Vitec - Hjemmesideprojekt

Meldgaard, Mike Holst
mike2860@edu.ucl.dk

Rasmussen, Anders Fredensborg
ande714b@edu.ucl.dk

Sandbøl, Kaare Veggerby
kaar1498@edu.ucl.dk

Thomsen, Johannes Ehlers Nyholm
joha321j@edu.ucl.dk

19. december 2019

Indhold

1 Formål

Vi har i dette projekt arbejdet med Vitec MV, som leverer software, der hjælper ordblinde. Lige nu er deres primære kunder skoler og kommuner. De ønsker dog at kunne sælge deres produkter til privatpersoner, da ordblindhed er en lidelse man har gennem hele livet. Derfor har vores opgave været at bygge en platform, der henvender sig til privatpersoner.

Vitec MV ønsker at sælge CDORD og Intowords til privatpersoner. Vores formål var at lave et proof of concept på, hvordan man kan henvende sig til det ønskede kundesegment. Vitec MV vil gerne have, at der fokuseres på brugervenlighed.

2 Hjemmeside

Vi har valgt at prøve at opfylde Vitec MVs krav ved at bygge en hjemmeside, som kan findes på URL'en `http://skilsmis.se/`.

Hjemmesiden er lavet med ASP.NET Core. Fordelen ved ASP.NET Core er, at det giver os Razor Page filformaten .cshtml, hvilket betyder, at vi kan skrive C#i vores HTML. Det betyder, at vi kan lave interaktive hjemmesider uden at skulle skrive Javascript.

3 Arkitektur

ASP.NET Core bruger to lagmodeller: Razor Pages, hvilket desværre har samme navn som filformaten, og MVC. Razor Pages kører med arkitektur mønstret model-view-viewmodel(MVVM), og MVC bruger selvfølgelig arkitektur mønstret model-view-controller.

Begge arkitekturmønstre har fordele og ulemper. Det er meget nemt at overholde designprincippet Single Responsibility i Razor Pages, da en side kun kan have ansvaret for, hvad der skal vises på den side. Derudover er det meget hurtigt og nemt at lave simple hjemmesider, der blot skal lave CRUD på datamodeller. Consensus er dog, at hvis din hjemmeside skal være mere kompleks, som f.eks single page applications, eller hvis den gør brug af en REST api, er MVC bedre at bruge.

Da vi har delt vores hjemmeside op i flere dele:

- En database, der indeholder vores data
- En API, der facilitere kommunikationen mellem database og hjemmeside.
- En hjemmeside, som brugeren tilgår.

har vi også valgt at køre med MVC arkitekturen. For at undgå god classes har vi valgt at have flere controllers, som hver har ansvar for dele af vores hjemmeside. Hver controller har ansvar for de dele af hjemmesiden, som har med én datamodel at gøre.

4 API

Til at håndtere kommunikationen med vores database har vi valgt at gøre brug af en RESTful API. Fordelen ved dette er, at det er nemmere at skifte en database ud, uden at vi skal modificere vores hjemmeside. Derudover betyder det også, at vores hjemmeside er sat op til at kommunikere med en API, hvilket er en fordel, da vi ved at Vitec MV gør brug af en API til deres brugerhåndtering.

I vores hjemmeside har vi flere forskellige datamodeller, som vi henter fra API'en, blandt andre produkter og deres priser. For at undgå at skrive den samme kode flere gange har vi i stedet lavet nogle generisk metoder, der kommunikerer med API'en, en til at hente data fra API'en(HTTPGet request) og en til at modificere data(HTTPPut request).

Vi gør også brug af Swagger i vores API til nemmere at kunne overskue, hvad den indeholder. Swagger giver API'en en side, hvor man kan se, hvilke URL'er API'en indeholder, hvilken type HTTP requests hver URL er, og hvilke JSON objekter, som API'en udbyder. Derudover giver den også mulighed for at API'ens ejer kan skrive sine kontaktinformationer og lignende. Vores implementation af Swagger kan ses på `http://vitecapi.azurewebsites.net/index.html`.

Swagger er måske ikke det mest relevante til vores projekt for Vitec, da deres API ikke er offentligt tilgængelig, og det derfor ikke er så vigtigt at kunne fremvise, hvad den tilbyder. Vi har dog valgt stadig at implementere det, da det var et emne fra undervisningen, som vi gerne ville have lagt helt fast, og det gav os en nem måde internt at overskue vores API.

5 Sikkerhed

En vigtig ting i den moderne verden er IT-sikkerhed. De mest almindelige angrebstyper, som hjemmesider er ofre for er:

- SQL-Injection
- Cross-side request forgery(CSRF)
- Cross-side scripting(XSS)
- Open redirect attack(ORA)

Alle angrebstyperne kan skabe store problemer, hvis man ikke beskytter sig mod dem. Heldigvis tilbyder ASP.NET Core en række værktøjer, vi kan bruge, som gør det sværere at angribe hjemmesiden.

For at undgå SQL-Injection bruger vi Entity Framework(EF) som vores ORM. Fordelen ved det er, at EF altid parametiserer input til databasen, og det derfor bliver meget sværere at lave SQL-Injections, da en SQL server behandler parameter præcis som de står og ikke som en del af en SQL-statement, der skal udføres.

ASP.NET Core tilbyder request tokens, hvilket betyder, at når en bruger først beder om at få siden, hvor brugeren kan modificere eller slette et dataobjekt bliver der også sendt en unik token. Når brugeren så sender sit put eller delete request, bliver denne token sendt med. Hvis tokenen ikke er magen til den, som vores hjemmeside forventer bliver requestet afvist. Det betyder, at en ond

hjemmeside ikke bare kan sende requests efter en bruger er logget ind på vores hjemmeside, da den hjemmeside ikke har vores token. Det kan dog lade sig gøre at opsnappe denne token, så hvis man har med meget vigtig data at gøre, som f.eks en bank, er det en rigtig god idé at bede om at brugeren skal indtaste sit kodeord, inden hjemmesiden behandler brugenes request.

XSS er meget nemt at beskytte sig mod i ASP.NET Core, da næsten alle teksttyper, som vi kan vise i vores HTML automatisk bliver rengjort af ASP.NET Core. I vores undervisning lykkedes det os kun at finde én teksttype, som er svag overfor XSS: HtmlString. Så for at beskytte vores hjemmeside mod XSS undgår vi blot at gøre brug af den teksttype.

Vi beskytter os mod ORAs ved simpelthen ikke at have nogen open redirects på vores hjemmeside i skrivende stund. Hvis vi havde nogen open redirects kunne vi dog beskytte os ved at bruge redirect metoden LocalRedirect. Denne metode behandler den angivne URL og kaster en exception, hvis URL'en ikke er lokal, altså ikke peger på en del af vores hjemmeside.

6 Authentication og Authorization

Noget af det sidste, som vi nåede at implementere på vores hjemmeside var et login system og tilhørende authorization. Dog har vi ikke noget at gøre fuldt brug af det, men har lige nu kun sat authorization på en enkelt del af hjemmesiden, som proof of concept til os selv.

Vi gør brug af policy-based authorization, hvilket betyder, at vi tilføjer en policy under startup af projektet, og for at kunne tilgå en del af hjemmesiden, skal du opfylde den policy. Vores policies er lige nu kun baseret på brugeres roller, da vi kun har 2 brugertyper i form af en kunde og en administrator. Vi har dog valgt stadig at køre det som policy based istedet for role based, da vi mener, at det er nemmere at skalere policy based authorization.