Indhold

[Indledning 1](#_Toc10033996)

[Løsnings-afgrænsning 2](#_Toc10033997)

[Problemformulering 2](#_Toc10033998)

[Opgaven 2](#_Toc10033999)

[Backend – Web-Api og DataStore 3](#_Toc10034000)

[Projekt struktur 3](#_Toc10034001)

[InMemory DataObjects 3](#_Toc10034002)

[Beskrivelse af DataModellen 3](#_Toc10034003)

[EntityFramework – Implementering 6](#_Toc10034004)

[Repository Pattern 8](#_Toc10034005)

[Web-Api Controllers 10](#_Toc10034006)

[DataTransferObjects 10](#_Toc10034007)

[PostMan kald 15](#_Toc10034008)

[Klient Applikation – MVC WebApp 15](#_Toc10034009)

[Konfiguration 16](#_Toc10034010)

[Design og Bootstrap 17](#_Toc10034011)

[HttpClient-Service 17](#_Toc10034012)

[Controllers, Views og Models 19](#_Toc10034014)

[Diskussion 21](#_Toc10034015)

[Findings 21](#_Toc10034016)

[Generisk BaseRepository 21](#_Toc10034017)

[Mapning 21](#_Toc10034018)

[Mangler 21](#_Toc10034019)

[JWT-sikkerhed 21](#_Toc10034020)

[Unit og Integrations-test 22](#_Toc10034021)

[Konklussion 22](#_Toc10034022)

[Github 22](#_Toc10034023)

[Anvendte værktøjer 22](#_Toc10034024)

# Indledning

I løbet af kurset har jeg selvfølgelig overvejet, hvad mit afsluttende projekt skulle være, og hvilket fokus i backend´en jeg skulle have. Jeg har længe gået med et ønske om at udvikle en budget-app forvaltning af brugerns månedtlige rådighedsbeløb, og jeg fandt det derfor fordelagtigt at tage udgangspunkt i et projekt jeg brænder lidt for 😊

Så når jeg har skulle tage stilling til hvilke teknologier or patterns jeg ville vise i dette afslutningsprojekt, har jeg rigidt ladet budget-app´ens behov diktere implementering, frem for at forsøge at dokumenterer faglig-bredde.

I det følgende vil jeg skitsere projektets fra oprindelig ide til et MVP (Minimum Viable Product).

Ide:  
På væggen på min arbejdsplads står der at Bing Crosby engang skulle have sagt:

*”At kende sine udgifter…? Det er let! Man tager bare sine indtægter og lægger ti procent på.”*

Drevet af tilbagevenden irritation over måned efter måned at have brugt mere end ønsket, udarbejde jeg og fruen et budget efter metodikken:

1. hvad er det vi gerne vil have råd til i løbet af året (ferier, tøj, gaver osv)
2. hvad VED vi, at vi har af gentagne udgifter år efter år
3. er der noget vi vil spare sammen til

For de faste udgifters vedkommende, har de fleste af os en budgetkonto i banken hvor alle transaktioner registreres med relativt sigende data, og det er således let at overskue hvad pengene går til.

Det kan man derimod ikke sige om rådighedsbeløbets anvendelse. Jeg definerer rådighedsbeløb som hvad der er tilbage til dagligt forbrug efter de faste udgifter er betalt.

Det udarbejdede budget var og er meget nuanceret og overskueligt, men det er jo komplet umuligt at holde rede på i hovedet, og tanken om, hver aften at skulle løbe dagen igennem og registrerer ind- og især udgifter er langt fra gangbar.

Løsningen må derfor være en mobile-app, hvor transaktioner let kan registreres umiddelbart efter, at de er foretaget. At let tilgængelighed og lethed i anvendelse er et must, siger sig selv, men problematikken rangerer under UX og er i sagens natur derfor ikke en del af denne opgave.

I mit daglige virke skriver jeg Angular og NativeScript, altså udelukkende frontend, der datamæssigt bygget på json-responses fra diverse web-api. I denne sammenhæng er websites baseret på MVC Core relativt uinteressante, ikke meget Razor, men masser af javascript. Derimod er Asp.Net Core Web-Api og Entity Framework ‘et overordentligt interessante, og har afgjort været mit fokus i denne opgave. Jeg har dog i et sekundært projekt oprettet en MVC baseret klient website

Opgaveformuleringen fordrer dog at jeg viser MVC funktionalitet, og jeg har derfor oprettet en .net Core MVC-webapplikation, hvis formål alene er at vise data fra kommunikation med WebApi´et og dermed databasen.

# Løsnings-afgrænsning

Optimalt består løsningen af tre moduler:

1. *Budget Modul*

Funktionalitet, der gør det muligt at fastlægge et budget

1. *Registrerings Modul*

Hvor brugeren registrerer, hver gang dankort eller kontanter anvendes

1. *Rapporterings modul*

Hvor brugeren får indsigt i ”hvordan det går”

For dog at afgrænse opgavens scope, har jeg valgt pt. 2 – Registrerings modulet, først og fremmest fordi dette modul kan give værdi i sig selv, også under fravær af Budget- og Rapporterings modulet. De eneste krav er at have fastlagt en datamodel der understøtter sammenligning af budget og faktiske udgifter. I princippet kunne man sige, at Registrerings modulet skal overholde den kontrakt, Budgettet definerer.

# Problemformulering

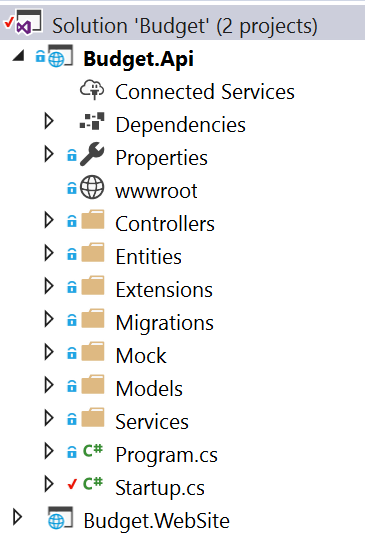
* Hvordan kan jeg bygge et web-api til håndtering af data-udveksling mellem klient-applikation og database.
  1. Hvordan kan man faciliterer kommunikation mellem webapi og database v.hj.a EntityFramework Core
  2. Hvordan kan Single Responsibility Principle understøttes, og hvordan adskilles implementering og data-kommunikation ved brug af Repositories
     1. Hvordan bruges repository-pattern til at muliggøre ’switch’ imellem forskellige data lagre
  3. Hvordan kan en webapplikation hente og vise data fra et dotnet Core WebApi v.hj.a. Microsoft´s MVC framework modelbinding
  4. Hvordan udfoldes de basale Entity-modeller i passende Data Transport Objekter
  5. Er Attribute routing en acceptabel routing løsning

# Opgaven

I det følgende beskriver jeg først Backend´s implementering af WebApi, Repositories og EntiryFramework, og dernæst gennemgår jeg klient MVC Core Webapplikationen

## Backend – Web-Api og DataStore

### Projekt struktur



### InMemory DataObjects

Initielt i forløbet valgte jeg, at lave en POC på en CRUD-implementeringen i web-api funktionaliteten, og startede derfor med at udvikle direkte mod et in-memory dataobjekt der overholder Entity-modellen. Dermed kunne jeg abstrahere EntitiyFramework og database-problematikker ud af POC´en, og alene koncentrere mig om implementeringen.

### Beskrivelse af DataModellen

Som sagt skal løsningen udstille mulighed for at en bruger kan registrerer sine løbende udgifter og holde dem op mod et udarbejdet budget, og selv om min afleverings-opgave ikke inkluderer budget funktionalitet, er budgettets egen datastruktur dikterende for registrerings-modulets datainterfaces indhold.

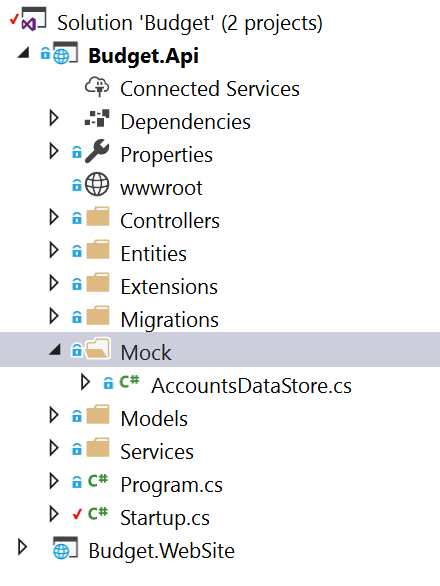
Således arbejdes der med tre indbyrdes linkede data-tabeller:

1. Account: overordnet budget-kategori
   1. Eksempler: ferier, jul, gaver, tøj, hushold
2. SubAccount: sekundær budget-kategori
   1. Eksempler: ferier/sommerferie, ferier/1-uges vinterferie, ferier/forårsferie
3. PostingLines – transaktionslinie, den egentlige registrering

Objekt Hierarki: Account -

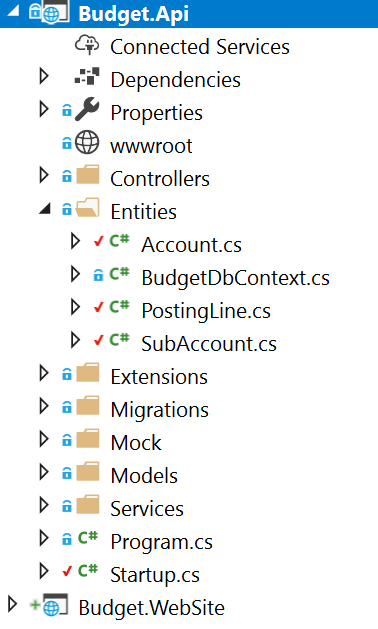
En Account kan altså indeholde flere SubAccounts, som igen kan indeholde flere PostingLinier.

Eksempel fra AccountDataStore (*seedes fra statisk property i AccountDataStore.cs*)

#### DataBasen

Data af struktur som Account-objektet vist herover, er bedst repræsenteret i en relationel database. Hv vior tabellerne Account, SubAccount og PostingLines er indbyrdes relateret således at SubAccount tabellen viser sit tilhørsforhold under Account ved implementering af et AccountId ligesom en PostingLine indeholder en reference til SubAccountID.



Entity Modeller

|  |  |
| --- | --- |
| Account |  |
|  | Indeholder en generisk ICollections af typen SubAccounts |
| SubAccount |  |
|  | Indeholder reference til parent account ”ForeignKey”  Og ikke mindst en lazyloaded udgave af selv samme parent Account-objekt  Desuden indeholder SubAccount også en generisk ICollection indeholdende objekter af typen PostingLine |

|  |  |
| --- | --- |
| PostingLine |  |
|  | Det sidste – og i objekthierakiet: nederste – Entity-objekt PostingLine, der repræsentere den egentlige registreringslinie  Hvad har du købt, for hvor meget og evt. hvor  Desuden registreres tidspunktet for transaktionen  Sidst men ikke mindst indeholder PostingLine en lazy-loaded udgave af registreringens parent-SubAccount, via ForeignKey: SubAccountId |

### EntityFramework – Implementering

Den trænede lytter vil her notere sig, at objektstrukturen vist herover har bemærkelsesværdig lighed med EntityFrameworkets data-objekter, og det er også lige hvad det er.

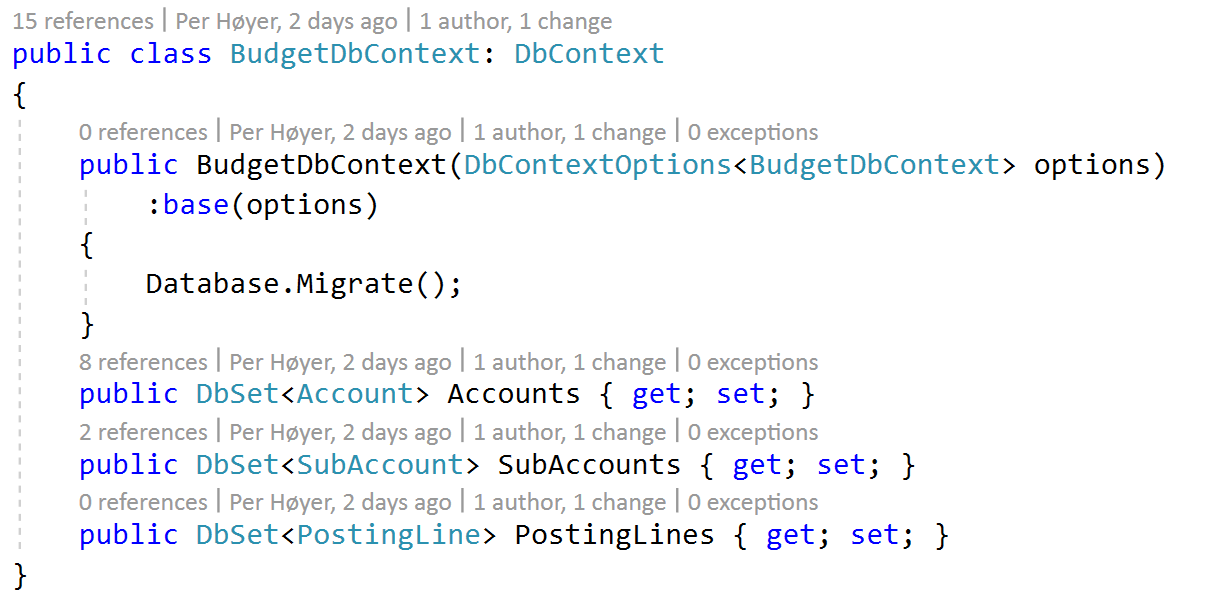
Som udgangspunkt for databasen har jeg valgt en CodeFirst approach. Det betyder, at jeg alene med ovenstående EntityModeller kan genererer database inklusiv relationer med EF´s migration, der også gør det muligt, at reflektere ændringer i modellen direkte i databasen med kommandoerne: *add-migration* og *update-database.*

For at ovenstående skal virke, skal EF selvfølgelig vide, hvor databasen skal residere, og hvad den skal hedde. Dette gøres i connection-strengen, hvor også det ønskede databasenavn ’*BudgetDb*’ angives:

var connectionString = @"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=BudgetDb;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

Det er klart at ovenstående connection-string skal ændres såfremt det ønskes at hoste databasen fx på winAzure SQL.

Ud over selve connectionstringen er det nødvendigt at vise hvordan EntityModellerne skal oprettes i databasen. Dette gøres i forbindelse med, at det der kendes som Database Context´en dannes. Dette angives for denne løsning i filen BudgetDbContext.cs



Bemærk udtrykket Database.Migrate i constructoren, der sikre at databasen dannes/opdateres skulle det være nødvendigt, når en given bruger forsøger at tilgå sitet.

Det sidste der mangler er så at fortælle dotnet Core løsningen hvad connection-strengen faktisk er og hvilken type database server der skal anvendes. Det gøre i startup.cs filen i ConfigureServices-metoden

services.AddDbContext<BudgetDbContext>(o => o.UseSqlServer(connectionString));

Og det er alt der skal til. Den eneste kommando nødvendig er *add-migration,* med angivelse af migrationens navn. Update-Database-kommandoen kan også anvendes, men er faktisk ikke nødvendig når Database.Migrate() kaldet er tilføjet til DbContext filens constructor.

I forbindelse med deploy fx til winAzure, er den eneste nødvendige en opdateret connection-streng opdateret efter forholdende.

Sidst men ikke mindst ønskede jeg at fylde databasen med en mindre sæt data. Dette har jeg opnået ved at oprette en extension-metode til min databaseContext: BudgetDbContext

public static void EnsureSeedDataForContext(this BudgetDbContext context){}

Og heri med conditionen

if (context.Accounts.Any())

{

return;

}

Sikre at seed-data kun populeres til databasen hvis den er fuldstændig tom (*contex.Accounts.Any()*)

Selve seed-data repræsenteres med et objekt der fuldstændig ligner det det jeg som udgangspunkt brugte til InMemory Objektet – dog er Id-property´en fjernet, eftersom databasen med auto-encremented index automatisk generere fortløbende Id´s

Objektet skrives til databasen v.hj.a. kommandoen

context.Accounts.AddRange(accounts);

context.SaveChanges();

Det eneste jeg mangler for at danne databasen og populere den med de ønskede data er, at kalde extension-metoden i startup-filens Configure-metode med:

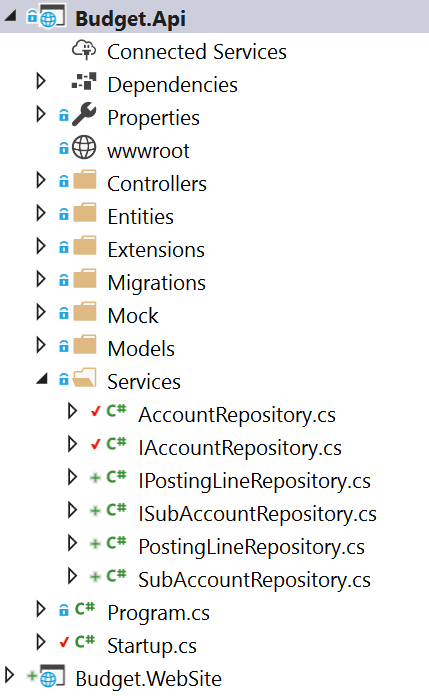
budgetDbContext.EnsureSeedDataForContext();

Og voilá… databasen er oprettet og indeholder et minimalt sæt af data

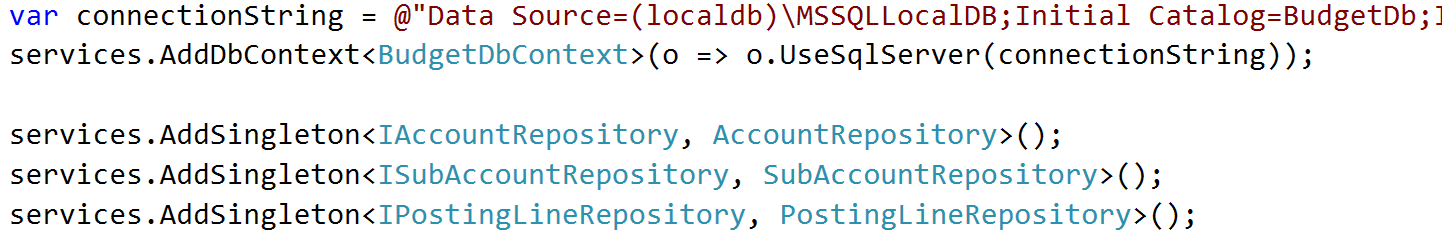
### Repository Pattern

Eftersom jeg allerede har to datalagre i spil, både et InMemoryObjekt og - med EF også - en SQL server, blev det klart, at det ville være fordelagtigt at kunne isolere den logiske implementering i selve API´et fra egentlige data-operationer mod databasen for let at kunne skifte mellem de to (og måske senere at kunne vælge andre typer af datalagre, uden at skulle ændre implementeringen).

Jeg har derfor valgt at lægge repositories ind mellem selve api´et og databasen. Metodikken er kendt som Repository Pattern.



Repositories implementeres i løsningen som services og skal derfor registreres under metoden ConfigureServices() i startup.cs filen, for at gøres klar til at blive injected i fra web-apiets controllers.



Repository-services implementere et interface, der for Gud og hvermand definerer, hvilke metoder repository’et udstiller. I denne løsningen implementeres et traditionelt sæt af CRUD metoder (Create-Read-Update-Delete), samt enkelte hjælpe-properties, som fx. Property: *bool* *AccountExists { get; set; }*

Det skal her nævnes, at jeg har valgt at lave tre repositories, ét til hvert af de tre Entity-objekter. Jeg mener nu at dette var en fejl. Til at begynde med havde jeg flere funktionaliteter udstillet i de forskellige interfaces, i princippet var jeg i gang med at udviske grænsen imellem logisk implementeret og data-operationer, hvilket fik mig til at trimme interfaces´ene gevaldigt. Og… efter trimning fremstår de tre repositories funktionelt fuldstændig ens, og alt i alt burde jeg have lavet ét fælles generisk repository.

|  |  |
| --- | --- |
|  | Nederste udkommenterede linie:  Et eksempel på funktionalitet der svæver på grænsen mellem implementering og data-operationer  Bemærk den boolske includeSubAccount (se næste afsnit, LazyLoading) |
|  | .. og her:   * includePostingLines |
|  |  |
|  | BudgetDbContext, det centrale EF-objekt |

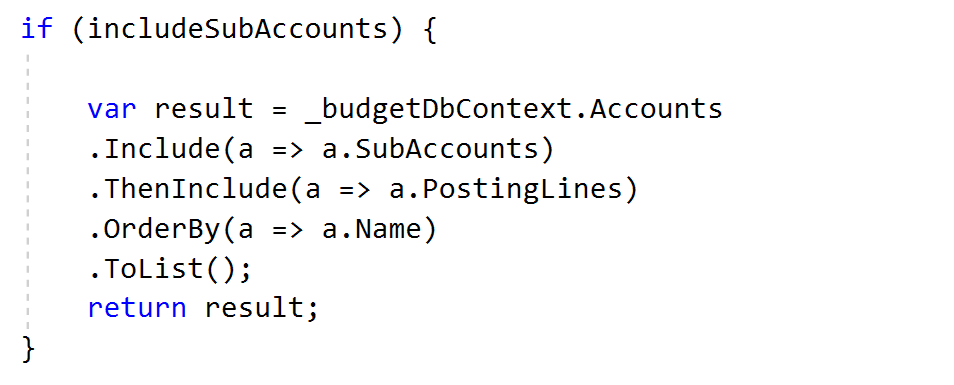
#### Implementering af CRUD – LINQ – LazyLoading

Jeg har brugt LINQ til at håndtere kommunikation med EF, og dermed databasen. Jeg satte ud for at undersøge Entity Frameworket, og støder i den forbindelse på begrebet Lazy-loading: objekter loader først når de forespørges.

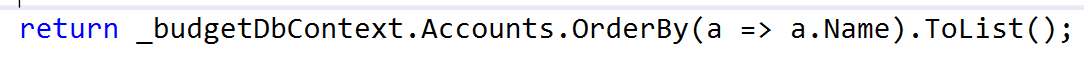
Det fik mig til at undersøge hvordan linq kan bruges til at hente data for lazyloadede entity-objekter henholdsvis IKKE at gøre det. Jeg viste tidligere variablene *includeSubActions* og *-PostingLines.* Disse muliggør fra kilenten at bestemme hvorvidt man ønsker child-collections fyldt eller ej.

Namespace´et Microsoft.EntityFrameworkCore giver os med metoderne *Include()* og *ThenInclude()* mulighed for at hente sub-lists i så mange niveauer man måtte ønske.

HttpGet metoden *GetAccounts()* med *Include()* og *ThenInclude()* kan aflevere det fulde datasæt :



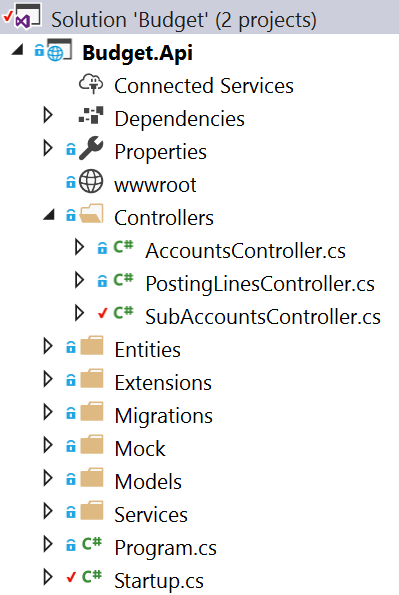
Alternativt at aflevere listen af Accounts uden sub-collections



### Web-Api Controllers

Efter således at have registreret de repository-services, der varetager kommunikation til og fra EntityFramework´et, skal den egentlige logik implementeres, og jeg vælger her at undersøge asp.net Core frameworkets WebApi-Controllers.

Når kravene til Controleren skrives tages der udgangspunkt i en faktisk klients behov . I mit tilfælde er klienten en web-app, hvis behov igen dikteres af den forventede slutbruger. Dermed kan man vel betragte dette Api som en BFF (Backend For Frontend), der jo i øvrigt – som repositories´ene – er isoleret fra klienten og dermed kan levere data til en hvilken som helst klient-app, der anvender en http-klient til data-kommunikation. (MobileApp, AngularApp, PWA (Progressive Web App)



### DataTransferObjects

DataTransferObjects – eller DTOs – beskriver dataobjekter som forholder sig til hvilke data-modeller Api´et forventes at loade og arbejde med på en given klient requests. De data-modeller der bruges her kan adskille sig betydeligt fra de kedelige 😊 entity objekter.

Det kan selvfølgelig diskuteres hvorvidt man skal anvende eet stort DTO, der indeholder alt, hvad man måtte skulle bruge i en lille løsning som denne, men jeg har valgt at undersøge pro/contras af at etablere mange DTOs

|  |  |
| --- | --- |
|  | De store:   * AccountDto * SubAccountDto * PostingLineDto   Til Create-PostRequests   * AccountForEditingDto * SubAccountForEditingDto * PostingLineForEditingDto   De små   * AccountWithOutSubAccountsDto * SubAccountsWithoutPostingLinesDto   Med sub-collection   * AccountWithSubAccountsDto * SybAccountWithPostingLinesDto |

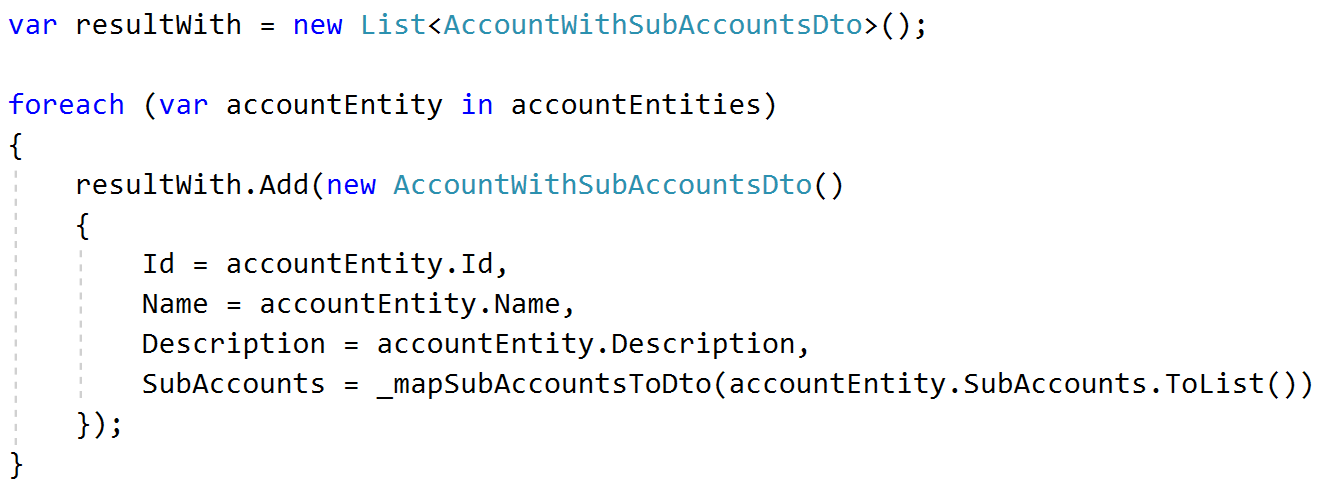
Ekesmpler:

|  |  |
| --- | --- |
|  | Ud over listen af SubAccounts indeholder AccountDto´en   * NumberOfSubAccount * Total (kun stub)   Der begge kan betragtes som pladsholdere for ekstra udregninger eller andet der afviger fra den originale entity-model |
|  | I den anden ende af spektret ser vi her DTO modellen for Create Account, der kun udstiller de to felter, der er nødvendige for at oprette en ny account: *Name* og *Description*  Eftersom denne model ultimativt genererer en write til databasen, er properties´ene som det ses dekoretet med forskellige restriktive annotationer |
|  | SubAccountDto, der også bruges af AccountsDto |

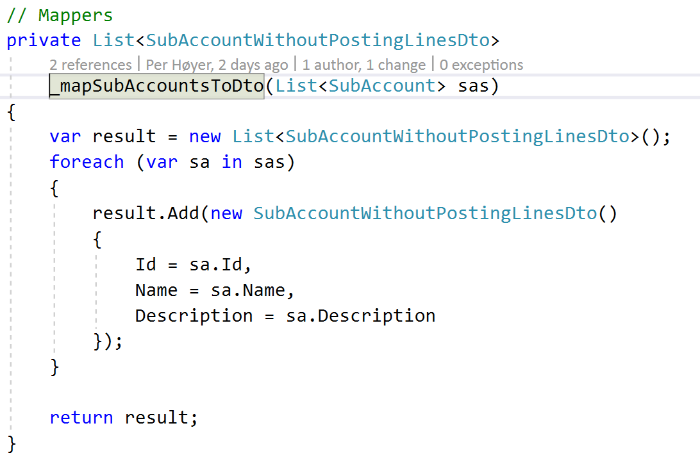
#### Mapning:

Med så relativt store forskelle mellem entity- og dto modellerne er en del mapning selvfølgelig nødvendig. I projektet har jeg valgt ikke at bruge ekstrene NuGet pakker eller andet til dette, og Controllerne indeholder således private mapping-metoder

Mapning af fra Entity Account til Dto AccountDto



Mappning af liste af SubAccounts



Jeg må dog konstatere at mappningen er omfangsrig og særdeles triviel, så fremadrettet vil jeg nok vælge fx AutoMapper til håndtere mapning.

#### Routing

Jeg har for at følge anbefalingen for routing på WebApi valgt at bruge Attribute baseret routing, det betyder også at jeg ikke har defineret routes i Startup-klassens Configure-metode, men slet og ret implementerer app.UseMvc();

Jeg har i forbindelse hermed undersøg hvorvidt attribute-baseret routing er den rette løsning til en løsning som denne. Lad os se på routes´ene:

I controller-klasser i projektet er dekoreret med en base-route:

[Route("api/accounts")]

Hvilket betyder at løsningen kan tilgås som *http(s)://domainname/api/account*

AccountsController

* **[HttpGet()]**

Url: *http(s)://domainname/api/account*

* **[HttpGet("{id}", Name="GetAccount")]**

Url: *http(s)://domainname/api/account/1*

Bemærk property´en Name i HttpGet dekorationen, der anvendes som reference til ResponseObjektet (se anvendte ResponseCodes)

* **[HttpPost()]**

Url: *http(s)://domainname/api/account*

igen kald direkte på ovenstående url, om end denne gang som Post, der indeholder payloadet til at oprette en ny Account

* Url: **[HttpPut("{id}")]**

*http(s)://domainname/api/account/1*

* Url: **[HttpDelete("{id}")]**

*http(s)://domainname/api/account/1*

SubAccountsController

* [HttpGet("{accountId}/subaccounts")]

Url: *http(s)://domainname/api/account/1/subaccounts*

* [HttpGet("{accountId}/subaccounts/{id}", Name ="GetSubAccount")]

Url: *http(s)://domainname/api/account/1/subaccounts/1*

* [HttpPost("{accountId}/subaccounts")]

Url: *http(s)://domainname/api/account/1/subaccounts*

* [HttpPut("{accountId}/subaccounts/{id}")]  
  Url: *http(s)://domainname/api/account/1/subaccounts/1*
* [HttpDelete("{accountId}/subaccounts/{id}")]  
  Url: *http(s)://domainname/api/account/1/subaccounts/1*

PostingLinesController

* [HttpGet("{accountId}/subaccounts/{subAccountId}/postinglines")]  
  Url: *http(s)://domainname/api/account/1/subaccounts/1/postinglines*
* [HttpGet("{accountId}/subAccounts/{subAccountId}/postinglines/{id}", Name="GetPostingLine")]  
  Url: *http(s)://domainname/api/account/1/subaccounts/1/postinglines/1*
* [HttpPost("{accountId}/subaccounts/{subAccountId}/postinglines")]  
  Url: *http(s)://domainname/api/account/1/subaccounts/1/postinglines*
* [HttpDelete("{accountId}/subaccounts/{subAccountId}/postingLines/{id}")]  
  Url: *http(s)://domainname/api/account/1/subaccounts/1/postinglines/1*

Routing parametrene *includeSubAccounts* og *includePostingLines* angives I url´en som query-parametre og kaldes med:

*?indludeXXX = true/false*

Ovenstående url´s følger Micorsofts anbefalinger for konfiguration af route-navigation for Asp.Net Core WebApi, som jeg altså har fulgt, men jeg tænker et jeg ikke nødvendigvis er overvældende begejstret, sådan at forstå at især url´erne til sub-sub objektet PostingLines bliver meget lange og komplicerede. Det går selvfølgelig i en begrænset POC som denne, men i et større live api med potientielt mange flere [HttpGet] dekorerede metoder.

I forhold til den større løsningen, med både Budget- og Rapporterings modul i aktion, kan jeg godt være bekymret for overskueligheden af denne attribut-baserede routing

#### Anvendte ResponseCodes

200 Ok  
Anvendes når apiet finder data at aflevere. IActionresult implementerer en Ok() der returnerer et HttpResponse med status: 200. Tilmed kan et objekt sendes med tilbage i Request body som parameter til Ok-metoden:

*Ok(model);*

404 Not Found  
IActionresult implementere også en NotFound() metode

400 Bad Request  
IActionresult implementerer også en BadRequest()

En fejl 400 signalere at request-objektet ikke er i orden. Validity check af indkomne request-objekter foregår i AstNet Core v.hj.a ModelState.IsValid. True hvis property-values validerere mod data-anotationerne på det givne model-objekt (Required, MaxLength etc), og indeholder endda custom error-messages som: ”*Navn skal udfyldes*” og hvis ikke en custum ErrorMessage er angivet indeholder ModelState default fejlbeskederne. Dermed er objektet fordelagtigt at sende med tilbage til klienten, og med BadRequest(ErrorState)

201 Created

return CreatedAtRoute("GetAccount", new {id=finishedAccount.Id}, finishedAccount);

som det ses en lidt anderledes retur-funktion 201 Created bør returnere det nyoprettede Id fra databasen samt det objekt der blev sendt til databasen. ”GetAccount” sender brugeren til til før viste route:

**[HttpGet("{id}", Name="GetAccount")]**

Hvor det nyoprettede object så kan betragtes og nydes.

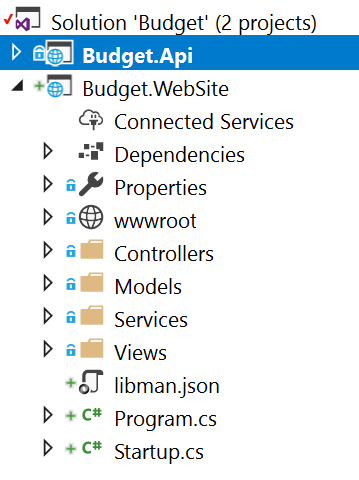
204 NoContent

Som er den anbefalede returværdi på update og delete

### PostMan kald

## Klient Applikation – MVC WebApp

Opgavens opdrag indeholder også at vise modelbinding til Views v.hj.a. MVC og jeg har derfor tilføjet en almindelig MVC baseret webapplikation, der tjener som klient.



Da jeg som sagt er angular-programmør i mit daglige virke, er det nok usandsynligt, at jeg i mit videre arbejde med BudgetApp´en ville vælge Asp.Net Core MVC type projekt til min klient, jeg har derfor valgt ikke at bruge tid på nuanceret Razor programmering, om end jeg viser, hvordan data bindes med modeller fra Controller til View og omvendt.

Klienten vil primært modtage og arbejde med de Dto objekter vi sender til/modtager fra WebApiet. Klienten kender altså ikke implementeringen af databasen og Entity modellerne

### Konfiguration

Konfigurationen af denne Asp.Net Core applikation er lidt anderledes end den relativt simple konfiguration vi så i WebApiet. Eftersom denne del af løsningen tjener som den egentlige snitflade mod slutbrugerne, skal der tages højde for f.eks. design og kommunikation af fejlmeddelelser, ligesom routing mellem de forskellige views skal sættes op

Som før registreres MVC funktionaliteten i Startup-filens ConfigureMethods:

services.AddMvc();

ligesom MVC skal registreres som middleware for brugerrequests. Jeg har valgt at gøre det med metoden

app.UseMvcWithDefaultRoute();

Normalt ville jeg registrere og konfigurere mine Routes i UseMvc(), men Asp.Net Core tilbyder mig en metode der uden yderligere konfiguration tilbyder standard routing på formen

{controller=Home}/{action=Index}/{id}

Overholdes formen virker Routing out of the box. Således har jeg oprettet en HomeController med en Index-metode der peger på et Index-view

Endvidere har jeg oprettet en *\_ViewImports.cshtml* der implementerer MVC´s default taghelpers:

@addTagHelper\*, Microsoft.AspNetCore.Mvc.TagHelpers

TagHelpers muliggør navigation på formen

*<a asp-controller=”ControllerClassName” asp-action=”ActionMethodName”> Til Action</a>*

Der af MVC Core oversættes til det gængse html link:

*<a href=”controller/action>Til Action</a>*

Og evt. med routing-parametre som:

<**a** **asp-controller**="Accounts" **asp-action**="Details" **asp-route-id**="@item.Id">@item.Name</**a**>

Der oversættes til:

*<a href=”controller/action/id”>text</a>*

### Design og Bootstrap

For hurtigt at etablere et acceptabelt UI og lette designfasen har jeg besluttet at implementere BootStrap. Jeg har brugt LibMan til at hente og inkludere Twitter-BootStrap i projektet. Libman installere de relevante css og js filer i wwwroot-folderen. For at tilgå content herfra er det nødvendigt at bruge

*app.UseStaticFiles()*

i Startup.cs filens Configure-metode

Jeg anvender en layoutpage \_layouts.cshtml, der tjener som den oveordnede designside. I layouts.cshtml har jeg implementeret bootstrap her i header-sektionen.

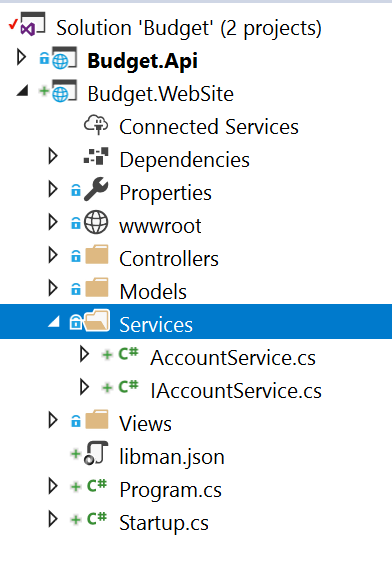
<link href="~/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />

Jeg har så brugt bootstrap-container som container for alt indhold, med undtagelse af menuen.

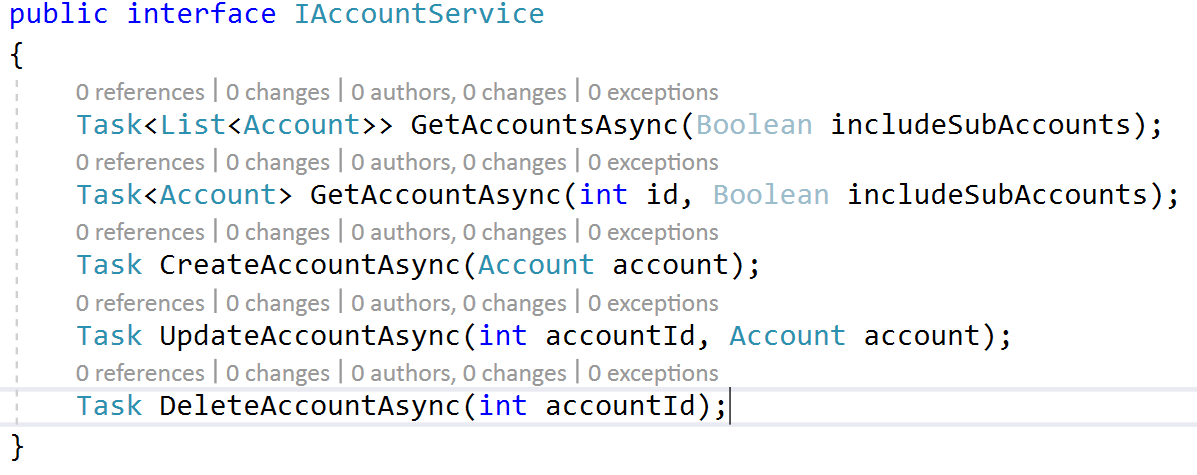
De enkelte views i løsningen kan så implementerer \_Layouts.cshtml, og få glæde af designet, for at undgå manuelt at skulle implementerer siden på alle views, har jeg implementeret en Views\\_ViewStart.cshtml, der netop indeholder en reference til Layouts.cshtml, og loades af Core-frameworket umiddelbart før et hvilket som helst view aktiveres.

### HttpClient-Service

Jeg kunne selvfølgelig kalde mit Api direkte fra min MVC controller, men for at overholde det første SOLID princip, Separation of Concerns, har jeg valgt at lægge kommunikationen med API´et i en service, gennem hvilken Controllerens action metoder kan opnå adgang til data fra WebApiet.

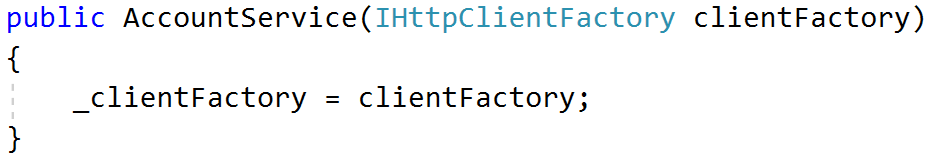


Interfacet afspejler her de samme kald, der går igennem hele applikationen, altså for så vidt også et CRUD interface.



Bemærk asynkroniteten, eftersom kaldet til Api´et kan tage noget tid er det vigtigt at ”vente” på responset før interfacet loades og viser data.

Eftersom servicen via http skal kontakte WebApiet er det nødvendigt i servicen at implementerer en klient, der kan håndtere HttpRequests og -Responses. Jeg har valgt at bruge et IHttpClientFactory som jeg injecter i servicens constructor:



#### Service-metoder

Eftersom interfacet kræver asynkronitet tasks annoteres metoderne i servicen med ’async’

### 

Etablering af http-klienten sker i linien var client = \_clientFactory.CreateClient();, og herefter kan klienten kontakte apiet på dettes url:

var response = await client.SendAsync(webapi-url);

Bearbejdning af responset sker kun såfremt servicen modtager en positiv 200 Ok statuskode fra Api´et. Dette gøres i linien: if (response.IsSuccessStatusCode)

Hvor responset afventes og læses:

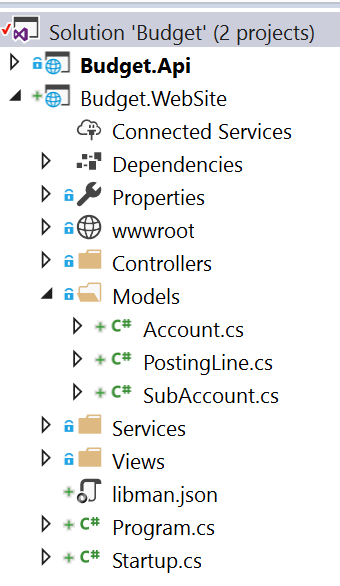
var result = await response.Content.ReadAsAsync<List<Account>>();

For at gøre servicen tilgængelig for resten af løsningen, specifikt controllerne, gøres AccountServicen tilgængelig for dependency injection i startup.cs´s ConfigureServices metode med linien:

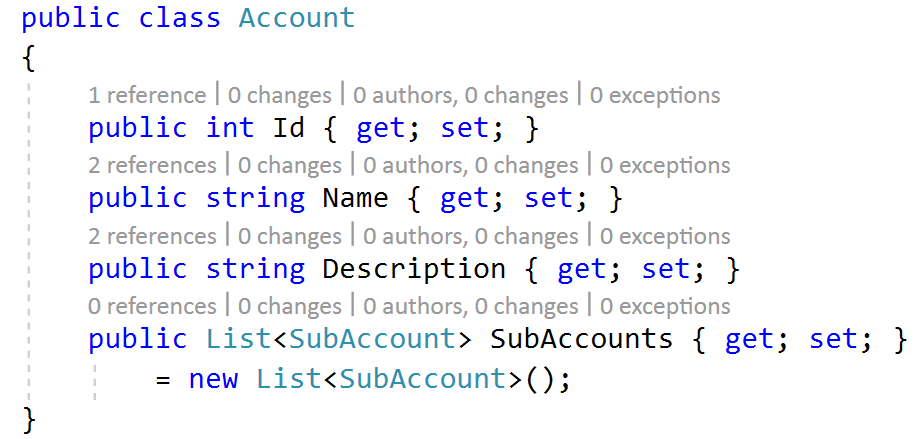
services.AddSingleton<IAccountService, AccountService>();

### Controllers, Views og Models

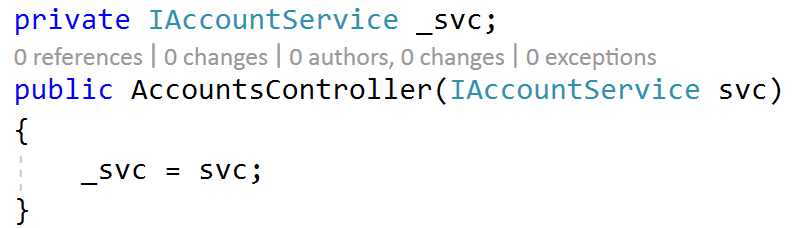
For at implementerer MVC (Model View Controller) er det selvfølgelig nødvendigt også i klient-applikationen at have modeller der afspejler de data der skal vises i UI´et.



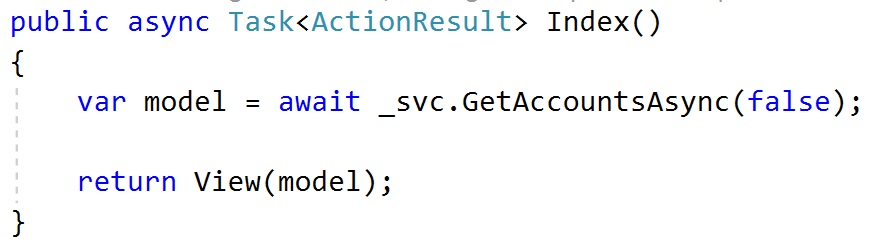
De typer der aflevers fra apiet er WebApi projektet defineret i DTO´erne, og pendenter til disse bør etableres i klient-applikationen også:



I Controllerne injectes de respektive services i constructoren med:



Controllerens enkelte metoder er som det var tilfældet i klient applikationens service også asynkrone

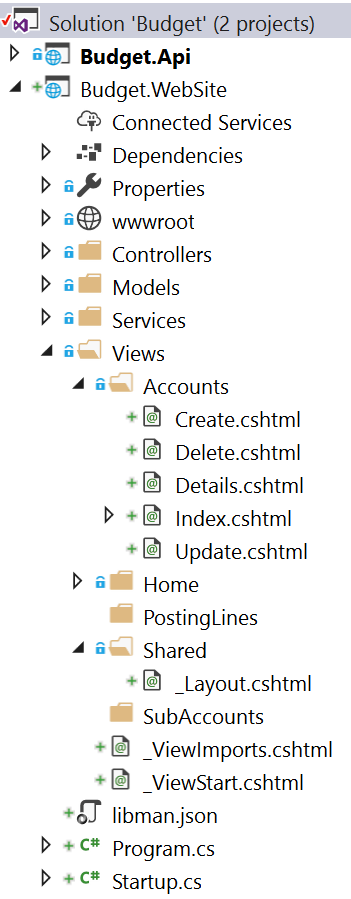


De annoteres med async og resultat fra servicen awaites med:  
var model = await \_svc.GetAccountsAsync(false);

Modellen – automatisk mappet til klient modellen sendes så til viewet med:

return View(model);

#### Views



Selve Views´ene har jeg valgt at scaffolde, og ikke koncentreret mig synderlig om design og ui blot accepteret resultatet af scaffoldingen.



# Diskussion

I det følgende vil jeg diskutere hvilke findings, jeg har haft i forbindelse med udvikling af budget-appen. Jeg satte som opdrag til mig selv at skrive alt i hele løsningen (med undtagelse af de scaffoldede views) fra bunden. Dvs jeg har for stort set alt taget udgangspunkt i et tom klasse, jeg så selv har konfigureret som hhv. controller, repository osv.

Den del er lykkedes ganske glimrende synes jeg – jeg har etableret et api som – om end i en simpel version – gør det muligt at bygge en klient applikation der kan håndtere data fra Accounts, SubAccounts og PostingLines

Processen – skriv alt selv – har selvfølgelig også givet anledning til rynkede øjenbryn, og ”… arh, dét må kunne gøres lettere” tanker 😊

## Findings

### Generisk BaseRepository

Først og fremmest, hvis de forskellige repositories ligner hinanden, og det vil vel ofte være tilfældet for et CRUD, bør man om muligt samle de enkelte Entity-modellers repositories i eet, eller i hvert tilfælde lade dem arve fra et Generisk Base Repository

### Mapning

Mapning er cumbersome, og på sigt bør der anvendes hjælpe pakker som fx AutoMapper. Når jeg alligevel har valgt den lange vej skyldes det dels en tids-parameter, dels at jeg med den manuelle metode har bedre overblik over hvad der sker – men senere, i en live løsning er der ingen tvivl om at jeg ville bruge AutoMapper fremadrettet

## Mangler

### JWT-sikkerhed

Som API’ et fremstår lige nu er det helt åbent og ikke underlagt nogen form for sikkerhedscheck, hvilket selvfølgelig er uholdbart. Der er potientielt tale om mange brugers budget- og forbrugsdata, som selvfølgelig skal være beskyttet, således at man kun har adgang til sine egne data.

Jeg har kigget en del på JWT-bearers, men nåede ikke at få implementationen funktionel, så jeg har valgt - for nu – at fjerne den igen.

### Unit og Integrations-test

Test, test, test – alle taler om det ingen gør det 😊 … Selvfølgelig noget sludder, naturligvis burde jeg have omsat test for både repositories og controllers i webapiet, ligesom især mapningen burde testes. Det er imidlertid ikke nået, og jeg har undladt at forsøge delvis implementering.

### Konklussion

Som sagt var mit forehavende først og fremmest at undersøge Asp.Net Cores implementering af WebApi. For at være sikker på ikke bare at acceptere autogenereret fuktionalitet satte jeg mig for at skrive alt fra bunden ( igen undtaget de scaffoldede views i klient applikationen ), og den del er lykkedes udmærket. Jeg har bragt mig selv i stand til fremadrettet at kreere de api´er mine front-end løsninger måtte kræve.

Jeg har vist at man kan faciliterer kommunikation mellem web api og database v.hj.a Entityframework Core

Jeg er lykkedes med at oprette og implementere brug af Repositories og set at jeg skal skifte mellem forskellige datalagre

Jeg har undersøgt og vurderet anvendelsen af attribute-baseret route-navigation

Og, jeg har oversat Entity modeller til relevante Dto´s

Efter bedste overbevisning er det således lykkedes mig at undersøge problematikkerne fra min problemformulering

### Github

Jeg har med Visual Studios GitHub klient etableret et repository indeholdende hele løsningen på:

<https://github.com/MonsterGameDev/Budget>

### Anvendte værktøjer

Visual Studio 2017 med Asp.Net Core 2.1.1

Entityframework Core 2.1.1