Aarhus Universitet
Semesterprojekt 4
Gruppe 5

Implementering

CarnGo

Gruppemedlemmer:
Edward Hestnes Brunton
(201705579)
Marcus Gasberg
(201709164)
Martin Gildberg Jespersen
(201706221)
Mathias Magnild Hansen
(201404884)
Tristan Moeller
(201706862)
Erik Mowinckel
(20107667)
Hamza Ben Abdallah
(201609060)

Vejleder:
Jesper Michael Kristensen, Lektor

29. maj 2019

1 Versionshistorik

Ver.	Dato	Initialer	Beskrivelse
1.0	27/04/2019	MG	Tilføjet introduktion af dokumentet
1.1	28/04/2019	MG	Tilføjet Frameworks til dokumentet
1.2	8/05/2019	MG	Tilføjet Patterns til dokumentet
1.3	15/05/2019	MG	Tilføjet Strategy Pattern til dokumentet
1.4	22/05/2019	MG	Tilføjet kodeudsnit til dokumentet
1.5	25/05/2019	MG	Opdateret i forhold til factories
1.6	25/05/2019	MG	Fjernet SOLID.tex og gjort underafsnit af patterns

Indhold

2 Introduktion

I dette afsnit præsenteres de overvejelser, der har været omkring implementeringen af systemet. Det er her valgene omkring, eksterne afhængigheder præsenteres, samt hvordan de gavner udviklingen af systemet. Derudover præsenters nogle af de valg, der har været omkring eventuelle patterns osv. under implementeringen. Til at starte med præsenteres de frameworks, der er brugt til at udvikle systemet, herefter de afhængigheder, der er til andet software og til sidst de anvendte patterns.

3 FrameWorks

I dette afsnit præsenteres de eksterne software biblioteker, der er blevet anvendt til at udvikle systemet, som er skrevet i C sharp version 7.1, samt begrundelsen for at inkludere dem i projektet.

3.1 Prism

Prism er et open source framework, der hjælper til udviklingen af XAML applikationer så det opfylder det MVVM arkitektur pattern. Der er i dette projekt benyttet prism Core version 7.1.0.431 [1].

3.1.1 Begrundelse

Da størstedelen af dette projekt omhandler udviklingen af en WPF applikation, giver det god mening at inkludere et framework, der laver en masse boiler plate kode, man ellers selv skulle skrive. Når man arbejder med MVVM er det strengt nødvendigt at anvende Commands, hvilket man får givet gennem dette framework. Derudover giver frameworket også en mulighed for kommunikation mellem ViewModels.

3.1.2 Anvendelse af biblioteket

Fra frameworket anvendes der primært den DelegateCommand-klasse, der gør det muligt at definere og binde Commands til XAML viewet. Kommandoerne inkludere både en CanExecute, der gør os som udviklere i stand til at bestemme, om kommandoen kan udføres, samt en ObservesProperty funktion, der gør kommandoen i stand til at overvåge en property.
Derudover anvendes EventAggregator også, som den primære form for kommunikation mellem ViewModels. Den gør at man kan samle alle events, der går mellem ViewModels, som f.eks. en søgning fra HeaderBarViewModel, der skal sendes til SearchViewModel, på et sted. Denne anvendes i sammenspil med Unity, for at sikre at alle ViewModels får den samme instans af EventAggregator.

3.2 Unity

Unity er en Dependency Injection Container, også kaldet en Inversion Of Control Container, der faciliterer lav kobling i applikationen. Unity giver mulighed for at simplificere instantieringen af nye objekter, så andre klasser ikke skal. Der er i dette projekt benyttet Unity version 5.10.3[2].

3.2.1 Begrundelse

Når man bygger en MVVM applikation, så er der mange Dependencies, man skal kende til, samt hvordan man tilgår dem og anvender dem. Fordelen ved at bruge en Dependency Container er, at man som udvikler ikke skal tænke særlig meget over, hvordan man får fat i de nødvendige objekter.

3.2.2 Anvendelse af biblioteket

Det sted, hvor det gavner allermest i applikationen er ved oprettelsen af en ny side/view. Når der oprettes en ny side, så skal den tilhørende ViewModel oprettes, samt dens afhængigheder skal løses. Oprettelsen af en side/view med tilhørende ViewModel kan ses i listing 1.

Listing 1: Oprettelse af en side med tilhørende ViewModel

```
1  public class BasePage<TViewModel> : BasePage
2          where TViewModel : BaseViewModel
3      {
4          private TViewModel _pageViewModel;
5
6          public BasePage()
7          {
8              PageViewModel = IoCContainer.Resolve<TViewModel>();
9          }
10
11         public TViewModel PageViewModel
12         {
13             get => _pageViewModel;
14             set
15             {
16                 if (_pageViewModel == value)
17                     return;
18                 _pageViewModel = value;
19                 DataContext = _pageViewModel;
20             }
21         }
22     }
```

Af listing 1 ses det at der i constructoren af BasePage sættes PageViewModel til IoCContainer.Resolve<TViewModel>(). Det betyder, at man får IoCContainer til at finde ud af hvilke afhængigheder, som den ViewModel har, og få fat i dem alt efter nødvendighed. Det kunne eksempelvis være gennem constructor injection, hvilket kræver at afhængighederne er registreret med IoCContaineren. Det betyder også, at hvis man har de nødvendige klasser registreret rigtigt, så kan man bare putte dem som argumenter i Constructoren, og så frit anvende dem.

3.3 Entity framework core

EF core er et framework, som bruges som en objekt-relationelle mapper. Der er i dette projekt benyttet Entity framework core version 2.2.4[3].

3.3.1 Begrundelse

EF core gør det muligt at arbejde med en SQL database, med objekter og fjerner det meste data access kode, som der normalt skal laves.

3.3.2 Anvendelse af biblioteket

Dette framework bliver brugt til projektets modellag, som giver data acess til de diverse modeller. Et model består entity klasser og context objekter, som gør at man kan bruge queries og gemme data på databasen. Instanser af entity klasser bliver hentet for queries, og der bliver benyttet instanser af entity klasser til at lave, slette, og ændre data.

3.4 NUnit

NUnit er et framework for unit-testing kode for .NET. Der bliver i dette projekt benyttet NUnit version 3.12.0[4].

3.4.1 Begrundelse

Når man har med kode, der hele tiden ændres, så er det en god idé at Unit Teste sin kode, for at sikre at funktionaliteten ikke ødelægges. Derudover er det en god måde at finde fejl i sin kode, samt teste at den gør det, der forventes af den.

3.4.2 Anvendelse af biblioteket

Frameworket er kun inkluderet i Test-projekter, hvilket vil sige Unittest- og Integrationstest-projekterne. Disse projekter gør brug af frameworket til at teste, det kode der er lavet i de andre projekter.

3.5 NSubstitute

NSubstitute er et mocking bibliotek for .NET, der gør det nemt at mocke sine afhængigheder. Det vil altså sige, at man kan teste om afhænighederne bliver kaldt på den rigtige måde. Der bliver i dette projekt benyttet NSubtitute version 4.1.0[5].

3.5.1 Begrundelse

Da der ikke er fokus på, hvor godt man kan skrive mocks til sin kode, så er det ikke tiden og umagen værd at bruge tid på det. Derfor anvendes et bibliotek til at gøre det for os.

3.5.2 Anvendelse af biblioteket

Når man har en afhængighed, som f.eks. noget validering af nogle bruger inputs, så kan man se om objekterne modtager de rigtige parameter. Det kunne f.eks se ud som i listing 2.

Listing 2: Oprettelse af en side med tilhørende ViewModel

```
1   [TestCase("1234")]
2   [TestCase("")]
3   [TestCase("asdf1234")]
4   [TestCase("asdf.!#")]
5   public void PasswordSecureString_SetPropertyTwice_ValidatorOnlyCalledOnce(string password)
6   {
7       var passwordSecureString = password.ConvertToSecureString();
8       _fakePaswwordValidator.ValidationErrorMessages.Returns(new List<string>() { "test" });
9       _fakePasswordMatchValidator.ValidationErrorMessages.Returns(new List<string>() { "test"
            });
10
11      _uut.PasswordSecureString = passwordSecureString;
12      _uut.PasswordSecureString = passwordSecureString;
13
14      _fakePaswwordValidator.Received(1).Validate(Arg.Any<SecureString>());
15  }
```

4 Patterns

I dette afsnit beskrives nogle af de patterns, der er anvendt i projektet, samt en forklaring om, hvordan de er anvendt, hvorfor de er anvendt og hvilke problemer de løser. Men først beskrives solid, der er de principper, som patterns prøver at støtte op om.

4.1 SOLID

SOLID er et akronym for fem vigtige design principper, systemet er designet sådan at implementeringen overholder disse fem principper så godt som muligt[6]. Der er dog visse udfordringer forbundet med dette i sammenhæng med en MVVM arkitektur. Et eksempel på dette er at ViewModels ofte indeholder meget logik, de er svære at opdele og de er ofte offer for ændringer, da ændringer i Viewet ofte skal afspejles i ViewModel.

4.2 Strategy

Strategy pattern bruges når der ønskes, at en algorithm kan vælges runtime[7]. Et eksempel i projektet, hvor dette anvendes er ved animationer. Der laves et interface, som alle animationer skal overholde, som ses i listing 4.

Listing 3: Interface for AnimationStrategy

```
1  public interface IAnimationStrategy
2  {
3      void AddAnimateIn(Storyboard sb, double duration);
4      void AddAnimateOut(Storyboard sb, double duration);
5  }
```

Denne strategy gør at det er muligt at implementere flere animationer, der implementere interfacet, og så tilføje dem allesammen til et storyboard. Et eksempel på dette er i klassen FrameworkElementAnimations, der ses i listing

Listing 4: FrameworkElementAnimation AnimateIn funktion

```
1  public async Task AnimateIn(FrameworkElement frameworkElement, double seconds)
2  {
3      var storyBoard = new Storyboard();
4      foreach (var animationStrategy in _animationStrategies)
5      {
6          animationStrategy.AddAnimateIn(storyBoard, seconds);
7      }
8      storyBoard.Begin(frameworkElement);
9      frameworkElement.Visibility = Visibility.Visible;
10     await Task.Delay((int) (1000 * seconds));
11 }
```

Strategy giver mening i dette eksempel, da man kan specificere, hvilke animationer man vil have på sine elementer runtime, og de herefter anvendes på elementet. Derudover giver løsningen også en nem måde at lave nye animationer og tilføje dem til elementer på en

4.3 Singleton

Singleton pattern er et meget omdiskuteret og et pattern man skal være meget påpasselig med at anvende for ofte[8]. Der er dog steder i vores applikation, vi har set fordele ved at

anvende det. Der er komponenter, hvor det giver mening, kun at have en enkelt instans af i hele applikationen. De komponenter vi kom frem til er:

1. ApplicationViewModel
2. IoCContainer
3. EventAggregator

ApplicationViewModel Dette er modellen, der modellere state på hele applikationen. Lige netop fordi den modellere hele applikationen giver det mening kun at have en af den. Den håndtere altså, hvilken bruger der er logget ind, hvilken side der skal vises i applikationen og nogle få funktioner der er helt basale for applikationen, som login og logout.

IoCContainer Denne container sørger for at initialisere objekter, give de rigtige parametre med i constructoren og sikre levetiden af objekter. Det giver derfor mening at have en instans af denne i applikatione,

4.4 Factory Patterns:

Factory method pattern går ud på at adskille dannelsen af objekter fra bruget af dem gennem et metode kald[9]. Dette koncept bruges i sammenspil med Unity, hvor man f.eks. kan kalde IoCContainer.Resolve<HeaderBarViewModel>() og få lavet et objekt, hvor der er taget hånd om alle afhængigheder af objektet.
Derudover kan man som i Abstract Factories registrere objekter i forhold til et interface, hvilket gør at det er nemt at udskifte en konkret implementering med en anden, så længe det samme interface er opfyldt[10].
Det bedste eksempel på dette, er i BasePage klassen, der står for initialisering af den side der hører til siden. Dette ses i linje 8 af listing 5.

Listing 5: FrameworkElementAnimation AnimateIn funktion

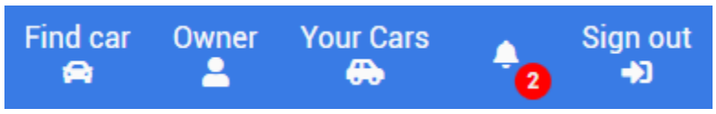
```
1  public class BasePage<TViewModel> : BasePage
2      where TViewModel : BaseViewModel
3  {
4      private TViewModel _pageViewModel;
5
6      public BasePage()
7      {
8          PageViewModel = IoCContainer.Resolve<TViewModel>();
9      }
10
11     public TViewModel PageViewModel
12     {
13         get => _pageViewModel;
14         set
15         {
16             if (_pageViewModel == value)
17                 return;
18             _pageViewModel = value;
19             DataContext = _pageViewModel;
20         }
21     }
22 }
```

5 Layout

Dette afsnit præsenterer de eksterne værktøj, der er blevet anvendt på applikationens GUI.

5.1 FontAwesome

FontAwesome bliver brugt for de forskellige ikoner, som bliver brugt på applikationen, som kan ses på 5.1**fontAwesome**



Figur 5.1: FontAwesome ikoner på navigationsbar, med tekst som har Roboto font

5.2 Roboto

Roboto er den font, som bliver brugt til alt tekst i CarnGo Applikationen**roboto** I billag **Kravspecifikation ?? ??** er der et ikke funktionelt krav, som siger at applikationen skal have en gratis font, og det er Roboto.

Referencer

[1] Prism Library, *Prism*. side: https://prismlibrary.github.io/docs/.

[2] Unity, *Unity*. side: https://github.com/unitycontainer/unity/.

[3] R. Peres, *Entity Framework Core Succinctly*, 2018. side: https://docs.microsoft.com/en-us/ef/core/.

[4] NUnit, *NUnit*. side: https://nunit.org/.

[5] NSubtitute, *NSubstitute*. side: https://nsubstitute.github.io/.

[6] Wikipedia, „SOLID“, 2019. side: https://en.wikipedia.org/wiki/SOLID.

[7] GeeksforGeeks, *Strategy pattern*. side: https://www.geeksforgeeks.org/strategy-pattern-set-1/.

[8] ——, *Singleton pattern*. side: https://www.geeksforgeeks.org/singleton-design-pattern-introduction/.

[9] Dofactory, *Factory Method*. side: https://www.dofactory.com/net/factory-method-design-pattern.

[10] ——, *Abstract Factory*. side: https://www.dofactory.com/net/abstract-factory-design-pattern.