Erhvervsakademi Sjælland

Hovedopgave

Datamatiker uddannelsen

# Continuous Integration & Continuous Deployment

*Vejleder*
Michael Claudius

7 Januar 2016

**310187** - Martin Kiersgaard
**091192** - Peter Nielsen

# Indholdsfortegnelse

# 1 Abstract

We have examined the practices and principles behind the Extreme Programming concept of *Continuous Integration* (CI) and *Continuous Deployment* (CD). We have developed a web application, using Continuous Integration and Agile development methods, in order to answer the following questions:

- What are the thoughts behind CI/CD?
- Which are the advantages and disadvantages of implementing CI/CD?
- Which tools are necessary to user CI/CD?
- Which effect does the implementation of CI/CD have on the development proces, both in terms of software quality and the proces as a whole?
- Which development methods are suited for using CI/CD?
- Can we develop an application, by using CI/CD?

We have reached the conclusion that the main advantages and reasons for implementing CI with CD, is to highten the quality and increase the delivery frequency of the software. This is done by automating tasks, like continuous unit testing and deployment to test- and staging-environments, using CI-servers like Jenkins or Team Foundation Server, and providing fast feedback to the developer, if errors occur in the build. Furthermore CI is based on continuous and smaller commits to CI-mandatory Source Code Management systems, resulting in less time spent on integrating code and fixing the resulting merge conflicts.

Although it is not exclusive for the Agile Methodologies, Continuous Integration is an agile practice and is, as such, designed to be used in them. But in theory any iterative development method could make use and benefit from it.

# 2 Projektgrundlag

## 2.1 Introduktion

Vi skal som afsluttende opgave, på Datamatiker-uddannelsen, skrive en rapport omkring et projekt, vi selv er med til at definere. I vores tilfælde har vi indgået et samarbejde med Martins arbejdsgiver; virksomheden TestHuset A/S, der har givet os en problemstilling de gerne vil have løst med et stykke software vi skal udvikle.

Det primære formål med projektet er, at undersøge et emne vi selv har valgt, ved praktisk og teoretisk at arbejde med emnet. I vores tilfælde har vi valgt at ville beskæftige os med *Continuous Integration* og *Continuous Deployment*, der er to DevOps-discipliner med rødder i Extreme Programming.

## 2.2 Virksomheden

TestHuset A/S blev grundlagt i 2005 af nuværende CEO; Allan Tange og Lisbeth Hylke Thomsen. TestHuset har fra begyndelsen været en konsulentvirksomhed der har softwaretest og kvalitetssikring som ekspertise. I dag har TestHuset ca. 65 ansatte, hvoraf 55 er konsulenter. Et nyt område inden for softwareudvikling, som TestHuset er begyndt at beskæftige sig med, er DevOps.

## 2.3 Hvad er DevOps?

DevOps-navnet er en sammentrækning af Development og Operations, hvilket også er meget forklarende for hvad der ligger i betydningen af ordet. Tanken bag DevOps er, at man samarbejder på tværs af udviklings- og driftsafdelingerne i en organisation. Dette giver meget større fleksibilitet og bedre forudsætninger for at udvikle agilt med korte sprint. DevOps er i høj grad kædet sammen med de agile udviklingsmetoder og kan ses som et værktøjslag der er lagt oven på den agile tankegang, kombineret med en solid gang automatisering af den infrastruktur der udvikles på.[1] Et eksempel kunne være opsætning af miljøer til udviklere.

I et udviklingsprojekt der arbejder efter vandfaldsmodellen kan man give udviklingsafdelingen en dato for hvornår miljøer til f.eks. test eller drift skal være klar. Dette vil typisk være flere måneder i forvejen. Driftsafdelingen har dermed masser af tid til at forberede miljøerne og sikre sig at alt fungerer korrekt.

Hvis udviklingsafdelingen derimod begynder at arbejde agilt, hvor der eksempelvis er nye releases hver anden uge, får driftsafdelingen en udfordring. Der skal pludselig afsættes flere ressourcer til at etablere miljøer meget oftere. En populær løsning på dette problem er Continuous Integration og Continuous Deployment.

## 2.4 Vores interesse

Vi har hørt en del omkring CI og CD gennem TestHuset og har fået det indtryk at DevOps generelt er et rigtigt spændende område, med en masse efterspørgsel efter folk der har forstand på dette. Vi har derfor besluttet os for at gå i dybden og undersøge hvad CI/CD helt specifikt er og også forsøge at anvende det i praksis.

Da CI og CD er typer af værktøjer der implementeres i et udviklingsmiljø og ikke noget der decideret skal kodes, har vi også behov for et projekt vi kan udvikle i vores nyopsatte miljø. Til det formål har vi fået stillet en opgave fra TestHuset, der har spurgt om vi vil lave en applikation der kan holde styr på de mange konsulenters kompetencer.

[1] Ernest Mueller et al. *What Is DevOps?* URL: `http://theagileadmin.com/what-is-devops/`.

## 2.5 Problemformulering

Hvis man i dag ønsker at udvikle et stykke software agilt, kommer man hurtigt til at støde på udtrykkene Continuous Integration og Continuous Deployment. Gennem erfaringer fra vores praktikperiode har vi bemærket at der er stor interesse for at implementere CI/CD løsninger hos virksomheder. Derfor har vi tænkt os at undersøge hvad der ligger bag disse praktikker.

**Spørgsmål**

- Hvad er tankerne bag CI og CD?
- Hvilke fordele og ulemper er der ved at implementere CI/CD?
- Hvis man ønsker at anvende CI/CD, hvilke typer af værktøjer er så nødvendige?
- Hvilken effekt har implementeringen af CI/CD på udviklingsprocessen, både i forhold til kvaliteten af softwaren, men også som en helhed?
- Hvilke udviklingsmetoder egner sig til brug med CI/CD?
- Kan vi udvikle en applikation ved hjælp af CI/CD?

**Fremgangsmåde** Vi vil besvare ovenstående spørgsmål gennem både praktisk arbejde og teoretisk research. Vi vil starte med at finde noget litteratur omkring emnet og opsøge professionelle personer med viden omkring CI og CD vi kan interviewe. Dernæst vil vi i denne rapport beskrive den fundne teori bag disciplinerne. Vi skal også undersøge hvilke værktøjer vi kan/skal bruge til at drive et udviklingsprojekt der anvender CI/CD.

Når vi har undersøgt mulighederne og besluttet os for hvilke værktøjer og praktikker vi vil anvende, vil vi udvikle en applikation, for at afprøve vores teoretiske viden i praksis. Det indebærer at opsætte et udviklingsmiljø med de førnævnte værktøjer, fastsætte nogle udviklingsmetoder og drive projektet som et rigtigt udviklingsprojekt.

## 2.6 Ressourcer

Vi har fået stillet en del ressourcer, både tekniske og menneskelige, til rådighed fra TestHusets side. Da Continuous Integration og Continuous Deployment afhænger af værktøjer, er det nødvendigt at have en platform disse kan køre på. Derudover har vi brug for noget feedback fra "kunden" og derfor har vi fået lov til at trække på den viden TestHuset besidder. Både som feedback til de krav der måtte være til projektet, men også faglig viden omkring udviklingsprocessen, programmeringen og platforme.

### 2.6.1 Menneskelige ressourcer

**Michael Claudius** er vores vejleder hos Erhvervsakademi Sjælland. Han vejleder os gennem opgaveskrivningen og giver sin feedback på denne rapport.

**Martin Skovgaard** har titlen COO (Chief Operations Manager) i TestHuset, og har det overordnede ansvar for alle konsulenter i virksomheden. Martin har flere års erfaring med IT-projekter og har tidligere arbejdet som Test-manager hos IO Interactive, samt som selvstændig konsulent. Martin har meldt sig frivilligt til at påtage sig rollen som vores Product Owner.

**Stine Osted** er til dagligt Service Manager i TestHuset. Hun står for ressourceallokering af konsulenter og er leder af den del af virksomheden der håndterer inhouse-opgaver for kunder. Derudover har hun ansvaret for den interne IT i virksomheden og har en masse erfaring med drift af IT-projekter. Stine har tilbudt at give feedback og råd ifm. med projektstyring.

**Martin Callesen** har 15 års erfaring som Java-udvikler. Derudover er han certificeret SCRUM-master og har tidligere arbejdet meget med optimering af udviklingsprocesser og miljøer. I TestHuset er han en del af afdelingen *Technology & Innovation*, der for tiden har fokus på DevOps og optimering af udviklingsprocesser. Martin er ekspert i Continuous Integration og Continuous Deployment. Martin har tilbudt at hjælpe med udarbejdning af Unit Tests, kravnedbrydning og kodning.

**Helge Nymand** er, efter eget udsagn, generalist. Han er oprindeligt uddannet arkitekt men har arbejdet med IT, og især software test, i over 10 år. Han er ekspert i Continuous Integration og Continuous Deployment og har stor viden indenfor alt hvad der har med softwareudvikling at gøre. Helge har tilbudt at hjælpe med udarbejdning af Unit Tests, opsætning af miljø og kodning.

### 2.6.2 Tekniske ressourcer

**Microsoft Visual Studio** Vi vil programmere i Visual Studio 2015 - Community Edition, da det er den IDE vi har mest erfaring med og tilmed også er tæt integreret med de øvrige værktøjer vi vil anvende. Visual Studio har en masse funktionaliteter, der vil være brugbare i forbindelse med vores projekt. Man kan afvikle unit tests, kontrollere kodedækning, der er snap-ins til Microsoft Team Foundation Server (forklaret senere) og så har den generelt en masse hjælpeværktøjer, der gør det nemt at skabe og udvikle applikationer i C# og .NET. Som udgangspunkt vil vi klare os med den gratis Community-udgave, da vi ikke har fundet noget behov for en af de betalte udgaver, som vi har adgang til igennem skolen. Så hvis vi opdager et behov kan vi altid opgradere i fremtiden.

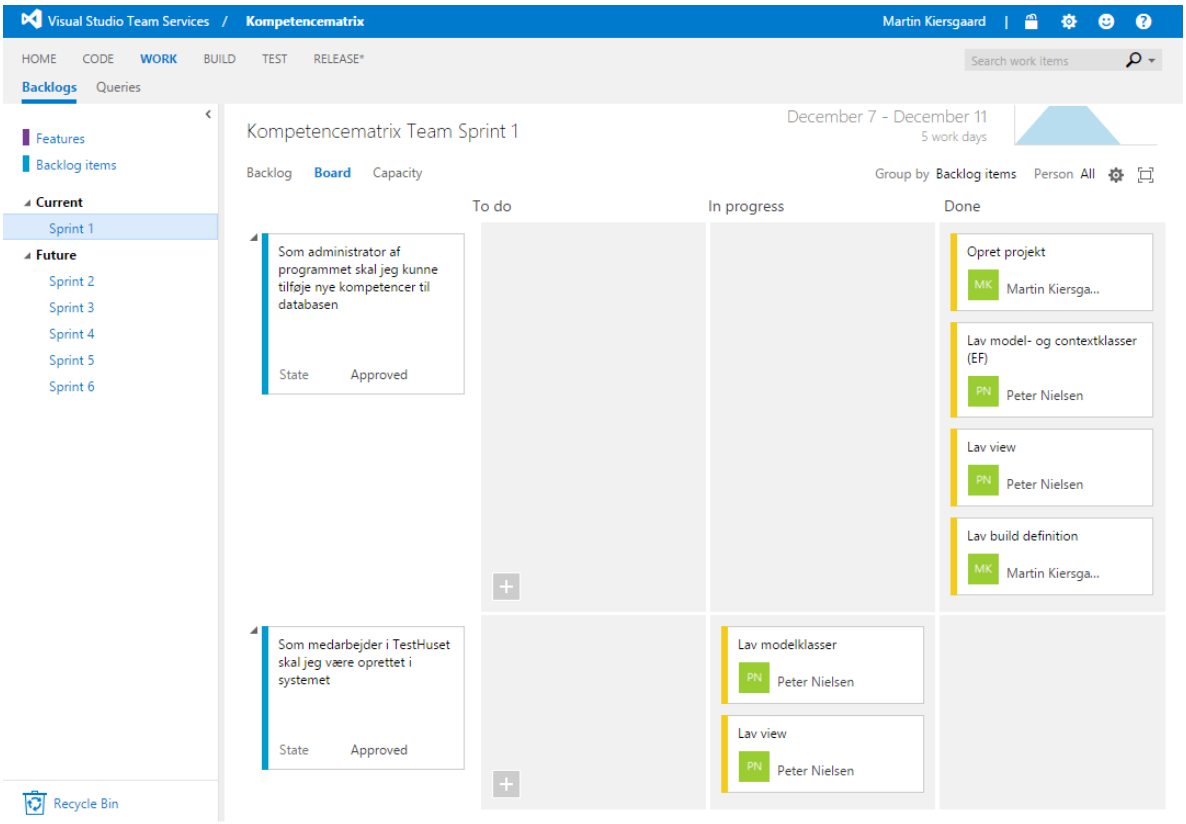
**Microsoft Azure** TestHuset har givet os tilladelse til at anvende deres Microsoft Azure-platform, til at hoste applikationen vi vil udvikle. Microsoft Azure er Microsofts Cloud-platform, hvor man kan få hostet, stort set, alle Microsofts produkter for en pris der bliver afregnet ned til minuttet. Det er muligt at få hosted Databaser, Virtuelle Servere, Web-Applikationer og meget andet. Microsoft har stillet en del funktionalitet til rådighed, uden beregning. Dette vil vi benytte os af i det omfang vi kan. TestHuset har dog tilbudt at betale, hvis vi får behov for noget funktionalitet der koster penge.

**Visual Studio Online** Vi har fået stillet TestHusets Visual Studio Online-løsning ( *VSO*) til rådighed. Her har vi fået oprettet vores eget projekt vi kan anvende til projektstyring, versionsstyring og automatisering af byg. VSO er en hosted Microsoft Team Foundation Server (TFS), der ligger i skyen hos Microsoft. Den er tæt integreret med Microsofts Sky-platform; Azure.

Man kan, i Visual Studio Online, oprette sine User Stories og skabe sine egne arbejdsgange for disse, således at de er tilpasset organisationens måde at arbejde på. Disse User Stories kan så tildeles et sprint og man kan få vist alle sine stories på et Kanban-board, for bedre overskuelighed.

På figur 1 har vi brugt vores Kanban-board for Sprint 1 som et eksempel. Der er tildelt to User Stories (I VSO kaldet Backlog Items) til Sprint 1 og vi har her oprettet nogle Tasks under hver Story. Når man påbegynder arbejdet på en Task starter man med at tildele Tasken til sig selv og flytte den til kolonnen *In progress*. Når man er færdig med opgaven, rykker man ligeledes Tasken til kolonnen *Done*.

Visual Studio Online indeholder også vores Git-repository og en byggeserver. Git er et såkaldt Distributed Source Control Management-system (Distributed SCM). Et SCM-system bruges til at versionere kildekode til softwareløsninger. Hver gang man foretager en ændring i sin kode, uploader (*Committer*) man hele ændringen til projektets *repository* i SCM-systemet, der så sørger for at holde styr på alle de ændringer der har været foretaget. På den måde kan man altid gå tilbage gennem versioner, hvis man skulle have behov for det. Det er også muligt at tage en kopi af en version (kaldet *Branching*) og arbejde med den parralelt af "hovedsporet" (*Master-Branch*). På den måde




Figur 1: Skærmprint af vores Kanban-board for Sprint 1

kan flere udviklere arbejde på hver deres funktionalitet, med fuld versioneringsunderstøttelse. Når udvikleren så har færddiggjort sin opgave, kan han splejse (*Merge*) sin Branch med Master-branchen og funktionaliteten er nu en del af hovedproduktet.

Git er et såkaldt *distribueret* SCM-system. Det betyder at alle udviklere sidder med en lokal kopi af det centrale repository og derfor også, udover at committe deres kode til deres lokale repository, skal synkronisere (*Push* og *Pull*) det lokale repository med det centrale på Git-serveren.

Byggeserveren i Visual Studio Online kan konfigureres til at hente kildekoden fra et SCM-repository, kompilere koden, køre Unit Tests og deploye applikationen til en server. Serveren kan konfigureres til at bygge enten på fastsatte tidspunkter eller hver gang der committes til en eller flere bestemte Branches. Man konfigurerer en *Build Definition*, hvori man angiver indstillingerne for det/de automatiske byg man ønsker at køre. Til at begynde med vælger man hvilke(n) Branche(s) man vil have sin Build Definition til at arbejde ud fra. Derefter skal man angive de opgaver der skal udføres af byggeserveren. Som regel vælger man at koden skal kompileres og at der skal køres Unit Tests. Hvis man ønsker at der automatisk skal deployes til en applikationsserver angives det også som et trin i sin Build Definition.

Visual Studio Online, sammen med Visual Studio og Azure, indeholder dermed alle de funktionaliteter og værktøjer vi har behov for, for at kunne besvare vores problemformulering.

# 3 Udviklingsmetode

Inden vi går i gang med projektet, er det vigtigt at vælge den rigtige udviklingsmetode. Derfor vil vi i følgende afsnit kigge på de forskellige udviklingsmetoder, som vi er blevet introduceret for og ud fra de enkelte metoder, vælge elementer der understøtter de krav vi stiller til metoden.

## 3.1 Krav til udviklingsmetode

Først vil vi definere de krav vi finder nødvendige for den metode vi vil tage i brug gennem projektet.

- Vi skal relativt hurtigt kunne præsentere et demonstrerbart produkt, således vi kan benytte os af *Continuous Integration* og *Continuous Deployment*
- Metoden skal understøtte brug af iterationer, da vi ønsker at efterkomme ønsker fra samarbejdspartner, men også for at kunne udgive regelmæssige releases
- Det er vigtigt for os, at have et møde med partner, således vi kan få feedback på det arbejde vi har lavet

## 3.2 Overvejelse om valg

Da det er vigtigt for os, at udvikle over iterationer, således at det er nemmere for os, at fremvise nye implementationer, har vi valgt at se bort fra sekventielle udviklingsmetoder og kun gøre overvejelser omkring de iterative modeller. Dette er de følgende udviklingsmetoder vi skal gøre os overvejelser om:

- Unified Process
- Extreme Programming
- Spiral modellen

Grunden til vi har valgt at overveje disse er, at de understøtter de krav vi har til udviklingsmetoden. De er alle iterative, dog er de stadig forskellige, idet de har forskellige fremgangsmåder på f.eks. releasefrekvens. Vi vil nu kigge på fordele og ulemper på de fremstillede udviklingsmetoder og herefter gøre os et valg, og evt. bruge artefakter fra andre udviklingsmetoder, såfremt de har relevans for vores krav, eller de er med til at gøre problemer mere belyste.

**Unified Process**

+ Løsning af projektets risici forbundet med kundens skiftende krav
+ Gode dokumentations- og modeleringsværktøjer (UML)
– Rollefordeling der kræver man er ekspert på rolleområdet, hvilket gør det svært for mindre teams at udvikle software under denne metode
– Udviklingsprocessen er kompliceret og fokuserer meget på organisation

**Spiral modellen**

+ Fokus på risiko analyse
+ Ekstra funktionaliteter kan implementeres løbende
– Er bedst beregnet til større projekter

**Extreme Programming**

+ Kundetilfredshed ved hurtig og kontinuerlig levering af fungerende software
+ Hyppig dialog mellem kunde og udviklere
+ Tilpasning til skiftende omstændigheder
– Kan let køres af sporet, hvis kunde ikke ved hvordan det endelige resultat skal være

Derudover har vi også taget et kig på *SCRUM* som projektstyring, da vi ønsker at have et overblik over de opgaver vi har i gang, men samtidig også manglende opgaver som ikke er påbegyndt.

+ Product-backlog til at holde styr på begyndte/manglende opgaver
+ Gode discipliner til løbende selvforbedring i teamet (Retrospectives)
+ Understøtter agile udviklingsmetoder med korte sprint-releases

## 3.3 Diskussion omkring valg

Det er vigtigt for os vores metode, at vi kommer til at arbejde i iterationer og derved hurtig levere et demonstrerbart produkt og fungerende software til vores kunde. Da tiden for projektet er kort, vil det også betyde, at vores iterationer bliver korte.

**Unified Process** Fokuserer på risikoanalyse, analyse og design løbende, hvilket kan være nyttigt til nogle projekter, men vi ser det ikke som nødvendigt, da vores projekt ikke virker abstrakt og uoverkommeligt. Derudover understøtter det ikke ændrende krav fra kunden, lige så godt som andre metoder, som gør det sværere at efterkomme ændringer i projektet og håndtere disse. *Unified Process* understøtter ikke udgivelse af tidlige prototyper, og iterationerne kan blive lange, netop pga. dokumentationen som skal foretages inden da. For at undgå lange iterationer, bliver man derfor nødt til at skære ned på artefakterne i denne metodologi. Derfor vil andre udviklingsmetoder være et bedre alternativ. Det kan dog være at det bliver nødvendigt at anvende UML diagrammer undervejs, til at forklare og formidle hvordan systemet fungerer.

**Spiral modellen** Understøtter kravet om brug af iterationer. Spiralmodellen Fokuserer meget på, at man i starten foretager analyse af risiko og er derfor mere egnet til udvikling af systemer, hvor kunden ikke rigtig ved hvad der ønskes og hvor der løbende tilpasses derefter. Selvfølgelig skal der foretages analyse og vurdering af systemet, men vi mener, at det ikke burde fylde for meget i vores udviklingsproces, da vi ikke finder det relevant i forhold den den definerede opgave. Vi vil selvfølgelig gøre brug af risikoanalyse eller andre spiralmodel-discipliner, hvis vi senere finder ud af at det kan være til gavn, Men vi ser andre metoder som bedre alternativer, end denne.

**Extreme Programming** Understøtter størstedelen af de krav vi har fremstillet til vores brug af udviklingsmetode. I XP har man forkus på at kunne levere fungerende software, hurtigt. XP er dog nyt for os, men vi er blevet introduceret til mange disciplinerne på 4. semester, da vi blev undervist i agil udvikling. Da XP er en agil arbejdsmetode, er dette jo egentlig ikke overraskende. Især det faktum at Continuous Integration og Continuous Deployment er discipliner der specifikt er nævnt som regler for Extreme Programming[2] taler kraftigt for at det er denne metode vi skal fokusere på.

[2] *The Rules of Extreme Programming.* URL: `http://www.extremeprogramming.org/rules.html`.

**SCRUM** er en projektstyringsmetode der er tilpasset til brug med agile udviklingsmetoder. *SCRUM* fokuserer på at få det bedste ud af et team. Måden det gøres på er ved, at man hver dag har et dagligt møde med sit team. Her diskuteres planerne for dagen, samt eventuelle problemer medlemmer måtte være løbet ind i. På dette grundlag kan medlemmer i teamet være med til at løse dette. Igen handler det om, at kunne tilpasse sig ændringer. Udover *product backlog*, der er den overordnede mængde opgaver for projektet, har man også en *sprint backlog* som indeholder de user stories man har bestemt skal være færdige for det pågældende sprint/iteration. Inden hvert sprint har man et sprint planlægningsmøde hvor man bryder disse user stories ned i mindre opgaver/tasks. Derudover har man for hver slutning af sit sprint, et møde med product owner hvor man overleverer produktet og får feedback på sin fungerende software. Herefter holder teamet internt et møde, om hvordan arbejdsprocessen har været, og om det har været fyldestgørende det man har fået ud af arbejdet, og hvad der kan forbedres.

## 3.4 Konklusion

Ud fra de krav vi havde til den udviklingsmetode som vi ville tage i brug, eller hvert fald bruge nogle ideologier fra, kan vi konkludere, at den der lægger sig mest op til kravene er, *Extreme Programming*. Dette er på baggrund af de praktikker som indgår i *Extreme Programming*. Heriblandt *pair-programming* og at man i XP gør brug af netop det vi vil undersøge i projektet her, *CI/CD*. Derudover har vi også haft taget et kig på fordele og ulemper for de udvalgte metoder. Herpå kiggede vi, hvorvidt de passede på projektets størrelse, men også i forhold til at komme i gang med udviklingen af det stykke software, som vi skal have produceret. Det gjorde sig også gældende, at det vi vil undersøge i forbindelse med opgaven her, bliver benyttet i udviklingsmetoden. Man kan dog argumentere for, at vi kunne tilpasse en metode til vores projekt af brudstykker fra andre metoder, så hvis vi undervejs finder ud af at vi kan få gavn af at anvende praktikker fra andre metoder, vil vi selvfølgelig gøre det. Vi har ydermere valgt, at benytte os af *SCRUM* med undtagelser, som projektstyringsmetode da det er med til, at skabe overblik over projektet og dets opgaver. Samtidig er der inden for *SCRUM* stor fokus på at man evaluerer sig selv hele tiden og sørger for at tage fat i problemer, gennem *retrospectives*. Dette vil vi benyttes os af, da vi også mener at det burde kunne tvinge os til at reflektere mere over måden vi arbejder på undervejs. Sammen med vores Product Owner vil vi også afholde *Sprint Planning Meetings* og *Sprint Reviews*, hvor vi dels planlægger hvad der skal laves i vores sprint, og løbende overleverer det produkt vi har udviklet. Vi vil dog styre uden om nogle af de roller og praktikker, som der findes indenfor *SCRUM*, heriblandt *SCRUM master* og daglige *SCRUM* møder. Årsagen er at vi ikke føler at der er behov for en *SCRUM master* i et team med kun to medlemmer. Og de daglige stand-up møder vil ikke være relevante, da vi ikke mødes hver dag. Igen er et to-mands team også for småt til at det er nødvendigt at holde relativt formelle statusmøder. Vi vil jævnligt have kontakt til hinanden, så eventuelle problemer, set ud fra projektets størrelse, burde være til at forholde sig til. På figur 1 ses det kanban-board vi har arbejdet ud fra, i vores første sprint. Boardet findes i vores Visual Studio Online projekt, der også indeholder mange af de andre værktøjer vi skal bruge til styring såvel som udvikling af projektet.

# 4 Continuous Integration og Continuous Deployment

## 4.1 Praktikker

Både *Continuous Integration* (CI) og *Continuous Deployment* (CD) er praktikker der stammer fra *Extreme Programming* (XP). Meget af vores research er baseret på Martin Fowlers tanker omkring emnet.[3] Continuous Integration er, kort fortalt, en arbejdsmetode udviklet som en naturlig udvidelse af den agile tilgang til sytemudvikling, ved at man som udvikler, hele tiden arbejder i små iterationer og committer sin kode en eller flere gange om dagen. Hele Continuous Integration-praktikken er funderet i udviklingsteamets *Source Code Management*-system (SCM), som f.eks. Git. Kort fortalt er Git et repository hvor man opbevarer sin kode og alle andre kodeartefakter der er nødvendige for at kunne bygge applikationen, man er ved at udvikle. Hver gang man har tilføjet kode til projektet, tilføjer (committer) man koden til sit repository og man kan til enhver tid gå tilbage igennem commits og på den måde gendanne tidligere versioner af applikationen.

En essentiel funktion i SCM-systemer er at man kan arbejde i spor eller branches, hvor man som udvikler kan tage en kopi af applikationen og arbejde parralelt med sin egen kopi. På den måde er det muligt for flere udviklere at arbejde på samme applikation eller for separate teams at arbejde på hver sin feature til samme applikation. Når man så har kodet sin feature eller den user story man skulle, fletter (merger) man tilbage til sit hovedspor (main-branch) og alle har dermed igen samme udgangspunkt at kode ud fra.

For at opsummere hovedtrækkene bag Continuous Integration er her en liste over de punkter vi har fundet frem skulle være de essentielle praktikker:

**Commit til hovedbrachen minimum en og gerne flere gange om dagen** Tanken bag Continuous Integration er at man, ved at arbejde i små branches i sit SCM-system (SCM) og jævnligt integrerer med sin main-branch, slipper for at bruge unødigt meget tid på at løse integrationskonflikter eller merge-conflicts. Det gør integrationen hurtigere at udføre og meget nemmere at overskue.

**Lav Unit Tests** En af grundstenene i CI er, at man koder unit tests. Al kode der bliver committet skal være selvtestende, således at man løbende kan holde et overblik over kvaliteten af koden.

**Hav en integrationsserver der bygger og kører Unit Tests ved hvert commit til hovedbranchen** En integrationsserver er en service der løbende holder øje med et SCM-repository og når der er nye commits, udfører den en række handlinger med koden. Det første serveren skal udføre, når der bliver committet ny kode, er at kompilere/bygge koden. Hvis koden ikke en gang kan bygges, så er der ingen grund til at fortsætte med de næste trin, da comittet reelt set er invalidt. Når serveren har kontrolleret at koden kan bygges, uden fejl, så skal den eksekvere de Unit Tests der er i softwareprojektet. Teorien er at projektet på den måde bliver selvtestende og da udviklerne har brugt tid på at gøre dette, kan man fange en del fejl som måske først ville være blevet fanget senere i udviklingsforløbet. I værste tilfælde bliver en fejl først fundet når applikationen er i produktion.

**Implementer Continuous Deployment** Ifølge Martin Fowler er det vigtigt at man kan deploye til drift-/testmiljøer automatisk. Denne praksis hedder Continuous Deployment og er en praktik der hører til, under Continuous Integration. Ifølge Helge Nymand hos Testhuset, er et af de problemer der bliver løst med Continuous Deployment, at man hurtigt kan få etableret en udgave af applikationen i et test-/staging-miljø, hvor f.eks. testere kan komme til at udføre deres testcases. Det er her at tankegangen bag *DevOps* bliver relevant. At man ikke som udvikler er afhængig af en IT-driftsafdeling

[3] Martin Fowler. *Continuous Integration*. URL: `http://www.martinfowler.com/articles/continuousIntegration.html`.

der skal leverere et produkt/miljø, men i stedet samarbejder om at skabe en et miljø der drifter sig selv, således at opsætning af miljøer sker automatisk. Dette er der blevet større og større muligheder for at opnå, med virtualisering og især med de nye sky-tjenester som Amazon Web Services (AWS) og Microsoft Azure. Med disse tjenester kan man kan have et, i praksis, ubegrænset antal miljøer der, når de skal bruges, kan startes op på kort tid og lukkes ned igen når de ikke længere skal bruges. Vi vil i vores projekt anvende Microsoft Azure, til at hoste vores applikation. Ud over at kunne deploye til andre miljøer er det også en god praksis at man altid kan finde den seneste version af applikationen, enten som en eksekverbar fil eller kørende på en applikationsserver der svarer til driftsmiljøet.

**Ødelagte byg har topprioritet i hele teamet** Hvis et automatiseret byg af hoved-branchen fejler, så er tanken at topprioriteten for hele teamet bliver at løse fejlen. Dette er ikke nødvendigvis at en Unit Test fejler, men kan være det. Hvis man anvender Git, hvor udvikleren har en lokal kopi af det fælles repository, og sørger for at foretage byg og køre sine Unit Tests på sin udviklermaskine, før man commiter til sit hoved-repository, vil risikoen dog være mindsket betydeligt. Vi vil gennemgå hvordan Git fungerer, senere i denne opgave. Da et fejlende byg i SCM-systemet blokerer for al yderligere udvikling er det derfor også naturligt at fejlrettelse bliver topprioritet for alle i teamet. Hvis alt andet glipper, så kan man altid gå tilbage til en tidligere version, der fungerer. I nogle tilfælde vil det bedst kunne betale sig. Men dette er en vurdering der skal foretages i teamet, sammen med PO. En måde, at håndtere et fejlende byg på er, at fastsætte en bestemt mængde tid til fejlretning og hvis fejlen ikke er rettet inden da, så reverter man til et tidligere byg og starter forfra. Da man ofte committer sin kode, vil det tabte arbejde dermed være overskueligt.[4]

Figur 2: Build light *Kilde: pinterest.com*

**Ingen committer et ødelagt byg til hovedbranchen** En vigtig regel er, at man ikke committer kode der ikke kan bygge. Der må aldrig committes kode til hovedsporet, som ikke kan kompilere, da dette vil ødelægge bygget.

**Ingen committer kode uden tilhørende Unit Tests** En anden vigtig regel er, at man committer ikke kode uden en tilhørende Unit Tests. Det er essentielt for at leve op til reglen omkring selvtestende kode.

**Sørg for at hele teamet er bevidste om byggets helbred** Feedback er en vigtig parameter, som CI hjælper med at give. I og med at de automatiske byg skal køre hver gang en udvikler integrerer med hovedsporet, så har man brug for en måde at notificere hele teamet, hvis et byg fejler. Hver gang der opdages en fejl, hvad enten det er et byg eller en Unit Test der fejler, så ved man det som udvikler, team leder, PO eller øvrig interessent med det samme og kan tage hånd om problemet. For at det kan lade sig gøre er man nødt til at have en måde at monitorere byggets tilstand. Nogle opsætter tv-skærme med byggets status eller laver andre kreative måder at holde øje med bygget på. (Se figur 2) Hovedtanken er at hele teamet er bevidste om byggets helbred og kan handle, hvis der opstår fejl.

[4]buildout.coredev. *Essential Continuous Integration Practices.* URL: `http://buildoutcoredev.readthedocs.org/en/latest/continous-integration.html`.

## 4.2 Værktøjer

Der findes et udvalg af forskellig software der tilbyder en integrationsservice, så vi spurgte Helge Nymand hos Teshuset, hvilke forskellige værktøjer der er på markedet. De mest populære er:

- **Hudson/Jenkins** er den mest udbredte, gratis, open source og skulle være meget alsidig.
- **Atlassian Bamboo** Atlassian har, blandt andet, lavet projektsstyringsværktøjet JIRA og SCM-systemet Bitbucket. Bamboo er ikke gratis, men integrerer rigtig godt sammen med Atlassians øvrige systemer.
- **Jetbrains Teamcity** er en meget rost og temmelig udbredt intergrationsløsning, lavet af samme firma der også har lavet ReSharper til Visual Studio og IntelliJ IDE'en.
- **Microsoft Team Foundation Server** er heller ikke gratis, men er rigtig godt integreret med Microsoft Visual Studio. Der findes en gratis udgave af TFS i Microsofts sky, Azure, som er den byggeserver vi vil anvende til vores projekt.

# 5 Sprint 0: Projektetablering

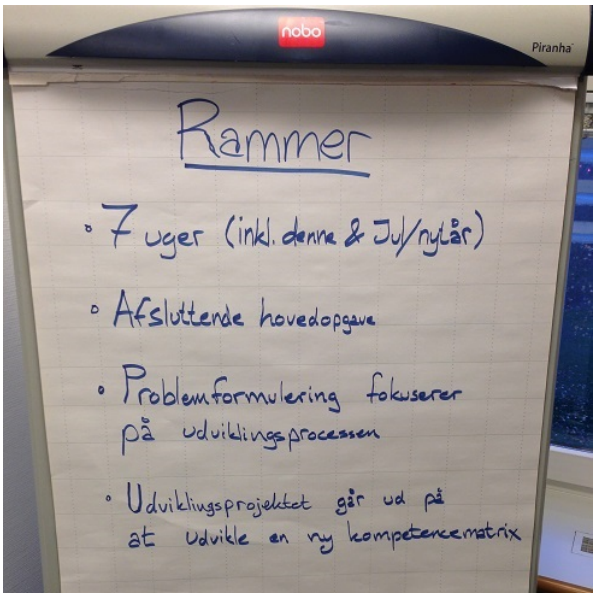
## 5.1 Krav-workshop (16. november 2015)

Vi afholdte et møde med interessenterne for applikationen. Formålet med mødet var at kortlægge visionen og formålet med applikationen, samt udpege en Product Owner fra virksomheden.

Mødet begyndte ved at vi bød velkommen og gav en introduktion til vores opgave, så alle var med på rammerne for projektet. Da vi har meget begrænset tid at arbejde i, så var det vigtigt for os at gøre det klart, at vores opgave kom til at fokusere på udviklingsprocessen og ikke så meget på selve produktet. Derfor vil vi formodentlig ikke komme til at få implementeret alle de funktioner som der skulle være ønsket. På figur 3 kan man se den tavle vi havde forberedt til gennemgang af vores rammer for projektet. På Tabel 1 er deltagerne på mødet listet.

| Navn: | Stilling |
|---|---|
| Allan Tange | CEO |
| Martin Skovgaard | COO |
| Jesper Molin | Salgschef |
| Stine Osted | Service Manager |
| Enzo Henriksen | Team Lead |
| Søren Lyhne Andersen | Konsulent |
| Lise Pedersen | Konsulent |
| Peter Nielsen | |
| Martin Kiersgaard | |

Tabel 1: Deltagere på mødet




Figur 3: Tavle med oversigt over rammerne for projektet

**Behovsafdækning** I dag har TestHuset et Excel-ark, omtalt som Kompetencematrixen, der indeholder 9 faner med alle konsulenter og forskellige kategorier af kompetencer. Der er flere problemer med denne løsning, der ville kunne afhjælpes med en applikation der var udviklet til formålet. De 9 faner repræsenterer hver en kategori af kompetencer, herunder: *Testfagligt, Testværktøjer, Teknologier og værktøjer, Modeller og Metoder, Samarbejde og Ledelse, Forretningsviden, Personlige, Uddannelsesniveau og Certificeringer.* På hver fane er Y-aksen befolket af de forskellige kompetencer indenfor den pågældende kategori og X-aksen lister alle konsulenter der er ansat. Se figur 4.

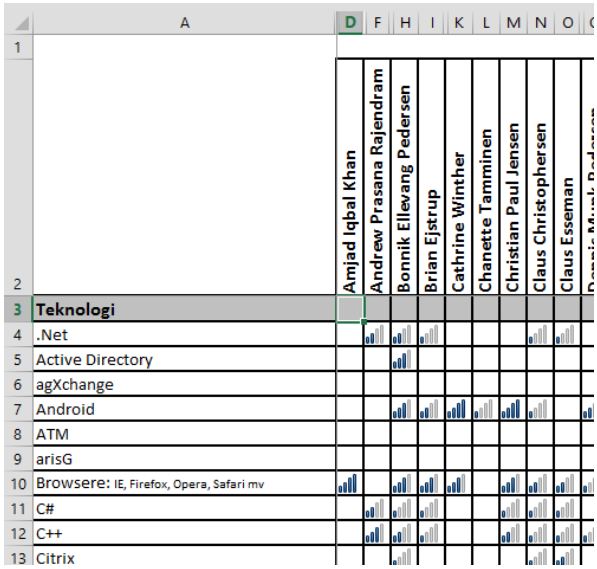
Hver konsulent skal så finde sit navn i matrixen og udfylde de kompetencer vedkommende besidder. Der bliver udfyldt med en værdi fra 1 til 4, alt efter hvor meget erfaring konsulenten har med denne kompetence. Dette gøres i hver enkelt af de 9 faner i arket. Arket gemmes herefter og er dermed opdateret med konsulentens kompetencer.

Man har nu et overblik over kompetencerne alle konsulenter i virksomheden besidder. Dette kan så anvendes af diverse medarbejdere i organisationen. Det kan være i en salgssituation, hvor der er et kundebehov for en konsulent med bestemte kompetencer. Her kan sælgeren slå op i kompetencematrixen og se hvilke konsulenter der besidder den pågældende kompetence og også i hvilken grad. Det

kan også være en konsulent der har udfordringer med en bestemt teknologi og har brug for hjælp fra en kollega. Man kan så slå op i Kompetencematrixen og se hvilke kollegaer man kan ringe til.

**Problemstilling** Kompetencematrixen ligger delt på et sharepoint-site som alle i organisationen har adgang til. Men kan dog ikke rette i arket, hvis en anden person har det åbent. Vedligeholdelse af arket er besværligt, især når der skal tilføjes eller fjernes medarbejdere, da alle 9 faner skal låses op og låses igen, efter opdatering. Derudover er det relativt uoverskueligt at søge efter specifikke kompetencer.

Der var også en del ønsker til nye funktioner i en ny kompetencestyringsapplikation. Især søgefunktioner og andre muligheder for at danne sig overblik, var savnet i matrixens nuværende udformning. Især muligheden for at søge kvantitativt på både kompetencer og medarbejdere, samt udføre gap-analyser på definerede kompetencer og medarbejdere, var noget der var ønsket af dem der til dagligt arbejder med salg og HR.




Figur 4: Udklip af kompetencematrixens faneblad *Teknologier og Værktøjer*

**Konklusionen på mødet** Vi talte om mange funktioner der kunne være meget smarte at få med i applikationen, men vi var nødt til at styre mødet og stoppede det efter en times tid. Der blev bl.a. foreslået at lave en kundeportal, hvor kunder kunne logge på og se igennem konsulenternes kompetencer. Endnu en gang måtte vi huske interessenterne på, at projektet har en skarp deadline om 7 uger og at det derfor ville være spild af tid at diskutere for mange funktioner, da det ikke er realistisk at de bliver implementeret. Vi gik ud fra mødet med en rigtig god idé om hvad interessenterne havde behov for. Vi fik også udpeget en Product Owner. Det blev COO Martin Skovgaard.

Vi talte efter mødet kort med vores PO og vi blev enige om, at næste skridt var, at vi sammen med vores PO tager et møde, hvor vi får udformet nogle user stories og planlagt vores første sprint.

## 5.2 Planlægningsmøde for det kommende sprint 1 (3. december 2015)

Vi afholdte et møde med vores Product Owner, for at udvælge *User Stories* til vores første sprint. Vores oprindelige Product Owner, Martin Skovgaard, var desværre blevet langtidssygemeldt siden vores krav-workshop, så Stine Osted var så venlig at træde til og tage rollen som vores nye Product Owner.

| Navn: | Stilling |
|---|---|
| Stine Osted | Service Manager |
| Peter Nielsen | |
| Martin Kiersgaard | |

Tabel 2: Deltagere på mødet

Formålet med mødet var at udvælge de User Stories der skal udvikles på vores første sprint og fastsætte datoer for sprintets start og slut.

Vi gik ind til mødet med nogle User Stories vi havde forberedt, efter bedste evne, ud fra diskussionen vi havde med interessenterne på det foregående møde. Der var ikke tænkt alt for meget i avancerede funktioner, men i højere grad fokuseret på at få defineret nogle User Stories for de grundlæggende funktioner i programmet. Disse blev taget godt imod, med nogle få kommentarer til definitioner og navngivning. Vi havde en User Story (Backlog item 29), der havde ordlyden: *Som administrator skal jeg kunne tilføje nye brugere til databasen.* Der blev stillet spørgsmålstegn ved hvad der menes med brugere og om formuleringen ikke begrænsede mulighederne for tilføjelse af brugere, til at være en manuel operation. PO mente at man måske skulle formulere US'en således, at man gav mulighed for lidt mere kreativitet. Derfor endte den med at lyde *Som medarbejder i TestHuset skal jeg være oprettet i systemet.* Med denne formulering bliver skal man udvikle en måde for brugeren at være oprettet i systemet på. Men tilføjelsen af flere brugere er ikke defineret endnu. Dette kan så defineres i endnu en User Story, afhængigt af hvordan man ønsker at nye brugere skal tilføjes. Det kunne eksempelvis være at brugere skal indhentes fra et Active Directory.

Vi endte dermed op med med at have i alt 6 User Stories som vi blev enige var rigeligt til at begynde med. De 6 user stories er angivet herunder.

- **Backlog Item 16** Som administrator af programmet skal jeg kunne tilføje nye kompetencer til databasen
- **Backlog Item 29** Som medarbejder i TestHuset skal jeg være oprettet i systemet
- **Backlog Item 17** Som bruger af programmet skal jeg kun kunne angive mine egne kompetencer
- **Backlog Item 31** Som bruger af programmet skal jeg kunne søge efter alle medarbejdere med en bestemt kompetence
- **Backlog Item 30** Som bruger af programmet skal jeg kunne foreslå tilføjelse af nye kompetencer til databasen
- **Backlog Item 18** Som udvikler skal jeg have et visual studio projekt at udvikle programmet i

Vi blev enige om at *Backlog Item 18* ikke kommer til at indgå som en decideret User Story, men bliver lukket og er i stedet noget vi kommer til at klargøre, inden Sprint 1 begynder. Til sprint 1 vedtog vi at Item 16 og 29 bliver de to User Stories vi begynder at udvikle.

# 6 Sprint 1

I vores første sprint skal vi have oprettet en infrastruktur til projektet og have udviklet de to User Stories, der var aftalt med vores Product Owner. Undervejs vil vi afprøve nogle af de praktikker vi har fundet, der understøtter Continuous Integration og Continuous Deployment.

## 6.1 Arkitektur

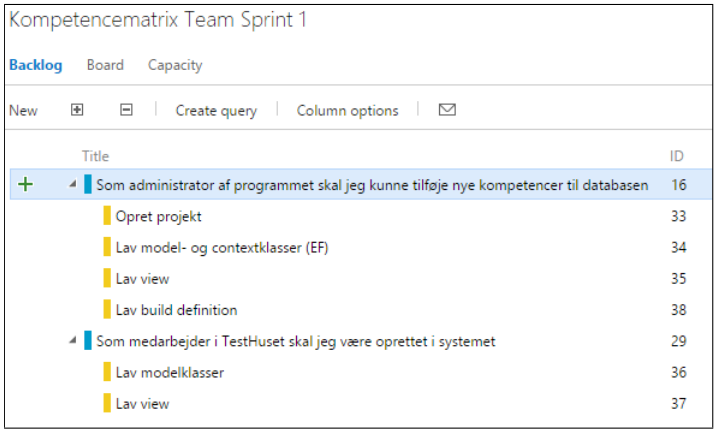
Vi har tænkt os at udvikle applikationen som en web-applikation, der er hosted hos Microsoft Azure. Efter at have undersøgt hvilke frameworks der findes og er "tried-and-tested", er vi nået frem til at vi vil anvende *ASP.NET MVC*-frameworket sammen med *ASP.NET Entity-Framework*.

MVC er et framework der gør det meget enkelt at lave en web-applikation, da den på forhånd laver et komplet Model-View-Control skelet man kan arbejde ud fra. Man kan generere Views ud fra model-klasser, komplet med formularer til oprettelse, redigering og sletning af elementer. Derudover er det stillet op i et pænt og dynamisk design, hvor det er enkelt at tilpasse udseendet gennem CSS og markupsproget Razor.

Entity Framework er et framework der håndterer databaseforbindelser, -forespørgsler og -design, på en meget overskuelig måde. Man kan, ud fra sine modelklasser og en database-context klasse, få frameworket til at generere tabellerne i databasen, komplet med Foreign Keys, uden selv at skulle beskæftige sig med SQL-queries eller åbning og lukning af databaseforbindelser.

## 6.2 Nedbrydning af User Stories

Vi har to User Stories der skal nedbrydes i Tasks. Den første User Story vi vil udvikle, er Backlog Item 16. Vær opmærksom på at User Story og Backlog Item er det samme, Backlog Item er blot Visual Studio Onlines navn for en User Story. Vi besluttede, at oprettelsen af Visual Studio projektet blev oprettet som en Task under Backlog Item 16. Ligeledes med oprettelsen af vores *Build Definition*, der skal drive vores Continuous Integration-byg; denne opgave er også oprettet som en Task under Backlog Item 16.




Figur 5: *Backlog* for *Sprint 1*

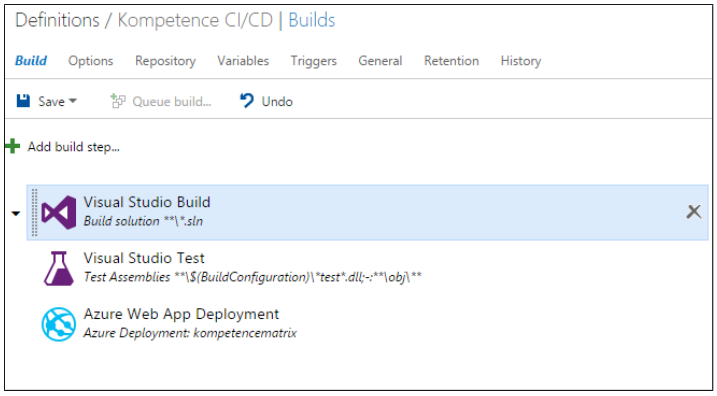
På figur 5 kan man se Backloggen for vores sprint 1. Tasks (gule) er sorteret under Users Stories (blå). Vi har besluttet ikke at forsøge at estimere for stramt på hvor lang tid de enkelte Tasks vil tage, da vi ikke har noget grundlag for at foretage estimeringen og vil søge for at vores Continuous Integration proces udføres rigtigt. Vi vil benytte nogle frameworks vi ikke har anvendt før og derfor tænker vi at det er for usikkert at fremsige estimater. Derudover har vi også en automatisk bygge- og deployment-proces der skal sættes op og værktøjer der skal konfigureres.

### 6.2.1 Backlog Item 16

**Task 33 - Opret projekt** Den første task vi kastede os over var at oprette projektet. Vi havde allerede oprettet et projekt i Visual Studio Online, der var vores domæne på VSO-platformen. Den indeholder alt hvad vi skal bruge til at drive projektet; User Stories, Git-repository, byggeserver, Kanban-board og meget mere. Derefter oprettede vi vores Visual Studio Solution og committede den til versionsstyringen (Git). Vores Git-repository er en central del af at kunne praktisere Continuous Integration, da det er her vi har al vores kode og øvrige artefakter til projektet opbevaret, hvor vi begge altid kan komme til dem.

Vi oprettede vores solution ud fra ASP.NET MVC-skabelonen, der automatisk opretter en grundlæggende MVC struktur, med tre standard HTML-sider: Home (forside), About og Contact. På disse er der noget standard tekst, der kan slettes eller ændres som man vil. Vi bad også Visual Studio om, samtidig, at oprette et Test-projekt der skal indeholde vores automatiske Unit Tests. Vi havde nu vores første commit med den grundlæggende platform at kode på.

**Task 38 - Lav Build Definition** I Visual Studio Online konfigurerer man de trin der skal gennemføres i en byggeproces. Man får derved en Pipeline for hele bygge-, test og deployment-processen. Vi valgte en skabelon for et standard visual studio projekt, der også skulle deployes til Azure. På figur 6 kan man se trinene som de ser ud i Visual Studio Online.




Figur 6: Oversigt over triene i vores *Build Definition*

**Trin 1 - Build** Det første trin er det der bygger (kompilerer) koden der er blevet committet. I vores tilfælde er der valgt en task der er tilpasset til at kompilere Visual Studio-projekter, vha. MSBuild, der er compileren der bruges til .NET-understøttede sprog. Man kan sagtens anvende andre compilere, hvis man ikke koder i sprog der understøttes af MSBuild. Deriblandt Maven, Gradle, Xamarin, Android Build og flere andre. Dette er det første trin i pipelinen, da det ikke ville give mening af at køre de efterfølgende trin, hvis koden ikke engang kan bygge. Byggeprocessen er konfigureret til at stoppe, hvis trin 1 stopper. Dette kan dog ændres.

**Trin 2 - Test** Det næste trin er kørsel af automatiserede tests; i dette tilfælde Unit Tests fra det Visual Studio testprojekt vi oprettede. Også dette trin har vi konfigureret til at stoppe, hvis der findes fejl. I man kan dog vælge at lade alle Unit Tests køre, uden at en fejlende test vil blokere kørslen af resten. På den måde får man kontrolleret om der er flere tests der fejler. Det er også muligt at sætte Build Definitionen op således at der automatisk bliver oprettet et Bug-item, der bliver tilføjet til Projektets backlog, hvis en test fejler.

**Trin 3 - Deployment** Det tredje trin er en foruddefineret deployment-skabelon der deployer en web-applikation til Microsoft Azure. I denne konfigurerer man en sikker forbindelse fra Visual Studio online til sin Microsoft Azure konto. Denne er sikret med Azures egen certifikatbaserede godkendelse. Når forbindelse og authentication er sat op og man har angivet hvilken foruddefineret Azure Web App-site man vil deploye til, behøver man ikke at gøre mere. Visual Studio Online og Azure er så tæt integrerede at alt fungerer automatisk med disse få konfigurationer. Det er dog også muligt at deploye

til andre platforme. Det er muligt, at få byggeprocessen til at overføre applikationen til en Windows Maskine og eksekvere kommandoer med Microsofts egen kommandolinje; PowerShell. Man kan også sende kommandoer til Azure vha. Powershell. Det kunne f.eks være opstart af Virtuelle Maskiner der skulle bruges til at hoste applikationen.

Vores Build Definition bliver sat op til at deploye direkte til et websted hos Azure. Vi kommer til at bruge dette websted som *Staging*-miljø, da det ikke er her det ville komme til at skulle køre i produktion. Det vil kun blive brugt til demonstration af det produkt vi udvikler.

Det er dog ikke kun i forbindelse med deployment til produktion man kan anvende disse funktioner. Man kan også have sat andre automatiserede tests op, der ikke er unit tests. Man kunne teste integrationer mellem applikationen man var ved at udvikle og andre applikationer denne var afhængig af, men som man ikke tester for i unit testen.

**De øvrige tasks for dette Backlog Item:**

- **Task 34 - Lav model- og contextklasser (EF)**
- **Task 35 - Lav view**

### 6.2.2 Backlog Item 29

Vi har delt Backlog Item 29 op i to Tasks, da det, meget lig Backlog Item 16, er en enkel oprettelse i databasen ud fra vores modelklasser.

- **Task 36 - Lav model- og contextklasser (EF)**
- **Task 37 - Lav view**

## 6.3 Entity Framework

Som nævnt tidligere, har vi benyttet os af et framework kaldet Entity Framework. Dette framework gør, at vi ikke skal bekymre os om SQL-queries og håndtering af åbne forbindelser til databasen. Alt dette håndterer Entity Framework. Afsnittet her, vil dog ikke handle om hvordan Entity Framework virker, men mere hvordan vi har brugt det, i vores software.

| **Ordforklaringer** | |
|---|---|
| Entity | En tabel i databasen omhandler et objekt, person eller ting - disse er kendt som entiteter |

**Model-klasser** er de klasser der bruges til at holde den data vi henter fra databasen og som bruges igennem systemet. De er simple klasser, som kun indeholder properties. Sådanne klasser kaldes *POCO*-klasser. *POCO*-klasser er simple objekter, som ikke indeholder kompliceret logik og bruges typisk som *Data Transfer Objects*. Sådanne objekter indkapsler dataen i et objekt og sendes fra et sted i systemet til et andet. Et eksempel her kunne være, at bringe data fra ens service-lag til programmets GUI.

**DatabaseContext-klasse** Entity Framework introducerer, for at kunne interagere med model-klasser og database, en klasse som nedarves. Denne klasse kaldes *DbContext*. *DbContext* er en vigtig del af Entity Framework, da denne klasse er "broen" mellem database og ens model-klasser. Det vil altså sige, at *DbContext* skal ses som en repræsentation af ens database. Kort fortalt, tager *DbContext* sig af, at mappe data til ens model-klasser og håndtering af dette. Ydermere oversættes LINQ-queries til SQL-queries og *DbContext* tager sig også af CRUD (Create, Retrieve, Update og Delete) operationer på vegne af ens entiteter.

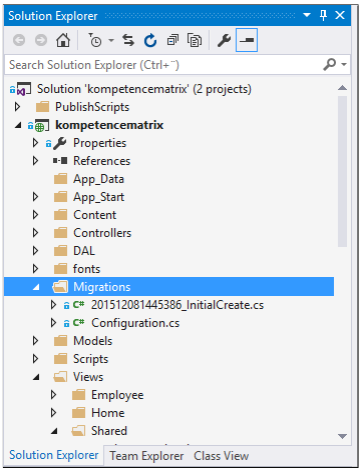
```
1       public class Employee
2       {
3           public int ID { get; set; }
4
5           public string FirstName { get; set; }
6
7           public string LastName { get; set; }
8       }
```

Kodeeksempel 1: Model-klasse for medarbejdere

```
1       public class SkillMatrixContext : DbContext
2       {
3           public SkillMatrixContext() : base("SkillMatrixContext")
4           {
5           }
6
7           public DbSet<Skill> Skills { get; set; }
8           public DbSet<Employee> Employees { get; set; }
9       }
```

Kodeeksempel 2: Vores SkillMatrixContext klasse efter sprint 1

I kodeeksempel 2 ses nedarvingen fra *DbContext* og hvordan vores context (en modellering af databasen) ser ud. I denne ses constructoren og to properties. Constructoren nedarver fra *DbContexts* constructor hvor denne tager en parameter. Parametren er ens *connectionString* som er defineret i ens `Web.config` (se kodeeksempel 16 som findes i bilag A). Dette sikrer, at når vi tester applikationen i vores stagingmiljø, kan vi tilgå databasen. De to properties (*Skills* og *Employees*) gør brug af Entity Frameworks `DbSet<TEntity>` klasse. Disse repræsenterer en tabel i ens database og derved alle entiteter der kan queries, af en given type. Når vi så gør brug af vores `SkillMatrixContext` i koden, kan vi tilgå databasen og derved udføre CRUD operationer på de "tabeller" vi har eksponeret. Dette gøres ved at instantiere vores `SkillMatrixContext` objekt og kalde et `DbSet` som findes i `SkillMatrixContext` og herfra kan der kaldes f.eks. `Add(TEntity)` som vil tilføje en given type til databasen, så snart `SaveChanges()` bliver kaldt. Et sådant eksempel findes i bilag A kodeeksempel 18.




Figur 7: *Migrations* mappen som den ses i Visual Studio

Da vi igennem sprint 1 implenterede de to user stories, valgte vi, at comitte på vores branch som tilhørte den user story vi var i gang med. Da vi nåede til den anden user story, måtte context-klassen ændres, og derved ændres vores databasemodel sig også. Dette medførte, at vi måtte migrere vores database over til en ny version. Heldigvis var det, på det plan vi arbejder på på nuværende tidspunkt, en nem sag. For at aktivere migrering af ens database, skal man igennem nogle forskellige trin. Fordelen ved at migrere ens database er, at man kan ændre i ens

databasemodel og deploye disse ændringer til ens *staging database* eller produktionsdatabase ved at opdatere ens databaseskema uden at man er nødt til, at droppe ens database og lave den på ny. Dette sikrer, at man ikke mister data. Måden man kommer i gang med at aktivere migrering i ens projekt gøres på følgende måde:

I Visual Studio skal man åbne *Package Manager Console*, hvilket gøres fra menuen **Tools**, **NuGet Package Manager** og derefter **Package Manager Console**. Dette åbner et lille konsol vindue i bunden af Visual Studio som illustreret på figur 21 (se bilag B). I denne konsol indtastes kommandoen *enable-migrations*. Dette vil lave en mappe i ens projekt, kaldet *Migrations* som det ses på figur 7. Denne mappe indeholer to filer, som er klasser. *Configuration*-klassen tillader os, at konfigurere hvordan vores migreringer opfører sig. Vi har valgt bare at bruge standard konfigurationen, men det er muligt ved hver migrering, at få udfyldt specifikke tabeller med fast data. Den anden klasse, *..._InitialCreate*, er en klasse som er blevet oprettet efter vi udførte den første migrering. Koden i denne klasse repræsenterer objekter som allerede er/var kreeret i databasen. I det her tilfælde er det *Skills*-tabellen, med felterne *ID*, *Title* og *Category* (se kodeeksempel 17 i bilag A). Når der så ændres i ens *context*-klasse (som er modellen af ens database), skal man tilføje en ny migration, så der kan migreres alt efter hvilken version af databasen man ønsker. Dette gør man efter man har tilføjet de model-klasser samt properties der gør sig gældende for det man implementerer og herefter ændrer sin *context*-klasse. Derefter eksekverer man kommandoen *add-migration* i *Package Manager Console*, og giver migrationen et navn. F.eks. *add-migration AddEmployee*. På denne måde ved man, hvilke ændringer der er foretaget for denne migrering. Dette opretter en ny klasse i mappen *Migrations* ligesom den foregående, *..._AddEmployee*. Et eksempel på denne klasse findes i kodeeksempel 3.

```
1       public partial class AddEmployee : DbMigration
2       {
3           public override void Up()
4           {
5               CreateTable(
6                   "dbo.Employees",
7                   c => new
8                   {
9                       ID = c.Int(nullable: false, identity: true),
10                      FirstName = c.String(),
11                      LastName = c.String(),
12                  })
13                  .PrimaryKey(t => t.ID);
14          }
15
16          public override void Down()
17          {
18              DropTable("dbo.Employees");
19          }
20      }
```

Kodeeksempel 3: Uddrag af migreringen der blev oprettet ved *add-migration AddEmployee*

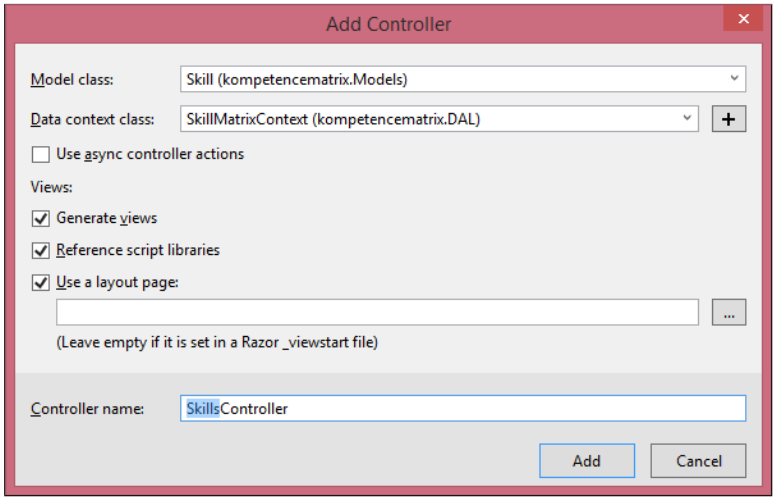
Når den nye migrering er på plads, og man er sikker på den er som den skal være, er man klar til at opdatere ens database til den nyeste version af ens migreringer. Dette gøres vha. kommandoen *update-database*. Denne kommando kører de SQL-scripts der gør sig gældende for den nyeste migrering.

Man kan specificere et *flag* (*update-databse -Verbose*) for at få vist de SQL-scripts der bliver kørt ved opdateringen. Når, eller hvis, migreringen lykkedes, vil det i det her tilfælde (jf. kodeeksempel 3) oprette en tabel ved navn *Employees*, hvor *ID* er primary key og identity (auto-increment), *FirstName* og *LastName* er kolonner i databasen, hvor indholdet er strings.

## 6.4 ASP.NET MVC

MVC er et framework der er lavet til ASP.NET og deri ligger skabeloner man kan anvende i Visual Studio. Da vi oprettede vores projekt, brugte vi netop sådan en skabelon. Frameworket kan endda arbejde sammen med Entity Framework, uden at der skal foretages yderligere konfigurationer, efter Entity er blevet installeret i projektet.

Når man anvender MVC skabelonen for et projekt, i Visual Studio, bliver der automatisk oprettet en grundlæggende infrastruktur samt et view med en forside med to statiske undersider; About og Contact.




Figur 8: Tilføjelse af MVC Controller med Entity Framework

For at tilføje en side med grundlæggende Create, Retrieve, Update og Delete (CRUD) funktionalitet i MVC, med Entity Framework, laver man først sin model klasse og tilpasser sin *Context*-klasse, som beskrevet i kapitlet omkring Entity Framework. Når man har styr på sin model, højreklikker man blot på Controller-mappen, der er blevet oprettet af skabelonen, og vælger *Add New Scaffolded Item...*. Man får dermed muligheden for at vælge oprettelse af en MVC controller der anvender Entity Framework. Når man så vælger at tilføje en MVC controller der anvender Entity Framework, får man vinduet frem, der er vist på figur 8.

### 6.4.1 Controller

I dette vindue indtaster man hvilken modelklasse man ønsker controlleren oprettet til, samt hvilken Context-klasse der anvendes. Man kan vælge om der skal oprettes et View automatisk, eller ej. MVC bruger et markupsprog der hedder Razor, til at lave den grafiske brugergrænseflade. Når man har valgt Context- og Modelklasse, trykker man på Add-knappen og controlleren bliver herefter oprettet. Controlleren der bliver oprettet indeholder som standard de metoder der er vist i kodeeksempel 4, med korte forklaringer.

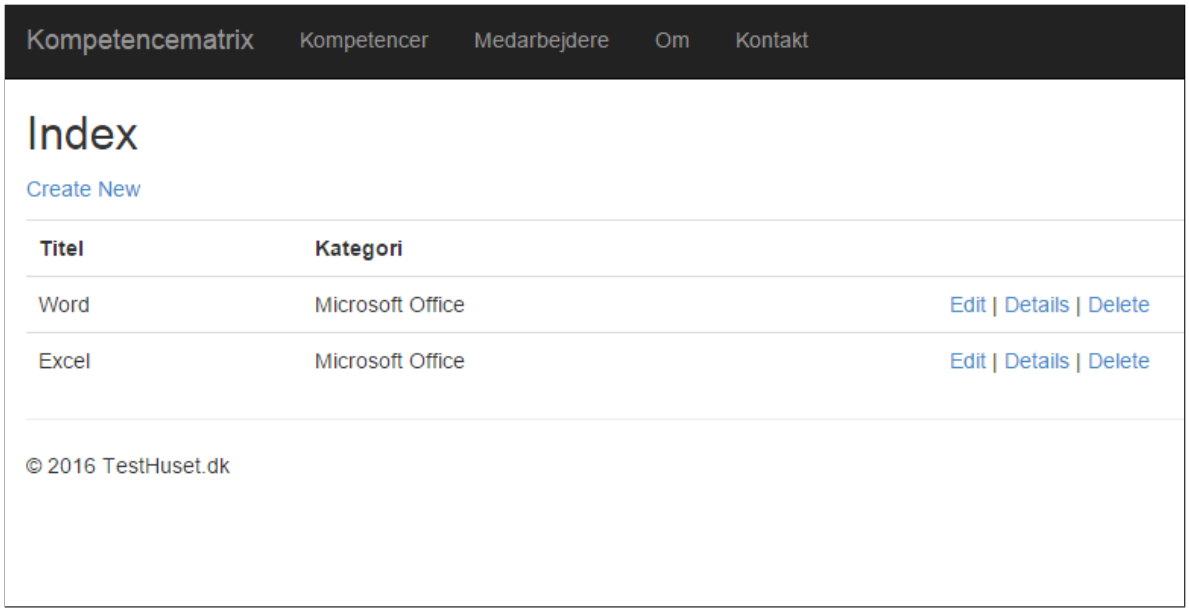
Controlleren i kodeeksempel 4 indeholder alle metoder for views der har med Skill at gøre. Det betyder at metoderne til de views der hører under Skill og ligger i controlleren. Så hvis man åbner siden Edit under Skills, for at redigere en enkelt forekomst, så bliver metoden Edit(int? id) kaldt i controlleren, for at hente informationerne omkring den bestemte skill som data model for viewet.

```
1       //GET: Henter alle forekomster af Skill i databasen
2       public ActionResult Index()
3       //GET: Henter information omkring en enkelt forekomst af Skill
4       public ActionResult Details(int? id)
5       //GET: Returnerer View'et for den side der kaldes
6       public ActionResult Create()
7       //POST: Opretter en ny Skill
8       public ActionResult Create([Bind(Include = "ID,Title,Category")]
        ↪  Skill skill)
9       //GET: Henter information omkring en enkelt forekomst af Skill
10      public ActionResult Edit(int? id)
11      //POST: Redigerer data for en forekomst af Skill
12      public ActionResult Edit([Bind(Include = "ID,Title,Category")]
        ↪  Skill skill)
13      //GET: Henter information omkring enkelt forekomst af Skill
14      public ActionResult Delete(int? id)
15      //POST: Sletter en forekomst af Skill
16      public ActionResult DeleteConfirmed(int id)
17      //Afslutter Entity Framework forbindelsen
18      protected override void Dispose(bool disposing)
```

Kodeeksempel 4: Metoderne i *SkillController*-klassen

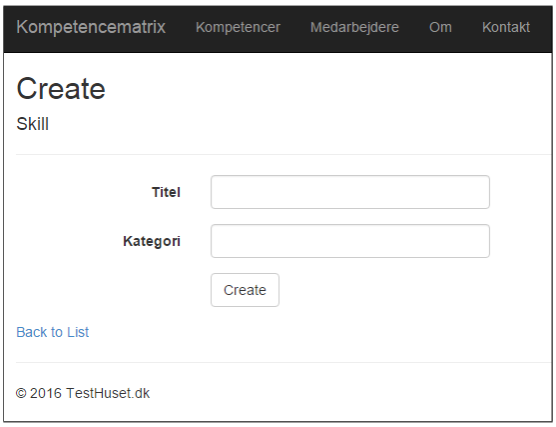
### 6.4.2 View

Selve viewet der indeholder websiderne bliver også autogenereret. I tilfældet med Skill bliver der oprettet en underside til sitet der hedder */Skill/* og indeholder en oversigt over alle de forekomster af Skill-klassen den er gemt i databasen. Se figur 9.




Figur 9: Oversigt over gemte Skills

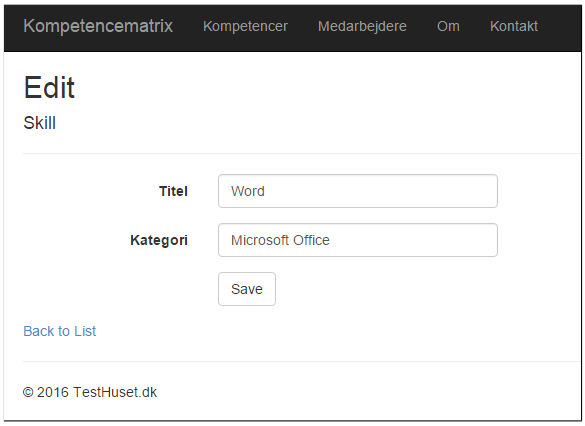
Som det ses bliver der oprettet et enkelt view der viser alle Skills i en liste, med links til, henholdsvis, at oprette en ny Skill, hente information omkring en enkelt Skill, redigere en Skills oplysninger og slette en Skill. Hvis man klikker på et af linksene kommer man til siderne der er vist på figur 10 til 13.



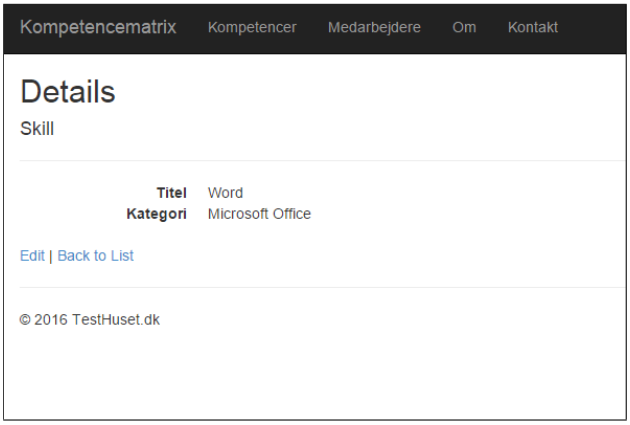

Figur 10: Oprettelse af en ny skill

ASP.NET MVC anvender et markup sprog der hedder Razor og et CSS framework der hedder Bootstrap. Disse gør de automatisk genererede views meget dynamiske at arbejde med, ved bl.a. at gøre designet responsivt. Det betyder at hvis man er på en lille skærm, som f.eks. en smartphone eller tablet, så tilpasser designet af hjemmesiden sig, så det altid er ordenligt at se på. På figur 14 kan man se to skærmprint af samme side som på figur 10, bare i et smalt vindue. Bemærk at menulinjen i toppen af siden er blevet til en dropdown-menu i øverste højre hjørne.

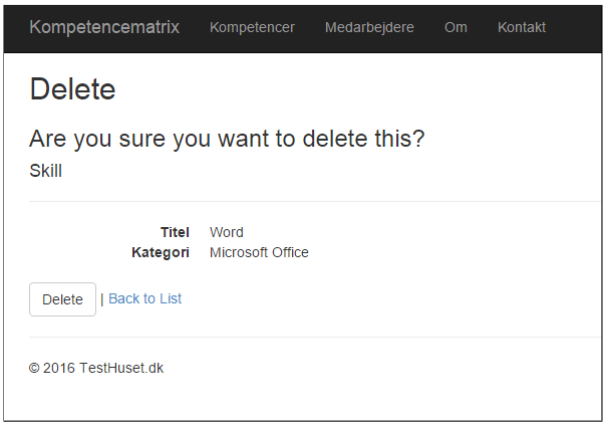
Markupsproget Razor bruges til at embedde C# kode på websider. MVC bruger sine egne klasser til at kalde metoderne i Controlleren fra View'et. Når man beder om at få åbnet en webside, bliver der




Figur 11: Redigering af en gemt Skill

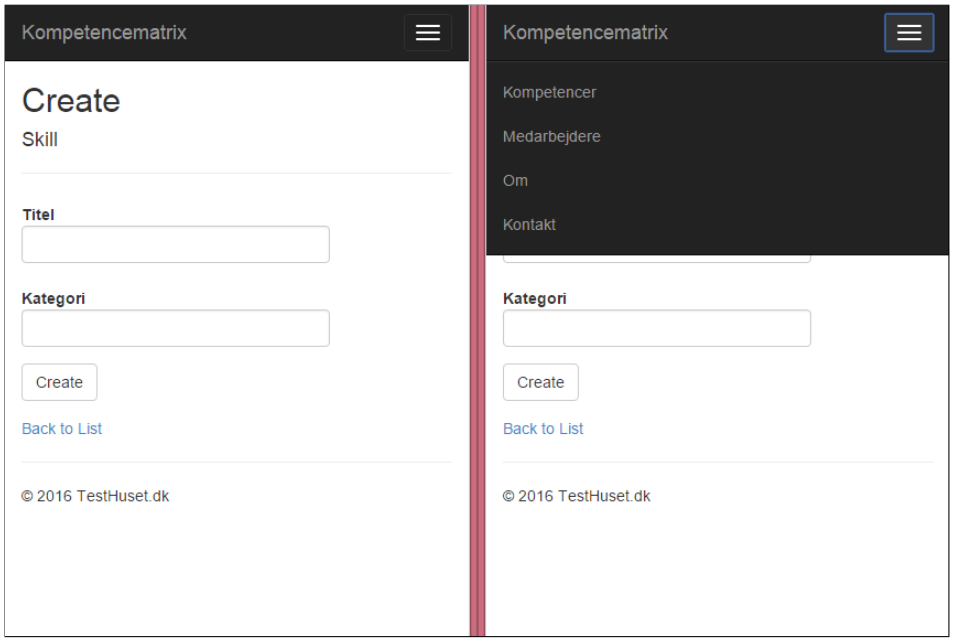



Figur 12: Detaljer omkring en gemt Skill




Figur 13: Prompt for sletning af Skill

kaldt en *Index()*-metode i controlleren for den side. Denne kan så returnere en datamodel med data fra databasen, så snart siden åbnes. I tilfældet med Skills kaldes *context*-klassen (db) og returnerer en liste med alle Skills i databasen. Se kodeeksempel 5.




Figur 14: Siden til oprettelse af ny Skill i et smalt vindue. Til højre med dropdown-menuen udfoldet.

```
1  public ActionResult Index()
2  {
3      return View(db.Skills.ToList());
4  }
```

Kodeeksempel 5: Uddrag af Controller-klasse for medarbejdere

Når Viewet så indlæses kan man kalde den medsendte datamodel i Razorkoden, for det pågældende view der er gemt i *.cshtml*-filformat. Det ser ud som på kodeeksempel 6 når man kalder datamodellen fra Razor.

I linje 1 er et embedded stykke C# kode, angivet med et @, som er måden hvorpå man indikerer embedded kode i Razor. Her er lavet en helt enkel *foreach*-løkke der vil iterere over hvert objekt der er med i den medfølgende datamodel. For hvert object vil der i dette tilfælde blive genereret en ny række i en tabel. Der vil på hver række være tre celler, indeholdende henholdsvis navn på Skill (linje 4), Kategori på Skill (linje 7) og i den sidste celle vil der være tre links til de førnævnte sider til redigering, detaljer og sletning (linje 10-12).

Som man måske kan se ud fra kodeeksempel 6 er der ikke angivet nogen form for styles i View-filerne for websiderne. Alt styling bliver håndteret af tilknyttede Cascading Style Sheets (CSS). Så selve View-filerne er kun markup med indholdet struktureret i tabeller. Vi vil ikke gå i detaljer med CSS, da det ikke har relevant for os at beskæftige os med det, i dette projekt.

## 6.5 Unit Testing

Unit Tests er afgørende for at praktisere Continuous Integration, da de er essentielle for at få projektet til at være selvtestende. Vi var lidt i tvivl om hvordan man lavede vores Unit Tests i praksis,

```
@foreach (var item in Model) {
<tr>
    <td>
        @Html.DisplayFor(modelItem => item.Title)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.Category)
    </td>
    <td>
        @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
        @Html.ActionLink("Details", "Details", new { id=item.ID }) |
        @Html.ActionLink("Delete", "Delete", new { id=item.ID })
    </td>
</tr>
}
```

Kodeeksempel 6: Uddrag af Controller-klasse for medarbejdere

da vi skulle anvende nogle frameworks vi ikke havde arbejdet med før. Vi fik dog lidt hjælp fra Martin Callesen fra Testhuset.

```
public class EmployeeController : Controller
{
    ...

    // GET: Employee
    public ActionResult Index()
    {
        return View(employeeRepo.GetAllEmployees());
    }

    // GET: Employee/Create
    public ActionResult Create()
    {
        return View();
    }

    ...
}
```

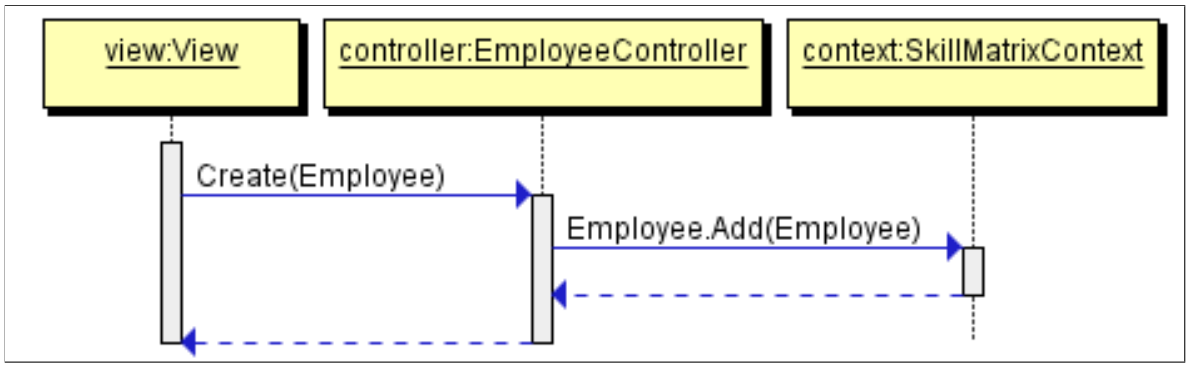
Kodeeksempel 7: Uddrag af Controller-klasse for medarbejdere

På kodeeksempel 7 kan man se de to metoder der anvendes til henholdsvis at oprette og hente nye medarbejdere. De returnerer begge typen *ActionResult*, der er en klasse i MVC. Vi har ikke kunne finde en måde at anvende *ActionResult*-klassen til at udhente data på, men der er heldigvis nedarvede klasser man kan anvende i stedet. Vi har anvendt en nedarvet klasse der hedder *ViewResult* der med sin *Model()*-metode kan returnere den modelklasse der er indeholdt i *ActionResult*.

Udover at vi skulle lære vores frameworks at kende, så skulle vi også gøre koden mere testbar. Da Unit Tests skal afvikles *in-memory* og derfor ikke skal skrive direkte til databasen, så var det nødvendigt at gøre dette muligt.

### 6.5.1 Testbar arkitektur

For at gøre det muligt at teste vores kode med Unit Tests, i en *In-Memory* database, har vi været nødt til at implementere et Repository-pattern i vores arkitektur.[5] På den måde kan vi *injecte* en repository-klasse ind, der agerer database, i stedet for den rigtige database. På figur 15 er et sekvensdiagram der viser hvordan en medarbejder bliver gemt, når vi blot anvender MVC og Entity Framework, når databasen kontaktes direkte. View'et beder *Controller*-klassen om at oprette en ny medarbejder. *Controller*-klassen giver herefter beskeden videre til vores *Context*-klasse, der håndterer alt hvad der skal skrives og hentes i databasen. Men da vi ikke vil have vores Unit Tests til at skrive til databasen, er vi nødt til at finde en måde at teste koden, uden at databasen kontaktes. Det er ikke muligt, uden at ændre en smule i vores arkitektur.

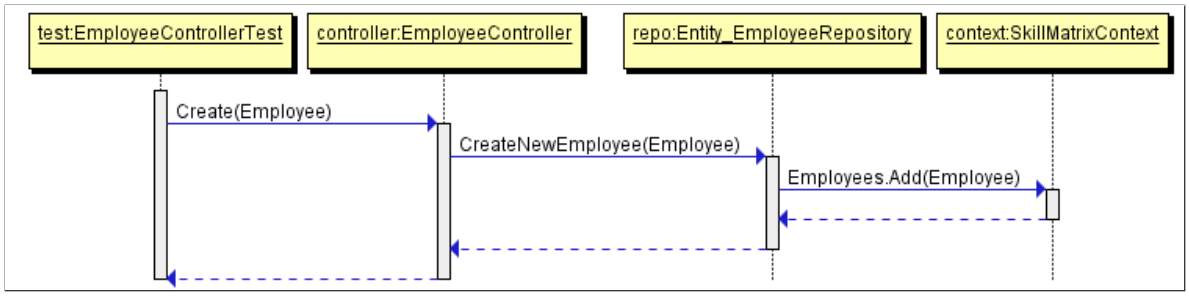



Figur 15: Sekvensdiagram over oprettelse af medarbejder i ASP.NET MVC med Entity Framework

Vi er nødt til at lægge et lag ind i MVC arkitekturen, som kan tilpasses alt efter om vi skal skrive til databasen eller til anden klasse, der kun kører i hukommelsen. På figur 16 har vi lavet et sekvensdiagram der viser hvordan applikationen gemmer en ny medarbejder, efter vi har implementeret vores *Repository*. Hvis man sammenligner med figur 16 kan man se, at der er implementeret en Repository-klasse mellem controller- og context-klasserne, kaldet *Entity_EmployeeRepository*. Repository-klassen anvender et interface, kaldet *IEmployeeRepository*, der indeholder de metoder der skal bruges (i dette tilfælde *CreateNewEmployee()*. På den måde kan vi oprette andre Repositories der ikke nødvendigvis skriver til databasen, gennem context-klassen, men implementerer *IEmployeeRepository*-interfacet. Vi har implementeret en overloaded constructor i EmployeeController, der gør at man kan angive et andet repository der skal anvendes, når man instantierer *Controller*-klassen. (Kodeeksempel 8)

Når vi har tilpasset vores arkitektur, så den nu anvender repositories mellem *controller*- og *context*-klasserne, kan vi lave vores unit tests korrekt, således at de ikke skriver til databasen, men en in-memory collection i stedet. På figur 17 er et sekvensdiagram over hvordan vores Unit Test oprettelse af en medarbejder kommer til at se ud. Vi har nu lavet en klasse der hedder *InMemory_EmployeeRepository* og implementerer det førnævnte *IEmployeeRepository*-interface. I *InMemory_EmployeeRepository*-klassen er der i stedet for metoder til at gemme i en database, blot blevet lavet en collection (i dette tilfælde en `List<Employee>`) der gemmer medarbejderne i hukommelsen. Unit Testen kan dermed instantiere *EmployeeController*-klassen og angive *InMemory_EmployeeRepository* til at være det repository der skal anvendes.

[5] MSDN. *Using TDD with ASP.NET MVC*. URL: `https://msdn.microsoft.com/en-us/library/ff847525`.




Figur 16: Sekvensdiagram over oprettelse af medarbejder med Repository implementeret

```
1       public class EmployeeController : Controller
2       {
3           private IEmployeeRepository employeeRepo;
4
5           public EmployeeController()
6           {
7               employeeRepo = new Entity_EmployeeRepository();
8           }
9
10          public EmployeeController(IEmployeeRepository repo)
11          {
12              employeeRepo = repo;
13          }
14          ...
15      }
```

Kodeeksempel 8: Constructors i Controller-klassen for medarbejdere

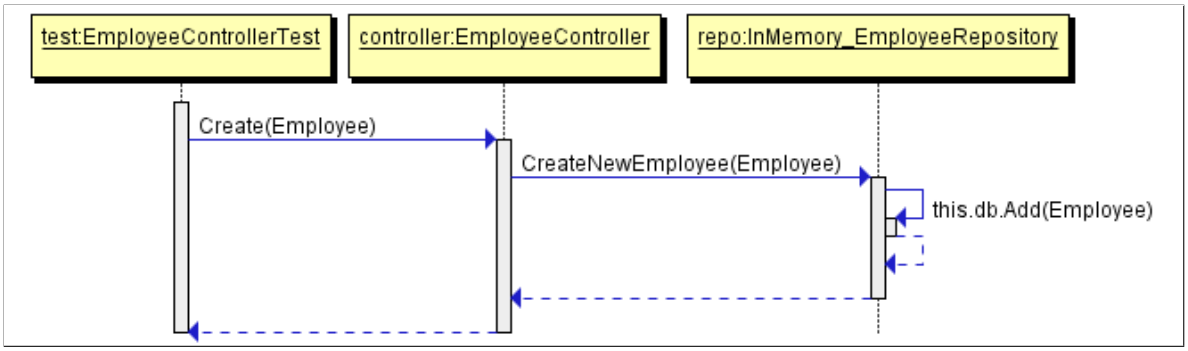
### 6.5.2 Udformning af Unit Tests

Da vi havde udfordringer med komme i gang med at kode Unit Tests, gik vi til Martin Callesen hos Testhuset, der har stor erfaring med Unit Tests. Han bad os om at undersøge *Given, When, Then*-metoden at lave Unit Tests på. Derudover gav han os også en introduktion til metoden. Hovedtanken bag *Given, When, Then*-metoden er at den skal være læselig for folk der ikke nødvendigvis har forstand på at kode. Metoden er taget fra Behaviour Driven Development (BDD).[6]

På kodeeksempel 9 er den Unit Test vi har skrevet, der tester oprettelsen og forespørgslen af flere medarbejdere. På linje 6 angiver vi en metode der beskriver en forudsætning eller antagelse (*Given*) om at der er tilføjet nogle brugere. Metoden *GenerateAllEmployees()* repræsenterer de brugere der skal oprettes i Unit Testen.

På linje 7 angiver vi en handling der skal udføres. I dette tilfælde beder vi om at få indhentet informationen fra databasen omkring alle brugere. Sidst på linje 8 beskriver vi hvad vi forventer der skal ske når *When*-metoden er kørt. Vi vil her gerne have at alle medarbejdere der er gemt i databasen bliver hentet ud. Før de nævnte metoder, på linje 4, har vi en metode der klargør vores *Mock*. Dette er instantieringen af *EmployeeController*-klassen der skal sættes op sammen med en instans af vores *InMemory_EmployeeRepository*-klasse.

[6]Martin Fowler. *GivenWhenThen*. URL: http://martinfowler.com/bliki/GivenWhenThen.html.




Figur 17: Sekvensdiagram over test af oprettelse af medarbejder med Repository implementeret

```
1           [TestMethod]
2           public void GetAllEmployees()
3           {
4               SetupMock();
5
6               GivenEmployeesAreAdded(GenerateAllEmployees());
7               WhenGettingAllEmployees();
8               ThenAllEmployeesAreGotten();
9           }
```

Kodeeksempel 9: Udformining af unit tests ved brug af metoden fra BDD

Når man har skrevet Unit Testen som vist på kodeeksempel 9, kan man begynde at skrive koden i metoderne. I kodeeksempel 10 er Metoden der genererer medarbejderne til databasen vist. Det er en *List<Employee>*-collection indeholdende en mængde *Employee*-objekter.

I kodeeksempel 11 har vi metoden der tilføjer alle medarbejderne til *In-memory*-databasen. Det er en *foreach*-løkke der opretter alle medarbejderne i den angivne collection ved hjælp af *controller*-objektets *Create()*-metode. *controller*-objektet er en instantiering af *EmployeeController*-klassen der har fået en instans af *InMemory_EmployeeRepository*-klassen medsendt. Se kodeeksempel 12.

På kodeeksempel 13 er de to sidste metoder der bliver brugt i vores Unit Test. I *WhenGettingAllEmpoyees()* på linje 1 henter vi alle medarbejdere fra *in-memory*-databasen ud og gemmer dem i en property af typen *ViewResult* der hedder *viewResult*. *ViewResult* er en nedarvning af *ActionResult*, som vi tidligere nævnte som return typen for kald til Controller-klasserne i ASP.NET MVC. På linje 6 er metoden *ThenAllEmployeesAreGotten()*, der indeholder vores *Assert*-metode. Da vi henter flere objekter der skal kontrolleres, bruges der her klassen *CollectionAssert*. Denne klasse kan kontrollere om to collections indeholder ens objekter. Vores liste med forventede objekter er gemt i attributten *allEmployees*, som vi angav en værdi for i kodeeksempel 10. Den faktiske liste er den *viewResult()* attribut vi hentede ud i *WhenGettingAllEmployees()*-metoden. *ViewResults' Model*-property returnerer datamodellen for det data der er hentet ud. Det gør at man kan typecaste den som en *List<>*.

Ved at strukturere sine Unit Test på denne måde, får man brudt den ned i mere overskuelige komponenter. Det tvinger også udvikleren til at tænke mere uafhængigt af den øvrige kode, da man kun har en lille opgave at løse; som f.eks. at gemme en liste af medarbejdere. Derudover kan man, ved at kigge på testmetoden, også nemt læse hvad det er testen tester, da navngivningen er formuleret i

```
1           private IEnumerable<Employee> GenerateAllEmployees()
2           {
3               List<Employee> employeesToBeAdded = new List<Employee>()
4               {
5                   new Employee() { ID = 1, FirstName = "Peter", LastName
                    ↪  = "Nielsen" },
6                   new Employee() { ID = 2, FirstName = "Martin", LastName
                    ↪  = "Kiersgaard" },
7                   new Employee() { ID = 3, FirstName = "Kasper", LastName
                    ↪  = "Schmeichel"},
8                   new Employee() { ID = 4, FirstName = "Delvin", LastName
                    ↪  = "Breaux" },
9                   new Employee() { ID = 5, FirstName = "Mark", LastName =
                    ↪  "Ingram" },
10                  new Employee() { ID = 6, FirstName = "Nicklas",
                    ↪  LastName = "Bendtner" },
11                  new Employee() { ID = 7, FirstName = "Peter", LastName
                    ↪  = "Schmeichel" },
12                  new Employee() { ID = 8, FirstName = "Allan", LastName
                    ↪  = "Tange" }
13              };
14              return allEmployees = employeesToBeAdded;
15          }
```

Kodeeksempel 10: GenerateAllEmployee() i Controller-testklassen for medarbejdere

```
1           private void GivenEmployeesAreAdded(IEnumerable<Employee>
            ↪  employees)
2           {
3               foreach (Employee e in employees)
4               {
5                   controller.Create(e);
6               }
7           }
```

Kodeeksempel 11: GivenEmployeesAreAdded() i Controller-testklassen for medarbejdere

```
1           private void SetupMock()
2           {
3               controller = new EmployeeController(new
                ↪  InMemory_EmployeeRepository());
4           }
```

Kodeeksempel 12: SetupMock() i Controller-testklassen for medarbejdere

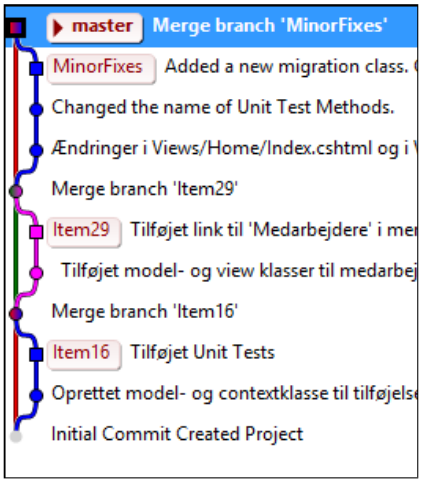
et forklarende sprog. Den beskrevne Unit Test tester både oprettelse og forespørgsel af data. Dermed får man testet flere kodelinjer/metoder i samme testmetode.

```
1           private void WhenGettingAllEmployees()
2           {
3               viewResult = controller.Index() as ViewResult;
4           }
5
6           private void ThenAllEmployeesAreGotten()
7           {
8               CollectionAssert.AreEqual(allEmployees, viewResult.Model as
                ↪  List<Employee>);
9           }
```

Kodeeksempel 13: WhenGettingAllEmployees() og ThenAllEmployeesAreGotten() i Controller-testklassen for medarbejdere

## 6.6 Branching-strategi

I forbindelse med, at vi benytter Visual Studio Online som repository for vores projekt, og derved også denne som vores Git-repository, var vi nødt til at kigge på, hvordan vi ville *pushe* de *commits* vi havde til koden. Da vores *master*-branch er den branch, som indeholder det produkt, som er unit testet, og klar til fremvisning, ville vi undgå, at *pushe* kode, som ikke er testet, men også kode som ikke består de beskrevne tests vi har. Derfor bestemte vi os for, at benytte os af branches under udviklingen af programmet. Dette gjorde vi ved, at for hver user story i vores *product backlog* for sprintet, oprettede vi en ny branch. Når vi arbejdede på programmet, sørgede vi for, kun at *pushe* til denne. Når unit tests var skrevet, og testet lokalt, *pushede* vi branchen endnu en gang, og derefter *mergede* vi denne ind i *master*-branchen. Dette medfører, at vi ikke rykker et *build* ind i *master*-branchen som ikke er funktionelt, i overensstemmelse med CI-foreskrifterne omkring byg og test.




Figur 18: Vores branches repræsenteret visuelt

På figur 18 er vist hvordan vores Git-repository så ud, efter sprint 1. Som man kan se har vi haft holdt et hovedspor, der ikke er blevet committet direkte til. I stedet har vi oprettet branches der har løbet parallelt med hovedsporet og når vi har været færdige med at kode, i dette tilfælde to User Stories og retning af småting, og sikret at vi kunne bygge og vores unit tests kunne fuldføres, har vi flettet vores parallelle spor ind i hovedsporet. Vi har på den måde bibeholdt en "ren" main-branch (i dette tilfælde vores *master*-branch).

## 6.7 Sprint review meeting d. 17. december

Efter lidt udfordringer med at få aftalt et møde med vores Product Owner, grundet travlhed, fik vi endelig afholdt et møde d. 17. december. Målet for mødet var at præsentere det arbejde vi havde udført i det første sprint af projektet. Samtidig ville vi også afholde Sprint planning mødet for sprint 2, men det vil vi gennemgå i kapitlet omkring dette.

Vi var spændte på at vise det vi havde lavet, da vi havde leveret et bud på en løsning, samtidig med at vi havde et fuldt funktionelt miljø at demonstrere applikationen på. Som det er forklaret tidligere i

dette kapitel, så bliver den kode vi committer til vores Git-repository automatisk deployet til Microsoft Azure og kan dermed demonstreres og anvendes kort tid efter, når de automatiserede byg og tests er afviklet.

På figur 19 kan man se forsiden på vores applikation. Det er en standard ASP.NET MVC forside, hvor vi har erstattet teksten med generiske *Lorem Ipsum* tekstblokke. Alle links er desuden ændret, så de ikke peger på eksterne sider; hvilket er tilfældet med standardforsiden.

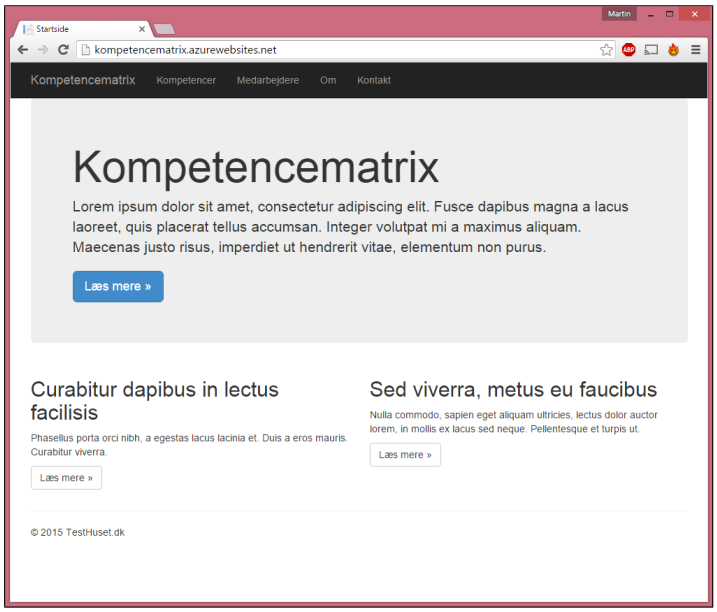
| Navn: | Stilling |
|---|---|
| Stine Osted | Service Manager |
| Peter Nielsen | |
| Martin Kiersgaard | |

Tabel 3: Deltagere på mødet

Da vi ikke har angivet acceptkriterier på vores User Stories, har vi kun formuleringen af opgaven som kriterium for om opgaven er løst. De to User Stories der var planlagt i dette sprint var:

- **Backlog Item 29** Som medarbejder i TestHuset skal jeg være oprettet i systemet
- **Backlog Item 16** Som administrator af programmet skal jeg kunne tilføje nye kompetencer til databasen

Vores Product Owner havde nogle kommentarer til vores løsning. Formuleringen af *Item 29* var ment som en opfordring til at indhente oplysninger omkring medarbejdere automatisk. Men da der var nogle sikkerhedsmæssige udfordringer i forhold til at tilgå TestHusets domæne, der befinder sig i et produktionsmiljø, fra en offentligt tilgængelig lokation, havde vi valgt den "mindre smarte" løsning, hvor man manuelt opretter hver medarbejder. Dette rationale blev accepteret af Stine. Fælles for de to User Stories var, at der kun var formuleret at man skulle kunne oprette henholdsvis medarbejdere og kompetencer. Men redigering og sletning var ikke en del af de udvalgte User Stories, så disse funktioner var ikke med i løsningen. Der blev vi enige om at gennemtænke fremtidige acceptkriterier lidt bedre. Det er dog nogle funktioner der kan implementeres enkelt og hurtigt på et senere tidspunkt, så de bliver oprettet som User Stories og tilføjet til vores Product Backlog. Alt i alt blev løsningen godkendt og vi kan dermed gå videre til næste sprint.




Figur 19: Vores applikation efter sprint 1

## 6.8 Retrospective

Vi indledte vores Retrospective med at kompilere en liste over de ting vi var tilfredse med og de ting vi gerne vil forbedre i næste sprint.

**Hvad var godt**

- Implementering af byggeserveren gik meget smertefrit
- Opsætning af automatisk deployment til Azure
- Oprettelse/organisering af User Stories/Tasks
- Sprint planning meeting gik rigtig godt
- MVC og Entity Framework virker rigtig godt sammen og vi har gjort os gode erfaringer

**Hvad kan blive bedre?**

- Håndtering af branches kræver tilvænning
- Skrivning af Unit Tests
- Planlægning af møder skal gøres på forhånd, så deadlines ikke skrider

Vi er alt i alt godt tilfredse med resultatet af vores første sprint. Vi har fået kodet de User Stories vi satte os for og vigtigst af alt, så har vi fået implementeret en masse CI praktikker. Den største udfordring vi har haft, har været at kode vores Unit Tests og huske at lave nye branches i Git, før vi begynder at kode. Det har været stort set hver gang vi skulle i gang med at kode, at vi har glemt at lave en ny branch og været nødt til at skrotte alle rettelser og starte forfra efter at have lavet en. Det er et spørgsmål om vane, at vi vænner os til at se på hvilken branch vi har checket ud, inden vi begynder at kode.

Med hensyn til Unit Tests, så kan vi kun blive bedre ved at lave flere af dem. Man bruger meget ekstra tid på at lave sine tests, men vi kan sagtens se idéen i at man, for fremtiden, ikke behøver at bekymre sig unødigt om det kode man har lavet tidligere, stadigvæk virker efter hensigten. Martin Callesens forslag om at opdele unit testen i mindre komponenter var en stor hjælp, så vi er fortrøstningsfulde hvad angår de tests vi skal kode i fremtiden.

# 7 Sprint 2

I andet sprint, er ønsket, at det skal være muligt at foretage operationer på de allerede oprettede medarbejdere og kompetencer. Derudover skal det også være muligt at søge på allerede oprettede kompetencer og medarbejdere. Undervejs vil vi igen, anvende de praktikker vi har valgt fra forrige sprint.

## 7.1 Sprint planning meeting d. 17. december

Ved mødet med Stine Osted, diskuterede vi hvilke user stories der ville være realistiske for os at nå, med henblik på rapportskrivning samt vi bevægede os ind i jul- og nytårsdagene. Derfor aftalte vi, at de user stories vi ville tage hul på, ville være operationer hvor administratoren kan udføre disse på en medarbejder eller kompetence. Derudover mente vi også, at det kunne være nødvendigt med et søgefelt for lettere at finde frem til relevante kompetencer eller medarbejdere, når disse er oprettet i systemet. Dette synes vi alle lød fornuftigt, og vi vedtog, at de user stories vi ville påbegynde i sprint 2 var:

| Navn: | Stilling |
|---|---|
| Stine Osted | Service Manager |
| Peter Nielsen | |
| Martin Kiersgaard | |

Tabel 4: Deltagere på mødet

**Backlog Item 39** Som administrator skal jeg kunne hente/ændre/slette medarbejdere i databasen

**Backlog Item 40** Som administrator skal jeg kunne hente/ændre/slette kompetencer i databasen

**Backlog Item 41** Som bruger af programmet skal jeg kunne søge på medarbejdere

**Backlog Item 42** Som bruger af programmet skal jeg kunne søge på kompetencer

Sprint review meeting aftalte vi skulle ske d. 28. december.

## 7.2 Nedbrydning af User Stories

Vi diskuterede internt hvorvidt det var nødvendigt at nedbryde de user stories vi vil arbejde med i dette sprint til tasks. Grunden til dette er, at der er ikke så stor abstraktion i disse og arbejdet kan let påtages af en enkelt person, da de minder så meget om hinanden. Der skal dog laves *views* for disse, hvilket kan argumenteres for, at den anden kunne lave i mellemtiden. Derfor aftalte vi, at den ene af os havde ansvaret for logikken, mens den anden havde ansvaret for at få lavet *views* og links til de forskellige operationer. Derudover skal koden også testes hvilket vi begge burde være med til, da vi stadig har udfordringer med at kode dem og mener at parprogrammering ville være en fordel. Vi endte derfor med at nedbryde hver enkelt user story ned i nogle tasks, som hver enkelt af os påtog. For de to user stories med redigering/ændring/slettelse af henholdsvis kompetence og medarbejder, blev det til følgende tasks:

**Task 43** Oprettelse af metoder der kan udføre operationerne RUD (*R*etrieve/*U*pdate/*D*elete)

**Task 44** Oprettelse af views/links til udførsel af operationerne RUD

**Task 45** Oprettelse og eksekvering af unit tests til operationerne RUD

For de to user stories med søgning blev de nedbrudt til følgende tasks:

**Task 46** Oprettelse af metod(er) der kan udføre en operation for en specifik søgetekst

**Task 47** Oprettelse af søgefelt i view så det er muligt at indtaste søgetekst samt knap til eksekvering af søgning

**Task 48** Oprettelse af unit test og eksekvering af disse

Disse tasks dækker som sagt over de user stories (eller backlog items) vi har valgt i dette sprint.

## 7.3 Kode

**RUD operationer** Da vi skulle implementere løsningen til disse operationer, kunne vi få ASP.NET MVC frameworket til at oprette dette for os, ved hjælp af scaffolding, som beskrevet tidligere. Dette skulle selvfølgelig tilpasses efter vores ønske om at bruge *repository*-pattern, samt ændringer til sproget på siden. Derfor i *controller*-klassen hvor vi havde kald til vores *context*-klasse skulle dette ændres til kald til vores *repository*. Derefter varetager vores *repository* kaldene til databasen eller det data der ligger i hukommelsen, når testene skal køres. Dette medfører, at vores *repository* skal indordnes herefter. Derfor skal der oprettes metoder der kan håndtere de metoder der er for *RUD* operationerne. I kodeeksempel 14 ses disse metoder implementeret for vores *InMemory_EmployeeRepository* som varetager kaldende fra *controller* til *"databasen"*. De samme metoder/implementeringer gør sig gældende for *InMemory_SkillRepository* da disse er ens. Udover dette, blev unit tests for disse to user stories implementeret.

```
1  private List<Employee> db = new List<Employee>();
2  ...
3  public Employee GetEmployeeById(int? id)
4  {
5      return db.FirstOrDefault(e => e.ID == id);
6  }
7
8  public void UpdateEmployee(Employee employee)
9  {
10     int index = db.FindIndex(e => e.ID == employee.ID);
11     db[index] = employee;
12 }
13
14 public void DeleteEmployee(int id)
15 {
16     db.RemoveAll(e => e.ID == id);
17 }
```

Kodeeksempel 14: Uddrag af *InMemory_EmployeeRepository* for de metoder der tager sig af *RUD* operationerne

Der var ikke de store skavanker set i forhold til forrige sprint, men der var stadig problemer med at håndtere måden hvorpå man sikrer, at en medarbejder i det her eksempel, f.eks. bliver slettet fra ens liste i hukommelsen. Til dette brugte vi *CollectionAssert*, som er en klasse i *MS-Test* som vi kan bruge til at tjekke hvorvidt en kollektion indeholder et givent element. Et sådant eksempel på vores unit test kan ses på kodeeksempel 15.

På kodeeksempel 15 ses det, at vi i *WhenUpdatingEmployee*-metoden kalder vores *controllers Edit*-metode. Dette returnerer vi som et *ViewResult* hvor vi kan hive den tilhørende model ud. Denne ligger vi over i en global variabel kaldet *updatedEmployee*. Når vi så har redigeret i en medarbejder, skal vi tjekke hvorvidt ændringen har fundet sted. Dette gør vi ved at kigge på den kollektion vi kan få ud af det *ViewResult* vi kan få fra *Index*. Denne indeholder vores "database" som ligger i

```
1   private void ThenEmployeeIsUpdated()
2   {
3       ViewResult result = controller.Index() as ViewResult;
4       List<Employee> employeeList = result.ViewData.Model as
        ↪  List<Employee>;
5       CollectionAssert.Contains(employeeList, updatedEmployee);
6   }
7
8   private void WhenUpdatingEmployee()
9   {
10      viewResult = controller.Edit(currentEmployee.ID) as ViewResult;
11      updatedEmployee = viewResult.ViewData.Model as Employee;
12      updatedEmployee.FirstName = "Johannes";
13      updatedEmployee.LastName = "Jensen";
14  }
15  ...
16  private void ThenEmployeeIsDeleted()
17  {
18      ...
19      CollectionAssert.DoesNotContain(employeeResult, currentEmployee);
20  }
```

Kodeeksempel 15: Uddrag af unit tests for *EmployeeController*

hukommelsen. Herpå benytter vi os af *CollectionAssert*-klassen for at tjekke, at den liste vi har fået indeholder den redigerede medarbejder. Det samme gør sig gældende for når *ThenEmployeeIsDeleted*, men her tjekker vi omvendt for, at kollektionen *ikke* indeholder den nuværende medarbejder, som vi ellers har haft oprettet i systemet tidligere. Derudover er unit tests for kompetencer de samme som for medarbejder, da de udfører de samme operationer.

**Søgemulighederne** skulle også implementeres i dette sprint, da vi mente det var en overkommelig proces. Vi startede med at kigge på hvordan andre havde haft implementeret denne løsning. Disse havde alle lagt deres søgelogik i selve *Index*-metoden for *controller*-klassen hvilket ikke var relevant for os. Derfor kiggede vi på, hvordan de hentede søgningslisten og kiggede på hvordan vi kunne tilpasse dette til vores *repository*-pattern. Det viste sig egentlig at være lige til. Det eneste der ved søgningen er blot, at vi modtager en parameter fra brugeren i form af et tekstfelt. Dette bliver sendt ned igennem *controller*-klassen til vores *EmployeeRepository* som derefter foretager de *queries* der er nødvendige for at hente den data ud fra det indtastede søgeord. Vi afgjorde i samråd med Product Owner, at de søgekriterier der måtte være på nuværende tidspunkt skulle være for- og efternavn for medarbejdere, samt kategori og titel for kompetencer. På kodeeksempel 19-20 i bilag A kan man se hvordan implementeringen af søgning på en medarbejder er foretaget for henholdsvis *EmployeeController* og *EmployeeRepository*.

Senere kan denne søgemetode udvides til at søge på medarbejdere der har en hvis kompetence ved at foretage queries på en tabel, der referer de to tilsammen. Søgemetoden i ASP.NET er som sådan skalerbar, man kan dog argumentere for om, at det tager for lang tid at foretage en søgning eller flere søgninger, i og med ens side bliver genopfrisket hver gang man foretager en søgning. En løsning på dette kunne være at anvende f.eks. AJAX og/eller jQuery.

## 7.4 Sprint review meeting d. 29. december

Vi fik afholdt et møde med vores Product Owner d. 29. december. Målet for mødet var endnu engang at præsentere det arbejde vi havde udført i dette sprint. Mødet var planlagt til d.28. men PO havde været nødt til at udskyde det til d. 29.

Vi var spændte på at vise den løsning vi havde lavet, da forrige sprint i bund og grund var en opstart af projektet, hvor grundstene skulle ligges og fundamentet skulle cementeres. Vi havde fået bevist, at de miljøer vi brugte var funktionelle og var derfor også klar til at prøve en gang mere, for at se, om det hele spillede endnu en gang.

| Navn: | Stilling |
|---|---|
| Stine Osted | Service Manager |
| Peter Nielsen | |
| Martin Kiersgaard | |

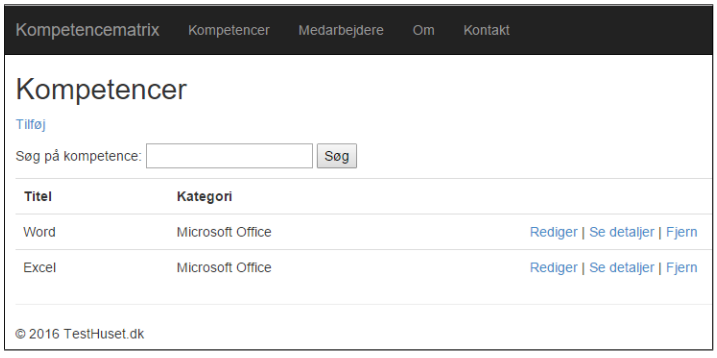
Tabel 5: Deltagere på mødet

På figur 20 kan man se siden for "kompetencer". Denne viser overordnet de implementationer der er foretaget i sprintet.

Vi fik en snak med Product Owner om hvordan sprintet var gået, og hvordan vi følte arbejdsprocessen var. Vi var begge enige om, at have et fastsat møde var klart at foretrække, men vi kunne sagtens se problemet i og med TestHuset er en virksomhed der også har andre at tænke på. Derudover mente vi, at arbejdsprocessen i dette sprint måske var for lidt arbejde i forhold til hvad vi havde af tid. Dette kan der argumenteres både for og i mod, sprintet har trods alt løbet hen over julen. Vi ser dog ikke produktet spille den største rolle i projektet her, men vi havde stadig en følelse af, at vi godt kunne have kigget på, hvorledes vi kobler en medarbejder op på en eller flere kompetencer og derefter vise disse. Således havde det begyndt at tage mere form, end som det gør lige nu. Derudover fik vi igen snakken om "bruger" og "administrator", men da vi som nævnt tidligere ikke kan tilgå TestHusets domæne, har vi valgt at holde det åbent. Product Owner valgte at godkende løsningen som den er nu, og vi kunne gå videre med skrivning af rapporten.

## 7.5 Retrospective

Som vi gjorde i forrige sprint, lavede vi en liste over ting vi begge var tilfredse med og de ting vi gerne vil forbedre.




Figur 20: Kompetence siden efter sprint 2

### Hvad var godt?

- Implementering af de user stories der var valgt, gik smertefrit og uden besvær
- Oprettelse/organisering af User Stories/Tasks
- Sprint planning meeting gik rigtig godt
- Vi har skabt bedre erfaringer med hvordan MVC og Entity Framework fungerer, samt hvordan vi bedst mulig strukturer vores unit tests

**Hvad kan blive bedre?**

- Strukturering af arbejdsopgaver
- Planlægning samt kommunikation
- Hyppige *commits* til den branch der arbejdes på

Som nævnt går vi ud af sprint 2 med blandede følelser, da vi syntes at have undervurderet vores evner i forhold til påtaget arbejdet. Vi kan dog gå ud af sprint 2, med løftet pande, da vi løste de user stories som var blevet aftalt med Product Owner. Derudover har vi benyttet os af *pair-programming* i forbindelse med oprettelse af unit tests samt kodning af nogle af implementationerne. Derudover har vi stadig problemer med at få *comittet* vores arbejde til den branch vi arbejder på, eller glemsom med at få oprettet en ny, når en user story er færdig. Kald det ivrig, men når man er i gang, så kører det derudaf. Derfor skal vi blive mere struktureret i hvornår vi *comitter* og hele tiden være opmærksomme på hvor vi befinder os. Unit Tests føler vi også, at vi begge har fået bedre styr på. Brugen af *BDD* var lidt uoverskuelig i starten, men jo mere man beskæftiger sig med det, jo bedre overblik får man. Frem for én lang testmetode med en masse kommentarer om hvad der foregår på hvert enkelt trin. Alt i alt er vi tilfredse, men kunne også se, at vi kunne have taget, hvert fald én, user story mere med ind i dette sprint. Dette har selvfølgelig også noget at sige med hvordan vi føler om hvorvidt vi har styr på de frameworks vi har taget i brug, men også estimeringen af hvor lang tid det ene og det andet tager. Tidsestimering er noget man bliver bedre til i løbet af ens sprints, jo flere der er/kommer.

# 8 Perspektivering

For at finde ud af hvordan andre ser på hele Continuous Integration-processen har vi forsøgt at finde nogle erfaringer fra folk der har anvendt den. Vi fandt en artikel af Stig Andersen i Prosa-bladet, der har været ude og interviewe nogle virksomheder der, blandt andet, anvender CI.[7]

Ud fra hvad vi kan læse i artiklen, så nævner de personer der bliver interviewet at en af de største udfordringer har været unit tests, hvilket også har været vores erfaring. Det kræver væsentligt mere arbejde af udvikleren at sørge for at levere en test sammen med sin kode. Hvis man ikke har været vant til at tænke på at sin kode skal være testbar og lave unit tests, så vil man komme til at bruge meget ekstra tid på at lære at gøre det. Men der er en gevinst i den anden ende, da man får større tillid til det kode der bliver committet til projektet.

Artiklen handler om Continuous Delivery, hvilket også er meget interessant når man har implemeteret CI. Man kan se Continuous Delivery som en naturlig fortsættelse af Continuous Integration, hvor man automatiserer hele processen fra kode til produktion. Vores løsning har også muligheden for at kunne blive deployet til et produktionsmiljø, men mangler automatisering af en del kvalitets-gates, før vi mener at det ville være "forsvarligt" at gøre. Der er stadigvæk flere ting der ikke bliver testet i vores setup. Der er integrationer med andre systemer, i dette tilfælde ville det være kritisk at få testet vores system op mod en rigtig database i stedet for en stub. Derudover er der behov for at få testet brugergrænsefladen og dermed applikationen som helhed. Vores applikation og udviklingsmiljø er forberedt på at kunne deploye til et miljø hvori man kan foretage integrationstest og brugeraccepttest, men vi har ikke fået undersøgt hvordan man skulle implmentere det i praksis.

I en anden artikel af Yegor Bugayenko på DevOps.com, skriver forfatteren om et andet problem der ofte opstår i CI, nemlig integrationskonflikter, når man skal merge sine branches til hovedbranchen.[8] Det bliver især et problem i store softwareprojekter, hvor der kan komme mange commits hver dag. Merges til hovedsporet er et svagt led i CI-kæden, hvis det er den enkelte udvikler der selv har ansvaret for at merget lykkes, uden at der efterfølgende sker fejl. Der kan man argumentere for SCRUM's tilgang, med retrospectives og konstant selvevaluering og -forbedring burde kunne hjælpe til med at uddanne udviklerne, så hyppigheden af disse fejl mindskes. Endnu et argument kunne være at hvis man prioriterer deadlines højere end kvalitet, så må man indse at der må indgås kompromiser. Som nævnt i Stig Andersens artikel, så er det vigtigt at have organisationen i ryggen, hvis man vil ændre på tankegangen og processerne i virksomheden. Det mener vi er entydigt at læse ud af de to artikler. Bugayenkos bud på en løsning er at den menige udvikler ikke får adgang til at merge sin kode ind i hovedsporet, men at der implementeres et script der sørger for at teste bygget igennem, inden det bliver merget, og afviser merget, hvis der vil opstå fejl. På den måde fjerne man den menneskelige faktor og indfører flere automatiserede quality-gates. Alt i alt en god idé der igen er et skridt hen i retning af at automatisere alt der kan automatiseres, med henblik på en forbedring af kvaliteten. Vi har dog ikke selv været påvirket af merge-problemer, da vi ikke har haft arbejdet sideløbende på branches. Derfor har der kun været et parrallelspor der skulle merges tilbage i det samme uændrede hovedspor. Men vi har på tidligere skoleprojekter oplevet problemet med mergekonflikter, så vi forstår udfordringen der bliver beskrevet i artiklen.

I store træk kan vi nikke genkendende til de oplevelser vi har læst at folk har med Continuous Integration. Et problem vi ikke har fundet omtalt, som vi selv oplevede, er måden at håndtere branches på. Vi har gjort meget ud af at holde vores branches pæne og præsentable. Men vi har også brugt meget tid på at genskabe arbejde vi allerede har lavet, fordi vi skrottede en branch. Vi opdagede at

[7] Stig Andersen. "Continuous Delivery". In: *Prosabladet* 12 (2015), pp. 16–25.

[8] Yegor Bugayenko. *Why Continuous Integration Doesn't Work.* URL: http://devops.com/2014/09/26/continuous-integration-doesnt-work/.

vi havde glemt at oprette en branch og var ved at committe direkte i hovedbranchen eller en anden forkert branch. I realiteten er det nok bare os der er for optagede af at det skal se ordentligt ud. I virkeligheden er man nok mere opmærksom på at det ikke behøver at se godt ud, bare det virker. Ihvertfald når man ikke har mere erfaring end vi har. Man bliver uden tvivl bedre til at håndtere sin branches, jo mere erfaring man får med Git. Det er formodentlig også derfor at de formodede erfarne udviklere der er talt med i artiklerne, ikke har dette problem.

# 9 Konklusion

Fremtiden ligger efter vores mening i automatisering af så meget af udviklingsprocessen som muligt og Continuous Integration er et oplagt sted for alle udviklingsteams at starte.

**Hvad er tanker bag CI/CD** I bund og grund er tanken at skabe bedre software. Ved at man tester mere omfattende og tidligt i udviklingsforløbet og sørger for at testene bliver kørt ofte, har man etableret et hvis niveau af kvalitet. At man har en masse tests beviser selvfølgelig ikke at fejl ikke eksisterer. Men hvis det kan fange nogle fejl som først var blevet fundet meget senere i forløbet, så er det en gevinst og et løft i kvalitet. Derudover er alt hvad der kan automatiseres og hjælpe den yderligere udviklingsproces, som f.eks. at kunne deploye sin applikation til testmiljøer inden for få minutter, uvurderlig i forhold til spildtid der bliver brugt på at afvente leverancer der skal udføres manuelt.

**Hvilke fordele og ulemper er der ved at implementere CI/CD** En klar fordel er at man har en masse automatiserede tests, der bliver eksekveret ofte og tidligt i udviklingsforløbet. Som nævnt tidligere kan man aldrig påvise fraværet af fejl og hellere en mindre effektiv test der bliver eksekveret ofte end en perfekt test der aldrig bliver skrevet. Man kan sige at jo tættere på udviklerne en fejl bliver fundet, jo nemmere er den at rette. Hvis en unit test fejler, så er udviklernes fornemmeste opgave at rette fejlen. Hvis man opdager en fejl, mens applikationen er i produktion er det straks en helt anden proces der skal igangsættes. Så det burde kunne give gevinst på bundlinjen at prioritere kvaliteten frem for deadlines.

En anden fordel er at man, gennem sine værktøjer, hele tiden har overblik over kvaliteten af sin software. Faktisk kan man diskutere at man ved at arbejde strengt efter Continuous Integration principperne og have udviklere der skriver gode unit tests, aldrig vil have problemer med basale problemer i sin software. Det er selvfølgelig ikke en garanti for fejlfri software, da der er mange ting der kan gå galt i et softwareprojekt. Men man vil altid have overblikket over den grundlæggende kvalitet af sit byg.

For de testere der er i udviklingsteamet er det en fordel hurtigt at kunne få et testmiljø med en udgave af applikationen på, til udførsel af manuelle tests.

En ulempe ved CI er at det tager tid for teamet at lære. Man skal, som udvikler, lære at håndtere sine Branches i SCM-systemet og man skal lære at kode unit tests. Det kan, i hvert fald i begyndelsen, tage meget ekstra tid for udviklerne at skrive sine unit tests til den kode der er skrevet. Men på den positive side, så tvinger det også udviklerne til at skrive mere testbar, og dermed mere fleksibel, kode. For at man kan unit teste effektivt er koden i høj grad nødt til at være modulær og til at bryde ned i mindre dele. Det giver kode der er genbrugelig og har lavere kobling med den øvrige kode.

**Hvis man ønsker at anvende CI/CD, hvilke typer af værktøjer er så nødvendige?** Det er helt essentielt at have et Source Code Management-system, som Git og en Continuous Integration server, som Jenkins eller Team Foundation Server. Da al kode bliver opbevaret i SCM-systemet og Integrationsserveren overvåger dette repository, er begge dele nødvendige for at have en kontinuerlig integrationsproces kørende. Når de to værktøjer er på plads, er resten af Continuous Integration processen et spørgsmål om arbejdsgange og hvordan man bruger værktøjerne.

**Hvilken effekt har implementeringen af CI/CD på udviklingsprocessen, både i forhold til kvaliteten af softwaren, men også som en helhed?** I første omgang har vi kunne konkludere at udviklingsprocessen bliver forlænget, grundet kravene til de omfattende unit tests. Men i den anden

ende bliver tiden brugt på deployment til andre miljøer skåret ned, da der ikke længere skal sidde en person og foretage opsætning af miljø og installation af applikationen manuelt.

Som vi også har nævnt i nogle af de tidligere punkter i denne konklusion, så gavner det kvaliteten at man unit tester ofte og omfattende. Udviklere bliver gjort hurtigere opmærksomme på problemer med bygget og er tvunget til at rette fejlene hurtigt. Derudover får man også en større tiltro til sin software, hvilket kommer af at den automatisk bliver mere solid og forudsigelig.

**Hvilke udviklingsmetoder egner sig til brug med CI/CD?** Det oplagte svar ville være agile metoder, da CI er udviklet som en del af eXtreme Programming. Der er nogle åbenlyse punkter i det agile manifest der understøttes af CI:

- Vores højeste prioritet er at stille kunden tilfreds gennem tidlige og løbende afleveringer af værdifuld software.
- Løbende levering af velfungerende software, jo hyppigere des bedre.
- Fungerende software er den primære måde at måle fremdrift på.

CI er netop udviklet som en forudsætning for at kunne foretage "løbende leveringer" og en måde at komme nærmere på at lave "fungerende software". Begge dele gennem fokus på automatisering af deployment og test, samt en generel bevidsthed om vigtigheden i at teste tidligt i udviklingen. Men i teorien burde alle iterative udviklingsmetoder kunne gøre brug af CI, med fordele.

**Kan vi udvikle en applikation ved hjælp af CI/CD?** Det korte svar er; Ja, det kan vi godt. Værktøjerne til at implementere en CI proces findes allerede og er tilgængelige i både gratis og betalte versioner. De er endda relativt enkle at sætte op, så det er noget vi mener at alle agile udviklingsteams burde undersøge og vurdere om ikke det kunne være til gavn for dem. Udfordringen består i højere grad i at lære at bruge værktøjerne korrekt. Vi har påbegyndt udviklingen af en applikation, der desværre ikke blev færdig. Men vi har undervejs nået at observere hvor gavnligt det kan være at have fokus på grundige og automatiserede unit tests.

Vi mener at have påvist at Continuous Integration er essentielt for at drive et effektivt agilt udviklingsteam. Hvis man har en idé om at man gerne vil være i stand til at kunne levere softwarereleases i korte intervaller, er CI stedet man starter, i sin stræben på at indføre en komplet Continuous Delivery setup. Det vil kræve en ændring i måden som mange udviklingsteams arbejder på i dag, men vi er sikre på at det vil gavne kvaliteten og konsistensen af deres leverancer.

# Kildeliste


[1] Ernest Mueller et al. *What Is DevOps?* URL: http://theagileadmin.com/what-is-devops/.

[2] *The Rules of Extreme Programming.* URL: http://www.extremeprogramming.org/rules.html.

[3] Martin Fowler. *Continuous Integration.* URL: http://www.martinfowler.com/articles/continuousIntegration.html.

[4] buildout.coredev. *Essential Continuous Integration Practices.* URL: http://buildoutcoredev.readthedocs.org/en/latest/continous-integration.html.

[5] MSDN. *Using TDD with ASP.NET MVC.* URL: https://msdn.microsoft.com/en-us/library/ff847525.

[6] Martin Fowler. *GivenWhenThen.* URL: http://martinfowler.com/bliki/GivenWhenThen.html.

[7] Stig Andersen. "Continuous Delivery". In: *Prosabladet* 12 (2015), pp. 16–25.

[8] Yegor Bugayenko. *Why Continuous Integration Doesn't Work.* URL: http://devops.com/2014/09/26/continuous-integration-doesnt-work/.

[9] Michael Reibel Boesen et al. *02131 Assignment 3: Implementation of an ECG co-processor.* English. 2014.

[10] MSDN. *Entity Framework Code First Migrations.* URL: https://msdn.microsoft.com/en-us/data/jj591621.aspx.

# A Kodeeksempler

```
<connectionStrings>
    <add name="SkillMatrixContext"
    ↪ connectionString="Server=tcp:kompetencematrix.database.windows.net,1433;
    Database=kompetencematrix_db;
    User ID=kmapp@kompetencematrix;
    Password=xxxxxxxxxx;
    Trusted_Connection=False;
    Encrypt=True;
    Connection Timeout=30;"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Kodeeksempel 16: connectionString som den er tilføjet i `Web.config`

```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Skills",
            c => new
                {
                    ID = c.Int(nullable: false, identity: true),
                    Title = c.String(),
                    Category = c.String(),
                })
            .PrimaryKey(t => t.ID);

    }

    public override void Down()
    {
        DropTable("dbo.Skills");
    }
}
```

Kodeeksempel 17: Indholdet af *InitialCreate.cs*

```
public class EmployeeController
{
    SkillMatrixContext db = new SkillMatrixContext();

    public void Create(Employee employee)
    {
        ...
        db.Add(employee)
        db.SaveChanges();
    }
```

Kodeeksempel 18: Brug af ens *context*-klasse inden implementering af *repository*-pattern

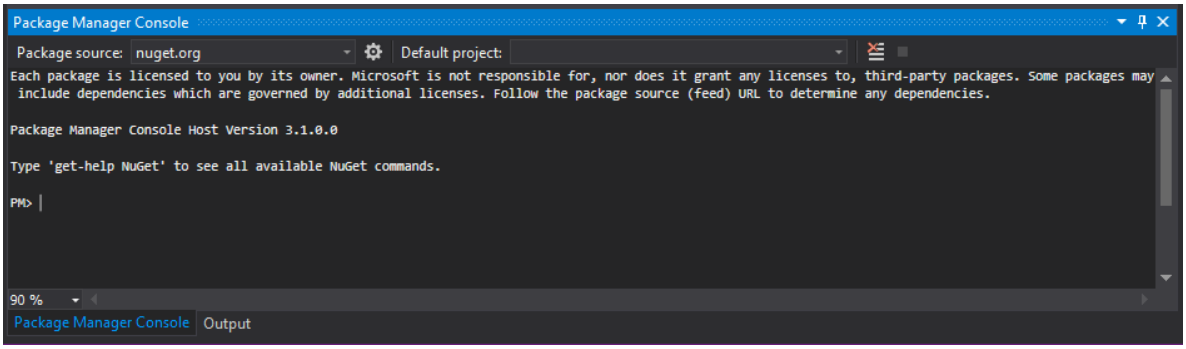
```
public ActionResult Index(string searchString = null)
{
    if (!string.IsNullOrEmpty(searchString))
    {
        return View(employeeRepo.GetQueriedEmployees(searchString));
    }
    else return View(employeeRepo.GetAllEmployees());
}
```

Kodeeksempel 19: Implementering af søgning på en medarbejder som det ses i *EmployeeController*-klassen

```
public IEnumerable<Employee> GetQueriedEmployees(string searchString)
{
    var employees = from e in GetAllEmployees()
                    select e;
    if (!string.IsNullOrEmpty(searchString))
    {
        employees = employees.Where(e =>
        ↪ e.FirstName.ToLower().Contains(searchString.ToLower())
                    ||
                    ↪ e.LastName.ToLower().Contains(searchString.ToLower()));
    }
    return employees;
}
```

Kodeeksempel 20: Implementering af søgning på en medarbejder som det ses i *EmployeeRepository*-klassen

# B Figurer




Figur 21: Package Manager Console