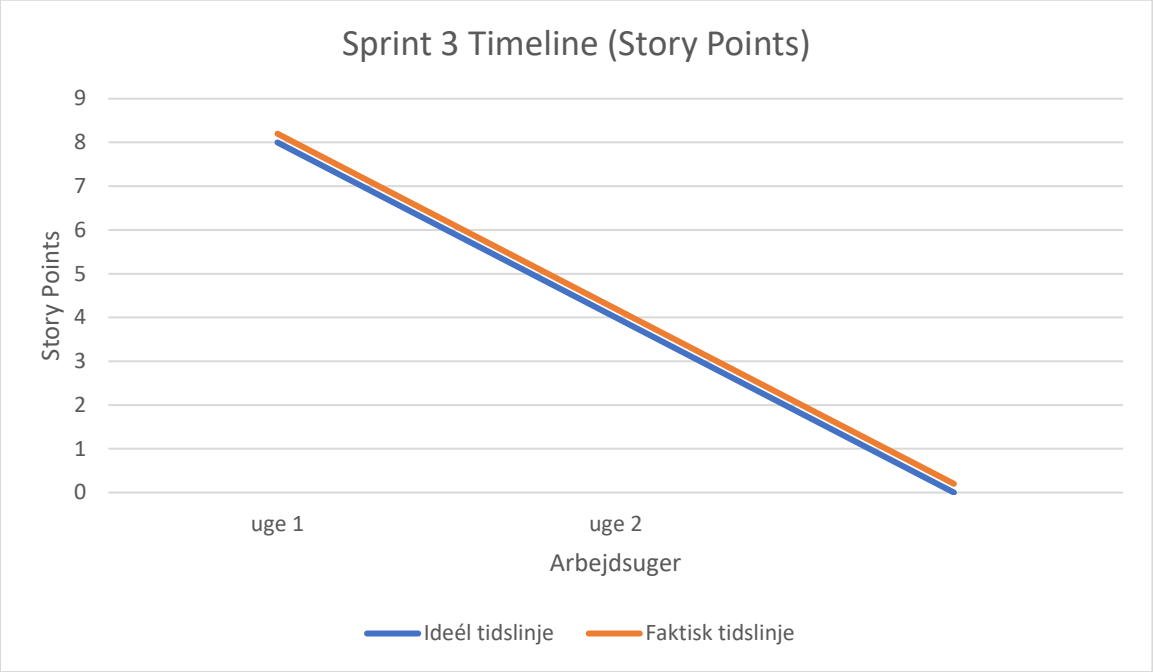
8.1 Bilag 1



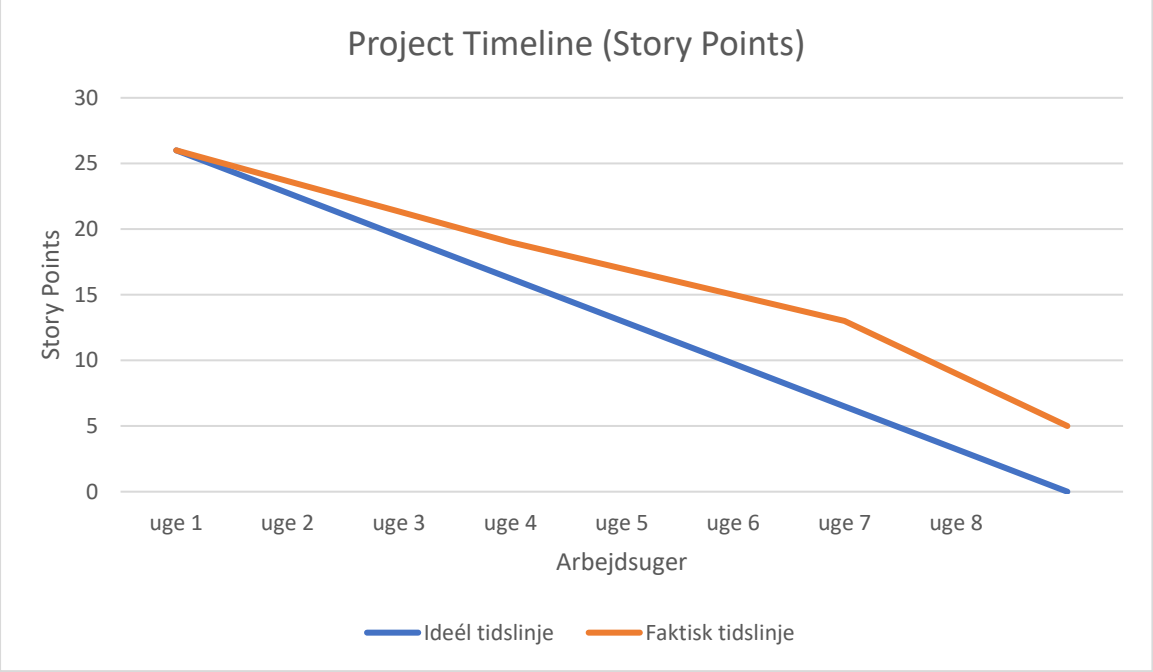
8.2 Bilag 2



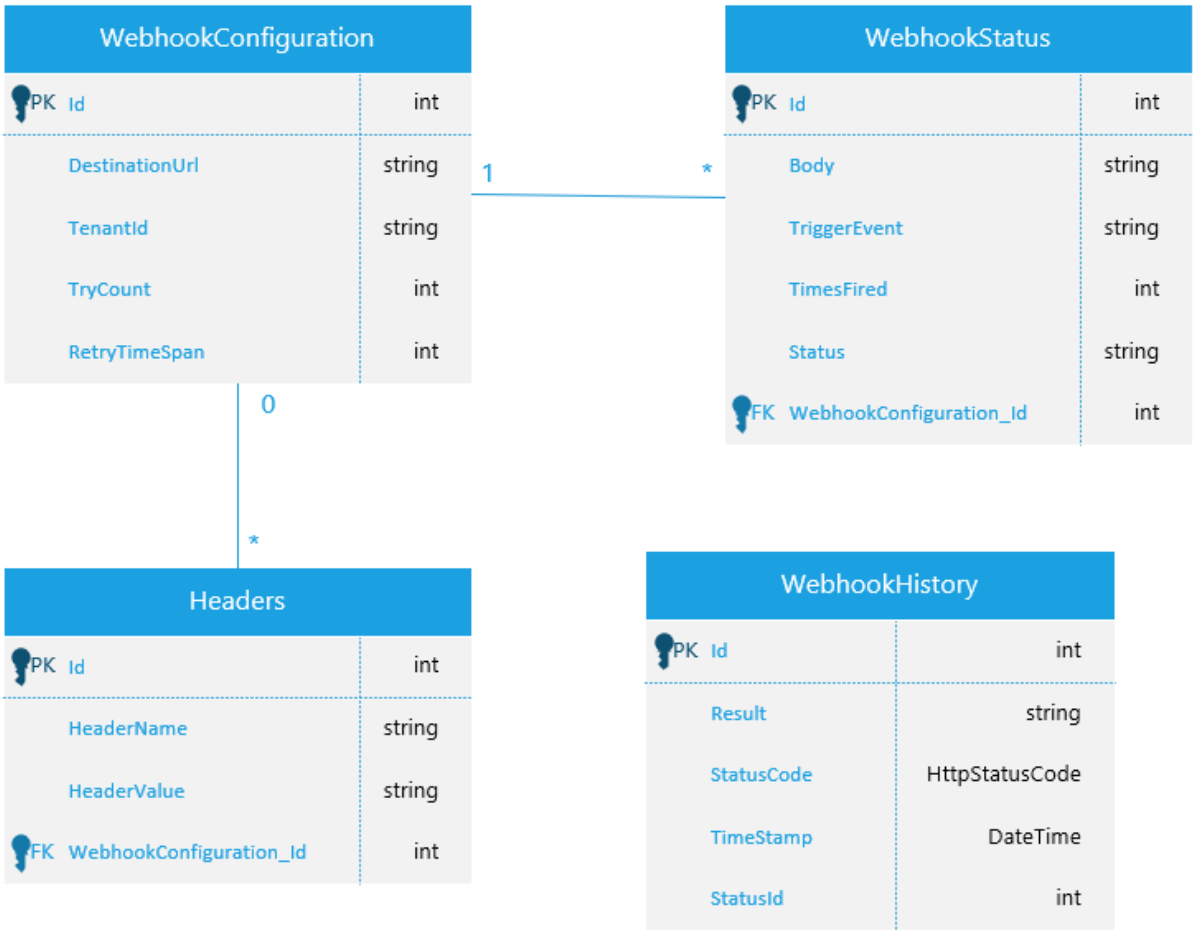
8.3 Bilag 3



8.4 Bilag 4



8.5 Bilag 5


WebhookConfiguration
PK Id int
DestinationUrl string
TenantId string
TryCount int
RetryTimeSpan int
1
*
WebhookStatus
PK Id int
Body string
TriggerEvent string
TimesFired int
Status string
FK WebhookConfiguration_Id int
0
*
Headers
PK Id int
HeaderName string
HeaderValue string
FK WebhookConfiguration_Id int
WebhookHistory
PK Id int
Result string
StatusCode HttpStatusCode
TimeStamp DateTime
StatusId int