Aarhus University School of Engineering

Funktionel programmering i webudvikling

E6PRJ Study Course

Diplomingeniøruddannelse I Softwareteknologi

Projektrapport

Udarbejdet af:

Christopher Thomas William Bernold – 201611828

Vejleder:

Henrik Bitsch Kirk

Omfang:

15000 Tegn med mellemrum

# Ordforklaring

|  |  |
| --- | --- |
|  |  |
| FP | Funktionel programmering |
| UI | User Interface |
|  |  |

# Indholdsfortegnelse

[Introduktion 5](#_Toc51758888)

[Problemformulering 5](#_Toc51758889)

[Forslag på løsning 5](#_Toc51758890)

[Krav 6](#_Toc51758891)

[Scoping 6](#_Toc51758892)

[User Stories 6](#_Toc51758893)

[Epic 1 – Authentication 6](#_Toc51758894)

[Epic 2 – View Tasks 7](#_Toc51758895)

[Epic 3 – Task oprettelse og generel funktionalitet 7](#_Toc51758896)

[MoSCoW 8](#_Toc51758897)

[Must 8](#_Toc51758898)

[Should 8](#_Toc51758899)

[Could 8](#_Toc51758900)

[Won’t 8](#_Toc51758901)

[FURPS+ 9](#_Toc51758902)

[Functionality 9](#_Toc51758903)

[Usability 9](#_Toc51758904)

[Reliability 9](#_Toc51758905)

[Performance 9](#_Toc51758906)

[Supportability 9](#_Toc51758907)

[Arkitektur 10](#_Toc51758908)

# Introduktion

Dette projekt vil kom til at omhandle hvordan funktionel programmering kan anvendes i web-udvikling. Her vil der overvejes om funktionel programmering kan bruges som erstatning eller blive mere anvendt til at udvikle diverse hjemmesider med færre fejl og om udviklingstiden er hurtigere frem for det meget anvendte imperativ programmerings paradigme. Projektet som udføres, bliver skrevet i sproget Scala som tilbyder muligheden for både objekt orienteret programmering og funktionel programmering. Der vil blive brugt tre hoved værktøjer ovn på Scala, her Twitter’s Finagle, som er et remote procedure call system, parret sammen med et rent funktionel bibliotek Finch som bruges til at udvikle API’er og et tyndt, lightweight lag ovn på finagle. Derudover bruges Play Framework til at udvikle et mere traditionelt objekt orienteret hjemmeside. Begge hjemmesider vil være en To-Do web-app, hvor der vil være samme funktionalitet og udseende.

# Problemformulering

Implementering af to hjemmesider med samme funktionalitet skrevet i henholdsvis objekt orienteret programmeringsparadigmet og funktionel programmeringsparadigmet, sekundært at performance teste de to programmeringsparadigmer og sammenligne udviklingstiden for disse hjemmesider. Funktionaliteten af API ‘en vil inkludere to-do end points, hvor data vil blive persisteret.

# Forslag på løsning

Ved at anvende funktionelt programmering til at udvikle en hjemmeside forekommer der ikke de typiske pitfalls som opstår i imperativ programmering, da der ikke længere vil være state-changes. Ved at bruge et purely funktionelt bibliotek til at udvikle forretnings-logikken øges udviklingstiden i det at der ikke burde opstå så mange fejl.

# Krav

## Scoping

Følgende afsnit vil beskrive projektets funktionelle og ikke funktionelle krav ved brug af User Stories og MoSCoW til at beskrive de funktionelle krav og FURPS+ til at beskrive de ikke funktionelle krav. Eftersom projektet udføres af en person vil der i høj grad være mange funktionelle krav i could og won’t delen af MoSCoW’en for at give mulighed for at der kan tilføjes ekstra funktionalitet til projektet hvis der er tid til det.

## User Stories

User Stories beskriver enkelte situationer fra brugerens perspektiv hvor hvert user story er en del af en større helhed kaldet Epics. Stiftede user stories, bruges herefter til at danne nogle ”features” som skal fremgå i systemet. User Stories følger typisk følgende struktur:

*Som en <type bruger>, vil jeg kunne < udføre et mål> så <begrundelse>.[[1]](#footnote-1)*

Ved at få styr på alle de features som skal fremgå i systemet, kan man efterfølgende trække de ikke funktionelle krav ud af hvert user story.

### Epic 1 – Authentication

#### Story 1 – Log ind

Som en bruger vil jeg kunne logge ind i system med min personlig e-mail så jeg kan finde mine tasks på enhver enhed.

*Godkendelseskriterier*: Brugeren kan logge ind i systemet med deres personlig e-mail.

#### Story 2 – Log ud

Som en bruger vil jeg kunne logge ud af systemet da jeg er færdig med mit arbejde.

*Godkendelseskriterier*: Brugeren kan logge ud af systemet og deres session bliver slettet.

### Epic 2 – View Tasks

#### Story 1 – Overblik af opgaver

Som en bruger vil jeg have en tavle så jeg kan få en overblik over alle mine tasks.

*Godkendelseskriterier*: Brugeren kan se deres personlig overbliks tavle.

#### Story 2 – Flytning af opgaver

Som en bruger vil jeg kunne flytte mine tasks rundt på min tavle så jeg ved hvor langt jeg er i min proces.

*Godkendelseskriterier*: Brugeren kan flytte rundt på deres tasks på tavlen.

#### Story 3 – Fælles tavle

Som en bruger vil jeg kunne dele min tavle med andre så vi kan have nogen fælles tasks.

*Godkendelseskriterier*: Brugeren kan dele sin tavle med andre brugere, hvor andre brugere kan interagere og tilføje tasks til tavlen.

Story 3 – Task beregning

Som en bruger vil jeg kunne holde styr på hvor længe jeg har været i gang med en opgave så jeg ved hvor længe det tager at blive færdig.

*Godkendelseskriterier*: Alle brugerens tasks har en ”tracker” som holder øje med hvor længe de har været i gang med opgaven.

### Epic 3 – Task oprettelse og generel funktionalitet

#### Story 1 – Oprette Task

Som en bruger vil jeg kunne oprette en ny task så den kommer ind på min tavle.

*Godkendelseskriterier*: Brugeren kan oprette en task som automatisk bliver sat ind på deres personlig tavle.

#### Story 2 – Task Beskrivelse

Som en bruger vil jeg kunne give min task et navn og beskrivelse samt dato den skal være færdig så jeg kan genkende den.

*Godkendelseskriterier*: Brugeren kan give en task en beskrivelse.

#### Story 3 – Redigere task

Som en bruger vil jeg kunne redigere min task så jeg ikke skal oprette en ny hvis jeg skal lave små ændringer.

*Godkendelseskriterier*: Brugeren kan ændre en tasks oplysninger/beskrivelse.

#### Story 4 – Slet task

Som en bruger vil jeg kunne slette min task hvis jeg ikke skal bruge den alligevel eller hvis jeg har gennemført opgaven.

*Godkendelseskriterier*: Brugeren kan slette tasks.

## MoSCoW

MoSCoW er en metode som bruges til at definere de vigtigste funktionelle krav og dem som enten ikke passer til det nuværende scope eller ikke skal klares øjeblikkeligt. MoSCoW består af fire punkter: **M**ust have, **S**hould have, **C**ould have og **W**on’t have.

Alle punkter i MoSCoW analysen er taget direkte fra alle user stories for at skabe nogle klare funktionelle krav som kan testes når systemet er færdigudviklet. Disse funktionelle krav beskrives på engelsk for at følge Must, Should, Could og Won’t strukturen.

### Must

The system **must** have an overview of all tasks.

The system **must** be able to move a task from in progress to done.

The system **must** be able to create tasks.

### Should

The system **should** allow tasks to be deleted

The system **should** allow editing tasks

### Could

The system **could** be able to track how long it takes to finish a task.

The system **could** be able to create tasks with a description field.

### Won’t

The system **won’t** allow users to share their task board with other users.

The system **won’t** have a log in system

## FURPS

For at definere alle de ikke funktionelle krav, bruges FURPS. Her kan det være muligt også at definere funktionelle krav, dog vil FURPS modellen/akronymet bruges til at definere alle krav som ikke er defineret i MoSCoW analysen. FURPS står for: **F**unctionality, **U**sability, **R**eliability, **P**erformance og **S**upportability[[2]](#footnote-2), hertil kommer **+** som er brugt til at definere ikke funktionelle krav som umiddelbart ikke passer til de andre punkter i FURPS. Denne opgave kommer primært til at omhandle performance, derfor vil der komme til at være mere kød på dette punkter frem for nogle af de andre FURPS+ punkter.

### Functionality

Tasks bliver persisteret når de bliver oprettet i en midlertidig datastruktur.

### Usability

API’en giver et JSON-respons tilbage.

### Reliability

Der skal ikke opstå nogen fejl i den funktionelle hjemmeside.

### Performance

Den funktionelle hjemmeside regnes med at være hurtigere end den imperativ hjemmeside.

Sikre at 100 kan være på serveren samtidig og samtidig kan behandle JSON requests (skriv mere).

### Supportability

API’erne kan tilgås af enhver system.

# Valg af teknologi

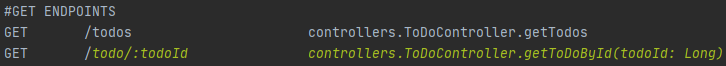
For at kunne teste både funktionelle og imperativ programmerings teknikker har det været nødvendigt at finde et sprog som kan begge dele. Her er der en del sprog som kan bruges da de er fleksible nok til at understøtte flere programmeringsparadigmer. Her er der f.eks. C# som har LINQ som bruges til lambda udtryk og andre FP-teknikker.[[3]](#footnote-3) Her er det oftest samme princip i andre sprog som primært anvendes til imperativ programmering, at der skal bruges et højere abstraktionslag/bibliotek for at kunne bruge FP. For at undgå at bruge meget krudt og undersøgelse på hvordan man mest effektivt kan bruge FP i disse sprog, bruges sproget Scala til dette projekt i det at sproget natively supporter funktionelt programmering og objekt orienteret programmering. Scala er JVM baseret og er udbygget af mange principper som Java understøtter, hvor Scala prøver at forbedre aspekterne af Java som bliver anset som gentagende, restriktivt mm. [[4]](#footnote-4)

Sammen med Scala bruges to web frameworks, Finch[[5]](#footnote-5) og Scala Play Framework[[6]](#footnote-6). Her bruges Finch da det er en purely funktionelt framework, som er nødvendigt i dette projekt da ingen erfaring før er opnået med funktionel programmering, hvilket sikre at alle brugte principper faktisk er FP baseret. Scala Play Framework bruges til det imperative del af projektet, hvor at både funktionelle og ikke funktionelle principper kan udnyttes. For at give det bedste sammenligning af de to paradigmer prøves der kun at anvende imperative principper i Play.

# Implementering

### Finch

Når man udvikler et API findes der naturligvis endpoints. Når man arbejder med endpoints i et standard imperativ MVC projekt, har man typisk en routing module, hvor der er ”tilkoblet” en funktion som udføres når man tilgår endpointet. Det given eksempel vises for hvordan dette gøres i play projektet:



Endpoints i Finch er en del anderledes i det at man specificerer Endpoint[A] dette kan repræsenteres som en lambda funktion: Input => EndpointResult[A]

Input er en datatype som wrapper en Finagle HTTP request med noget finch relateret kontekst

EndpointResult[A]er en algebraic data type (ADT), som er en måde at strukturere data på, hvor der er to cases som indikere om en endpoint matcher en given input eller ej.

Her virker EndpointResult[A] som Option[(Input, Output[A])], dette betyder at hvis inputtet(routen) ikke eksister returneres Skipped som medfører til en 404 HTTP status kode, og hvis at inputtet eksisterer bliver Matched returneret, herefter finder den endpointet og udfører business logikken. Altså hvis en endpoint er matched, bliver den passed til endpointet og business logikken udføres, ellers returneres en 404 Status kode. Her er Option[(Input, Output[A])] en optional tuple. I Scala understøttes NULL værdier, men er ukonventionelt og alle steder hvor NULL kan opstå, bruges Option typen i stedet for. Hvis at en man tager Option[A] returneres enten Some(a)eller None, som kan sammenlignes med imperativ programmering at en værdi enten kan være tilstede eller være null.

For at fuldt ud implementere et endpoint i Finch er der nogle generelle steps som skal udføres:

1. Definere en funktion
2. Hvad for en datatype at requestet skal transformes til i endpointet
3. Specificere HTTP request typen, routen , http headers, parametre og eventuelt body parsing
4. Udføre business logik
5. Serve endpointet:

Følgende kode snip viser hele opsætningen af et endpoint:

**val** *postedToDo*: Endpoint[IO , Item] = jsonBody[UUID =>Item].map(\_(UUID.*randomUUID*()))  
  
*//Create todos Endpoint***def** addToDo: Endpoint[IO, Item] = post(**"todo"** :: *postedToDo*){todo: Item =>  
 **val** newToDo = *InMemoryStorage*().add(todo)  
 Ok(newToDo)  
}

**def** service: Service[Request, Response] = Bootstrap  
 .serve[Application.Json](*addToDo*)

.toService

I eksemplet ses det at *InMemoryStorage*().add(todo) Bliver kaldt. Her er InMemoryStorage et singleton objekt som bruges til at instantiere klassen med alt funktionalitet som udføres i finch projektet.

**object** InMemoryStorage{  
 **private val** *storage*: StorageFunctionality = **new** Storage()  
 **def** apply(): StorageFunctionality = *storage*}

Add funktionen er meget simpelt og blot bruges til at tilføje et todo objekt til en mutable datatype, ListBuffer. For at se på mere interessante tilfælde hvor funktionelle programmerings typer anvendes ses på ”update” funktionen. Her fungerer endpoint opsætningen for update/edit funktionen helt på samme måde.

**override def** update(id:UUID, item: Item): Option[Item] =  
 **for** {  
 *//Zips this iterable collection with its indices.  
 //High order function:* n <- *db*.zipWithIndex.find { **case** (x, s) => x.id == id }.map(\_.\_2)  
 \_ <- *Try* {  
 *db*.update(n, item)  
 }.toOption  
 newItem <- get(item.id)  
 } **yield** newItem

Her er der nogle funktionelle metoder i spil. Update funktionen er først og fremmest en pure funktion. Dette betyder at funktionen altid returnerer de samme værdier for de samme inputs og at der ikke er nogen states eller side effects i funktionen. Ved at specificere at typen er Optional returneres enten Some(Item) eller None, hvilket betyder at Option er en monad.

n <- *db*.zipWithIndex.find { **case** (x, s) => x.id == id }.map(\_.\_2)

Kode snippet for ovn er en bruger ”find” som er en high order function. En high order function er blot en funktion som enten tager en funktion som input eller returnerer en funktion. Derudover ses det at .map(\_.\_2) bliver kaldt efter find funktionen, som returnerer en Option, dette er typisk en indikator af at en functor er i spil (*her tages udgangs punkt i en liste, men kan være andre typer som fx Option, Either mm.*).

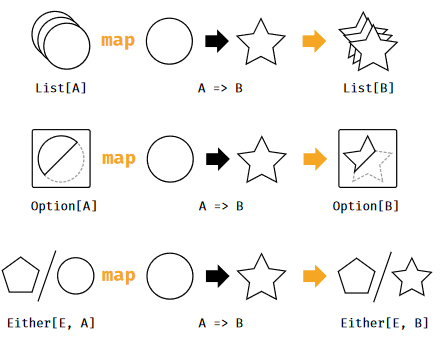


Figure 1 - Mapping over List, Option and Either[[7]](#footnote-7)

Hvis man har en map, men ingen mulighed for at flatten via en flatMap, ender man med en functor. I det given tilfælde bruges der også ”<-” Som er syntaktisk sukker for en flatMap. Dette betyder at hele linjen er en monad:

n <- *db*.zipWithIndex.find { **case** (x, s) => x.id == id }.map(\_.\_2)

Monads er generelt en svær ting at anvende og forstå og i Scala er de lidt specielle. Dette er fordi der faktisk ikke findes en monad keyword som så, men i stedet for bruges flatMap, som er en ”flattened map”. Ved brug af en map alene får man f.eks. får individuelle lister, og med en flatMap ”samles hele listen” og man ender med en lang liste af værdier, eller en samlet todo objekt. En monad består af en unit og en flatMap, hvor en monad kan defineres på følgende måde:

* def unit: A → F[A] 🡨 Creates a monad from an object of type A
* def map: F[A] → (A → B) → F[B]
* def flatten: F[F[A]] → F[A]

Alle wrappers in Scala hvor man kan anvender unit og flatMap operationer er *essentielt* en monad. [[8]](#footnote-8) Dvs. at List, Option, Either mm. er en Monad.

Hvad er en pure function

Bliver currying brugt?

Singleton pattern er brugt, men hvordan og hvad er det??

Hvis der er ting du mangler at skrive om til projektet, så skal du perspektivere til projektet

### Play

### Udviklingstiden

En af de sværeste hurdles ved at bruge funktionelt programmering i Finch og Scala generelt er hvor strict compileren er. Her er det meget ofte at man får Type mismatch fejl i mens at man skriver koden, som gør at implementations hastigheden sænkes ekstremt meget. Her kan man argumentere at det både er en god og dårlig ting, da man får et mere robust system, men uden meget erfaring er det svært at få lavet rigtig meget kode især når man skal lære et nyt sprog og to frameworks.

# Test

Endnu en del af projektet var at performance test begge API’er og finde frem til hvad for en programmerings paradigmer giver de bedste resultater. Her er der mange dele der skal tages stilling til og resultaterne vil aldrig være helt troværdige i det at de to frameworks/biblioteker som er blevet brugt. Finch er meget light weight ift play, hvilket kan have stor betydning for performance test resultaterne. Under forundersøgelser til projektet, blev der fundet frem til at Finch også kunne håndtere flere JSON responses pr sekund ift. Play[[9]](#footnote-9).

Performance-test resultater her!

# Evaluering

#### Fordele og Ulemper ved funktionel programmering versus imperativ programmering

Udviklingstiden, oplevelser, hastighed, performance Measurements.

Er det meget svært at flytte fra imperativ programmering til funktionel programmering?

Er det så simpelt from at have et andet mindset, eller kræver det faktisk mere omtanke?

# Konklusion

# Fremtidig arbejde

# IMPORTANT LINKS

How to implement finch todo  
<https://hackernoon.com/todo-application-using-finch-and-twitterserver-52fa08318c87>

HOW FAST IS FINCH COMPARED TO PLAY:

<https://kostyukov.net/posts/how-fast-is-finch/>

Serving static content in finch   
<https://github.com/finagle/finch/issues/347>

1. <https://www.mountaingoatsoftware.com/agile/user-stories> 23-09-2020 [↑](#footnote-ref-1)
2. <https://businessanalysttraininghyderabad.wordpress.com/2014/08/05/what-is-furps/> [↑](#footnote-ref-2)
3. <https://docs.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming> [↑](#footnote-ref-3)
4. https://www.toptal.com/scala/why-should-i-learn-scala [↑](#footnote-ref-4)
5. https://github.com/finagle/finch [↑](#footnote-ref-5)
6. https://www.playframework.com/ [↑](#footnote-ref-6)
7. https://www.scalawithcats.com/dist/scala-with-cats.html - Sektion 3.1 figure 1 [↑](#footnote-ref-7)
8. https://medium.com/free-code-camp/demystifying-the-monad-in-scala-cc716bb6f534 [↑](#footnote-ref-8)
9. <https://kostyukov.net/posts/how-fast-is-finch/> [↑](#footnote-ref-9)