# Fagprøve IT-Utviklerfaget

Rasmus Løvli Netland

13.08.2025 - 21.08.2025

Innholdsfortegnelse

[Fagprøve IT-Utviklerfaget 1](#_Toc206590933)

[Planleggingsdel 3](#_Toc206590934)

[Målet med oppgaven 3](#_Toc206590935)

[Fremgangsmåte 5](#_Toc206590936)

[Skisse av løsningen 7](#_Toc206590937)

[Tidsskjema 8](#_Toc206590938)

[Oversikt over utstyr 9](#_Toc206590939)

[Kostnadsoverslag 10](#_Toc206590940)

[Infrastrukturkostnader 10](#_Toc206590941)

[Engangskostnader 10](#_Toc206590942)

[Arbeidskostnader 10](#_Toc206590943)

[Totale kostnader 10](#_Toc206590944)

[Informasjonskilder og Samarbeidspartnere 11](#_Toc206590945)

[Forenklinger, Avgrensninger og Forbehold 12](#_Toc206590946)

[Dokumentasjonsdel 13](#_Toc206590947)

[Backend 13](#_Toc206590948)

[Teknologier og verktøy 13](#_Toc206590949)

[Helper classes og selv-lagde verktøy 15](#_Toc206590950)

[Endepunkter og Controllere 17](#_Toc206590951)

[Database 20](#_Toc206590952)

[Frontend 23](#_Toc206590953)

[Oversikt 23](#_Toc206590954)

[Detaljert oppbygning 25](#_Toc206590955)

[Problemer som oppstod under Frontend 27](#_Toc206590956)

[Standarer 28](#_Toc206590957)

[Hosting 29](#_Toc206590958)

[Testplan 30](#_Toc206590959)

[Testlogg 31](#_Toc206590960)

[Brukermanual 32](#_Toc206590961)

[Kilder 34](#_Toc206590962)

[Egenvurdering 35](#_Toc206590963)

[Oppsummering 35](#_Toc206590964)

[Utført arbeid og utfordringer 35](#_Toc206590965)

[Avvik i løsning, i forhold til plan eller oppdrag 36](#_Toc206590966)

[Hva kunne jeg gjort bedre 36](#_Toc206590967)

# Planleggingsdel

## Målet med oppgaven

Målet med oppgaven er å utvikle en funksjonell og sikker handleliste-applikasjon som gjør det enkelt for brukere å planlegge, og gjennomføre handleturer. Løsningen skal være tilgjengelig på PC og mobil, slik at den kan brukes hjemme for planlegging og i butikken under handling.

Applikasjonen skal tilby:

* **Brukerhåndtering** – Registrering og innlogging med sikker passordlagring, slik at kun autoriserte brukere får tilgang til egne handlelister.
* **Fleksible handlelister** – Muligheten til å opprette én eller flere lister, legge til, redigere og slette varer.
* **Kjøpsstatus** – En tydelig måte å markere hvilke varer som er kjøpt, eller lagt i handlekurven. Slik at brukeren slipper å tenke over dette selv under en handletur.
* **Sikker behandling av data** - Person- og handledata skal være beskyttet via sikker dataoverførsel (HTTPS), kryptering og sikker databasehåndtering.

Løsningen skal være enkel å bruke for alle, rask å lære, og skal kunne utvides med flere funksjoner i fremtiden om det skulle være relevant.

## Fremgangsmåte

For å nå målet vil arbeidet organiseres i flere faser som bygger på hverandre. Hver fase har tydelige leveranser og milepæler for å sikre fremdrift.

**Fase 1** **– Planlegging og design (Dag 1)**

* Analysere kravene i oppgaven og lage en løsning som dekker alle funksjoner.
* Lage skisser av arkitekturen, databasestrukturen, kommunikasjon og brukergrensesnitt.
* Planlegge databasestruktur.
* Velge rammeverk og hosting for backend og frontend.
* Lage en tidsplan og ressursoversikt.

**Fase 2 – Oppsett av utviklingsmiljø og database**

* Installere og sette opp nødvendige utviklingsverktøy.
  + Visual Studio
  + Microsoft SQL Server Management Studio
  + .NET 8
  + Docker Desktop (for testmiljø)
* Opprette database med tabeller.
* Definere relasjoner og sikre dataintegritet med fremmednøkler.
* Definere Stored Procedures for database operasjoner.

**Fase 3 – Backend Utvikling**

* Sette opp CRUD- endepunkter for oppretting, redigering, sletting og innhenting av:
  + Lister
  + Varer i listene
* Lage registrerings og påloggings endepunkter.
  + Returnerer en «token» ved pålogging, token genereres med «utløpsdato».
  + Token kan caches på klientside, og oppdateres via API for å slippe at brukeren må logge på konstant.
* Implementere sikker passordbehandling, via bruk av salt og hashing.
  + Å bruke salt (tilfeldige karakterer lagt til passordet) øker sikkerheten ved at brukere ikke kan ha samme passord hash.
* Implementere grunnleggende validering av data, og feilhåndtering.
  + Innebakt logger, for å beholde feilmeldinger for analyse. Logges til disk eller Azure ved behov.
* Sette opp Unit tester for å sikre API funksjonalitet ved re-deployement etter potensielt fremtidig arbeid.

**Fase 3 – Frontend Utvikling**

* Implementere registrerings- og innloggingssider med kobling opp mot API.
  + Lagre bruker «token» i ett «cache»
* Lage listeoversikt med mulighet for å opprette nye handlelister.
* Lage visning av varer i liste, med mulighet til å legge til, redigere, slette og markere vare som «kjøpt».
* Gjøre løsningen responsiv for PC og mobil.
* Lage ett brukergrensesnitt som er responsivt, brukervennlig og lett å forholde seg til.

**Fase 4 – Sikkerhet og optimalisering**

* Utnytte HTTPS (SSL- sertifikat) for kryptert og sikker kommunikasjon.
* Forbedre input-validering for å unngå mulig SQL-injection.
* Sikre at API-et kun gir tilgang til brukerens egne data.

**Fase 6 – Testing og dokumentasjon**

* Gjennomføre funksjonstester av alle hovedfunksjoner.
* Brukerteste på støttede enheter.
* Fikse eventuelle feil som oppstår, eller fornedrelser som må utføres.
* Skrive ferdig teknisk dokumentasjon og brukerveiledning.

**Fase 7 – Presentasjon**

* Presentere tjeneste og oppgave for prøvenemda.

## Skisse av løsningen

A diagram of a computer

AI-generated content may be incorrect.

## Tidsskjema

Estimert timebruk 45 timer. Ekskluderer fremføring av fagprøven, og gjennomgang av oppgaven.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Tid | 13.08 | 14.08 | 15.08 | 18.08 | 19.08 | 20.08 | 21.08 |
| 08:30-  09:30 | Se gjennom oppgave med prøvenemda. | Sette opp programvare som trengs | Sette opp autentifiserings system for brukere og API. | Definere basic grensesnitt i frontend. | Ferdigstille «hoved» siden, med lister og varer. | Starte testing av hele systemet. | Fremføre fagprøve |
| 09:30-  11:00 | Planlegging av gjennomføring | Sette opp database, og definere tabeller. |  | Etabler kommunikasjon mellom frontend og backend |  |  |  |
| LUNSJ | LUNSJ | LUNSJ | LUNSJ | LUNSJ | LUNSJ | LUNSJ | LUNSJ |
| 11:30-13:00 | Lage en oversikt over utstyr og kostnader | Definere påkrevde Stored Procedures for database opreasjoner. | Fullføring av API logikk. |  | Finpusse grensesnitt og styling. | Rett opp i evt. feil og forbedre det som behøves. |  |
| 13:00-14:00 | Lage skisser for løsning, og arkitekturtegninger | Sette opp API miljø, definer endepunkter. |  | Ferdigstille registrering og påloggings sider. | Etablere hosting med HTTPS kryptering, og SSL. | Ferdigstill hele prosjektet. |  |
| 14:00-16:30 | Ferdigstille og planleggings-dokument og sende inn til prøvenemda. | Opprette database kommunikasjon, og bygg logikk for endepunktene. | Starte opprettelse av frontend prosjekt. |  | Implementere sikkerhets-tiltak. Eks. input validering |  |  |

## Oversikt over utstyr

Programvare:

* **Utviklingsverktøy**
  + **Visual Studio 2022** – Hovedverktøy for utvikling.
  + **Postman** – For testing av API-endepunkter.
  + **TortoiseGit** – For versjonskontroll og sikker lagring av kildekode i GitHub.
  + **SQL Server Management Studio (SSMS)** – For databaseadministrasjon.
  + **ReSharper** – Verktøy som analyserer koden i VS, og legger til formatting på kodestilen.
  + **Docker Desktop** – For lokal deployement av programvare og miljø.
* **Backend**
  + **ASP.NET Web Api** **.NET 8**– Rammeverket for Web APIet som håndterer databehandling for web appen.
  + **MediatR** – Bibliotek for å håndtere logikken til endepunktene. Etablerer en «mellommann» mellom komponentene i APIet, som forminsker direkte avhengigheter og gjør kodebasen lettere å vedlikeholde. (CQRS pattern)
  + **System.Data.SqlClient –** Bibliotek for å la APIet kommunisere med database.
  + **NUnit –** Testbibliotek, slik at API kan kjøre tester på funksjonalitet ved deployement.
* **Frontend**
  + **Blazor .NET 8 -** Rammeverket webappen vil bli utviklet i.
* **Database**
  + **Microsoft SQL Server –** relasjonsdatabase for lagring av brukere, lister og varer.
* **Sikkerhet** 
  + **HTTPS/SSL** – kryptert kommunikasjon mellom klienter og server.

Utstyr:

* Pc med nødvendig programvare, og tilstrekkelige ressurser til å kjøre prosjektet og programvarene. (DELL Latitude 7400)
* Mobiltelefon for testing. (SM-S24)
* Stabil internettforbindelse.
* Azure cloud for hosting.

## Kostnadsoverslag

### Infrastrukturkostnader

|  |  |  |
| --- | --- | --- |
| Tjeneste | Beskrivelse | Pris (NOK/mnd) |
| Azure MsSQL database | Database i Azure med 2vCPUs, og 10GB Minne. (Kan måtte skaleres opp i fremtiden) | 4.293kr |
| Azure App Service (P1v3) | Hosting for API, 2 vCPUs, 8GB Minne, 250GB lagring. | 1.749kr |
| Azure App Service (P1v3) | Hosting for API, 2 vCPUs, 8GB Minne, 250GB lagring. | 1.749kr |
| SSL-sertifikat | Gratis via Azure | 0kr |
| Sum infrastruktur |  | 7.791kr |

### Engangskostnader

|  |  |  |
| --- | --- | --- |
| Utstyr | Beskrivelse | Pris |
| Dell Latitude 7400 | Laptop brukt for utvikling | 7.000kr |
| Samsung S-24 | Mobiltelefon brukt for testing | 7.450kr |
| Sum Engangskostnader |  | 14.450kr |

### Arbeidskostnader

I tillegg til disse kostnadene kommer kostnadene av arbeidet. Har estimert å bruke ca. 45 timer på dette arbeidet. Med utgangspunkt i en timelønn på 500kr, tilsvarer dette 22 500kr.

### Totale kostnader

Totale kostnader på dette, beregnet fra infrastrukturkostnader (7 791kr), engangskostnader (14 450kr), og arbeidskostnader (22 500kr). Blir de totale kostnadene 44 741kr. Med løpende månedlige kostnader på 7 791kr (infrastruktur).

## Informasjonskilder og Samarbeidspartnere

**Informasjonskilder**

* **Stack Overflow** – For løsninger eventuelle problemer som kan oppstå.
* **GitHub** – For løsninger på eventuelle problemer.
* **Microsoft Docs** – Dokumentasjon for rammeverkene jeg skal benytte.
  + **SQL Docs**
  + **ASP.Net Docs**
  + **Blazor Docs**

**Samarbeidspartnere**

* **Kollegaer –** Ved eventuell nytte så kan kollegaer benyttes for informasjon, eller hjelp med problemer.

## Forenklinger, Avgrensninger og Forbehold

Grunnet naturen av oppgaven så har jeg valgt å kjøre opp miljøet mitt i Docker Desktop, for å slippe store kostnader i Azure portalen. I prinsippet vil prosjektet være identisk, jeg vil benytte de samme tjenestene, og om prosjektet engang skulle bli deployed i Azure så ville det ha krevd minimale endringer. Eneste forskjellen vil at ettersom dette kjøres lokalt, så vil ikke HTTPS kryptering inkluderes, men det ble vurdert og tenkt nøye på under planlegging, og hadde vært veldig relevant i en reell deployement. Dette ville ha blitt gjort via Azure sin gratis Let’s Encrypt tjeneste.

Jeg gjør også forbehold om at det kan oppstå endringer undervis i oppgaven, grunnet problemer eller endringer fra prøvenemda. Men grunnet hvor modulært planleggingen min er, og mtp. at alle «komponenter» i prosjektet har ansvar for sin egen logikk og sin egen del så skal dette gå fint. Målet med planen min er å ha ett mest mulig fleksibelt design, som kan ta imot endringer, og lett bli utvidet i fremtiden.

# Dokumentasjonsdel

## Backend

Backend-delen av løsningen min er ansvarlig for all logikk og databehandling i applikasjonen. Den håndterer brukere, autentisering og autorisering, administrasjon av lister og varer, kommunikasjon med databasen samt sikrer en sikker og pålitelig dataflyt mellom frontend og databasen. Backend består av en **MS-SQL database** som kjører i Docker og et **ASP.NET 8 Web Api** utviklet i C#. Oppsettet er bygget slik at det lett kan hostes i Azure, men for fagprøven bestemte jeg meg for å hoste dette lokalt i Docker.

### Teknologier og verktøy

I utviklingen av backend valgte jeg en kombinasjon av moderne teknologier og rammeverk som sammen gir en løsning som er sikker, robust og enkel å videreutvikle. Valgene er gjort med tanke på både stabilitet i drift og på at løsningen skal være ryddig å vedlikeholde over tid. Rammeverket er også platformuavhengig og kan kjøres på både Windows, Linux og i containere, noe som gir fleksibilitet i forhold til drift og fremtidig skalering.

ASP.NET Core – Web API

Jeg valgte **ASP.NET Core .NET 8,** som rammeverk for å bygge Web APIet. ASP.NET Core gir ett stabilt og velprøvd grunnlag for å bygge sikre og skalerbare webapplikasjoner.

En stor fordel med ASP.NET Core er støtten for **dependency injection (DI)** som er innebygget i rammeverket. Dependency Injection er ett prinsipp hvor man ikke selv oppretter objekter direkte, men i stedet får de levert inn som avhengigheter fra rammeverket. Dette gir flere fordeler:

* **Løs kobling mellom klasser:** Klassene blir mindre avhengige av hverandre og enklere å gjenbruke.
* **Bedre testbarhet**: Når en klasse får avhengighetene sine injisert, kan man enkelt bytte ut ekte implementasjoner med mock- eller test-implementasjoner.
* **Mer fleksibilitet**: Hvis man ønsker å bytte ut f.eks en databaseklient med en annen, kan dette gjøres uten å endre selve koden i resten av systemet.

I mitt prosjekt har jeg brukt DI blant annet til å injisere:

* **Autentifikasjon og autorisasjon** via egendefinerte authentication handlers, hvor mine egne tokens blir validert og håndtert. Dette gir stor fleksibilitet om autentifikasjonen skal utvides i fremtiden.
* **Databasekommunikasjon** slik at det kan brukes på tvers av controllere og services uten å være hardkodet.
* **MediatR-handlere**, som håndterer commands og queries, og som automatisk blir oppdaget og injisert av rammeverket.

For å strukturere applikasjonen har jeg brukt **MediatR,** et rammeverk som gjør det enkelt å implementere **CQRS-mønsteret (Command Query Responsibility Segregation).** Dette mønsteret deler applikasjonen opp i to typer operasjoner:

* **Commands** – operasjoner som endrer data. For eksempel å skrive noe til databasen.
* **Queries** – operasjoner som hente data. For eksempel å lese inn data fra databasen.

Ved å bruke MediatR blir controllerne mine veldig «tynne» og ryddige. Controllerne mottar kun forespørselen, sender den videre som en command eller query, og returnerer resultatet. Selve logikken ligger i handlerne. Dette gir flere fordeler:

* **Enkel testing:** Hver handler kan testes isolert uten å gå via hele API-laget.
* **Bedre organisering:** Koden blir modulær og enkel å navigere i. Jeg vet at all logikk for en bestemt operasjon ligger i sin egen handler.
* **Skalerbarhet:** Hvis applikasjonen vokser, kan jeg enkelt legge til flere handlers uten å gjøre controllerne komplisert.
* **Seperasjonsprinsipp**: Lesing og skriving av data er tydelig adskilt, som reduserer risikoen for feil og gjør det lettere å forstå flyten i systemet.

For logging i APIet har jeg brukt **Serilog**. Logging er en kritisk del av enhver backend, og Serilog har den fordelen at det støtter **strukturert logging**. Det vil si at jeg ikke bare lagrer tekstlinjer, men kan logge data som felter. Dette gjør det langt enklere å filtrere og analysere loggene i etterkant.

Serilog integreres tett med ASP.NET Core, og jeg kan konfigurere loggingen i oppstarten av applikasjonen. På den måten logges alle kritiske hendelser, feilmeldinger og brukerhandlinger på en strukturert måte.

I tillegg til å bruke Serilog har jeg laget en egen ExceptionExtensions klasse, hvor jeg kan håndtere Exceptions i try/catch bokser, og logge dem basert på Exception type.

For å gjøre applikasjonen mer robust mot midlertidige feil, som ustabil databaseforbindelse eller nettverksproblemer har jeg brukt **Polly**. Polly gjør det mulig å definere retry-strategier, for eksempel at et databasekall skal forsøkes på nytt opptil tre ganger med et lite intervall mellom forsøkene. Dette gir økt stabilitet, og bedre brukeropplevelse. For å bruke Polly effektivt har jeg laget en Utilities klasse for SQL logikk. Denne definerer Polly «Handlers» som håndterer database connections, og kall.

For dokumentasjon og testing av APIet har jeg brukt **Swagger / Swashbuckle**. Swagger genererer automatisk en interaktiv dokumentasjonsside der jeg kan se alle tilgjengelige endepunkter, hvilke parametere de tar imot, og hva de returnerer.

Fordelene er:

* **Selvdokumenterende API**, som gjør at andre utviklere (eller brukere) lett kan ta i bruk APIet uten å måtte lese kode.
* **Versjonering,** hvor jeg kan legge til nye versjoner av endepunkter (f.eks v1, v2) uten å bryte eksisterende funksjonalitet.

Jeg har lagt til ett par custom Swagger regler i mitt API for mitt bruk. Jeg har skrudd av Swagger sin innebygde «test» funksjon, dette er for å øke sikkerhet. Hvor eneste måten man kan kalle APIet er via selve APIet. Jeg har også definert versioning, slik at om ett endepunkt skulle endres i fremtiden så kan en «v2» kontroller enkelt opprettes uten å ødelegge eldre logikk.

### Helper classes og selv-lagde verktøy

For å enklere programmere, og senere utvide APIet har jeg laget en del «helpers» og verktøy. Dette er primært for å styrke fleksibilitet, viderutvikling, ryddighet samt å følge prinsipper om ansvar. Hvor enkelte klasser har ansvar for diverse operasjoner.

CredentialsUtils

**CredentialsUtils** er en hjelpeklasse jeg definerte for håndtering av passord, og sikkerhet i løsningen. Den er laget for å tilby en enkel, men tydelig strukturert måte å generere og validere passord-hash med salt. Klassen har følgende funksjonalitet:

* Genererer et unikt *salt* ved hjelp av «RandomNumberGenerator», som gjør at to passord aldri vil få en lik «hash».
* Kombinerer passord og salt, og hasher dette med en kryptografisk algoritme. Her valgte jeg MD5.
* Returnerer hashen som en streng i heksadesimal-format, slik at den kan lagres trygt i databasen.

Denne klassen blir brukt for å generere salts, og å hashe passordet til brukeren når brukeren registreres. Hashingen blir også brukt for å sammenligne passord, med passordet lagret i databasen ved innlogging.

Jeg bruker MD5 som krypteringsalgoritme. Fordelen med MD5 er at den er svært rask, enkel å bruke og tilgjengelig direkte i .NET uten eksterne avhengigheter. MD5 har ett par sikkerhetsmessige svakheter, men det mest kritiske er hvor rask den er. Dette gjør det enklere å brute-force. I en større produksjon ville jeg ha benyttet ett mer sikkert rammeverk for kryptering som f.eks bcrypt. Men grunnet begrenset tid, og størrelsen på oppgaven valgte jeg her MD5.

SqlDatabaseUtils

**SqlDatabaseUtils** er en hjelpeklasse som standariserer tilkobling og kjøring av SQL-spørringer. Den er spesielt fokusert på robusthet gjennom bruk av retry-mekanismer ved bruk av Polly.

Klassen har funksjon for å generere en «Connection» til databasen, her bruker jeg Polly for å ha retry-funksjonalitet ved åpning av connectionen. Det er også diverse funksjoner for å utføre kall, basert på behov. Primært så er forskjellen at man kan gjøre kall som returnerer en reader, eller såkalte «Non Query» kall som ikke returnerer noe data. Alle kallene får 3 forsøk, med en intervall som vokser eksponentielt i forhold til antall forsøk.

Fordelen med denne klassen er økt robusthet, og at APIet fremdeles vil fungere ved korte problemer ved nettverksforbindelsen, eller databasen.

Extensions Classes

Jeg har to hjelper-klasser for å opprettholde ryddigheten i Program.cs, disse er **ApplicationBuilderExtensions** og **ServiceCollectionExtensions**.

**ApplicationBuilderExtensions** sitt ansvar er primært å bygge Swaggeren og å håndtere feilmeldinger, her har jeg definert egne endepunkter basert på versioning (v1, v2…). Denne klassen er svært liten, men det er en metode som er greit å holde i en egen klasse for å opprettholde ryddighet og lesbarhet i koden. Klassen har også en metode som bygger på feilmeldinger, her har jeg en switch statement som tar imot 3 egendefinerte Exception klasser (BadRequestException, NotFoundException og UnprocessableEntityException), dette gjør det lettere for meg å returnere korrekte http statuskoder ved feil. Dette brukes i database logikken, og gjør det lettere å vise feilmeldinger i frontend.

**ServiceCollectionExtensions** er en statisk klasse som utvider IServiceCollection med metoder for å registrere tjenster (via dependency injection), konfigurere autentisering, API-versjonering og Swagger dokumentasjon.

Her har jeg en metode (AddServices), hvor jeg legger til tjenester via Dependency Injection. Akkurat nå er det minimalt med tjenester her, men i fremtiden kan det bli lagt til mange flere, og da bidrar denne klassen til å minimere «rot» i Program.cs.

Jeg har også en metode som definerer custom autentifisering i APIet, denne metoden bruker min public RSA key for å lese bearer tokens som brukere sender inn til APIet. Ved gyldig token vil brukeren bli autentifisert. Det sjekkes også at tokenen ikke er gått ut på dato. Tokens er gylding i èn time.

Resten av klassen er dedikert til ekstra Swagger funksjonalitet. Hovedsakelig versioning, og definering av path hvor .xml filen brukt til Swagger lagres. Dette er bare konfigurasjon.

**ExceptionExtensions** er en statisk klasse som utvider Exception typen. Denne har en metode (ProcessAndLogException) som kan kalles på alle exceptions, i f.eks en try/catch boks. Denne metoden leser exceptionen, callingFunction og sourceFilePath. Dette logges så som Warning eller Error alt etter grad av Exception. Dette gjør det mye enklere å utføre logging, og gjør eventuelle feil mye lettere å etterforske.

**HttpContextExtenions** er en statisk klasse som utvider HttpContext. Her har jeg logikk for å hente ut brukeren som kaller APIet sin bearer token. Dette gjør jeg for å kunne lese «claims» i tokenen, for å hente ut f.eks navn, expiration eller i dette tilfellet bruker ID (database verdi).

**TokenService** er en egen service-klasse i API-et som har ansvaret for å generere JWT-tokens (JSON Web Tokens). Denne klassen er en sentral del av autentiserings systemet i prosjektet, og bidrar til at både brukerinnlogging og tilgangskontroll blir både sikker og skalerbar.

Når en bruker logger inn, trenger backend å kunne identifisere og validere brukeren ved senere forespørsler. I stedet for å lagre passord eller brukerinformasjon i hver request, genererer TokenService en kryptografisk token som inneholder nødvendig informasjon (f.eks Bruker ID). Dette tokenet sendes tilbake til klienten og legges ved i headeren på fremtidige API-kall. På denne måten kan API-et gjenkjenne brukeren uten at de må logge inn på nytt hver gang.

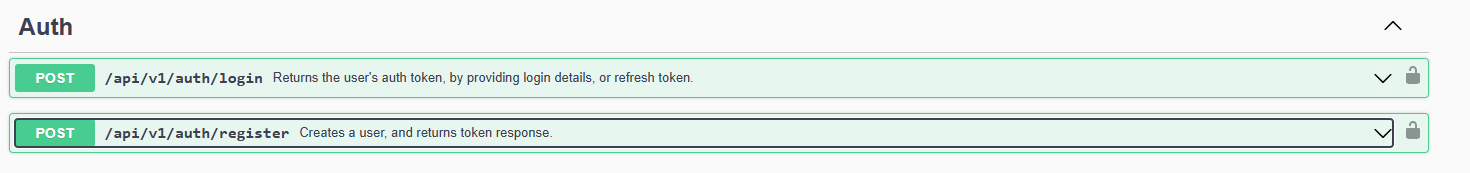
Klassen fungerer ved at den tar inn en kryptografisk nøkkel (RSA private key). Nøkkelen brukes til å signere tokenet slik at de ikke kan forfalskes. Først defineres en SecurityTokenDescriptor, som inneholder informasjon som BrukerId, brukerens navn, Issuer (hvem som lager tokenet), når tokenet ble produser og når det går «ut på dato». Denne informasjonen er veldig nyttig å ha på lagret slik at den kan hentes ut i frontend, eller ved fremtidige API kall uten å måtte generere eller hente dataen på nytt fra databasen. Deretter brukes RSA-algoritmen til å signere tokenet, så serialiseres tokenet og returneres.

### Endepunkter og Controllere

I Apiet er det definert to controllere, **AuthController** og **ListsController**. Disse har ansvar for hver sin del av funksjonaliteten i prosjektet. Og er derfor splittet opp for å styrke ansvars-prinsipper og forsterke ryddigheten og strukturen i prosjektet. Controllerne har egendefinerte paths, for å enklere splitte opp logikken. Dette gjør det også veldig enkelt å utvide APIet senere med nye endepunkter eller nye controllere

AuthController

AuthControlleren har ansvar for oppretting av brukere, og innlogging. Endepunktene som er definert i controlleren er:



Login endepunktet tar inn FormData:

A screenshot of a computer

AI-generated content may be incorrect.

Login har definert to AuthTypes, normalt sett så vil «Password» brukes. Her fungerer pålogging som normalt, ved at brukeren sender inn e-post og passord som matches opp mot databasen. Her hashes passordet, og matches opp mot passwordHash i databasen (ved bruk av salting). Så genereres ett token via TokenService og returneres til brukeren. Her returneres også en «RefreshToken» som lagres i databasen. Denne har en lengre utløpstid enn det normale tokenet som lages i TokenService, men inneholder ikke noen spesifikk informasjon.

Ved bruk av «RefreshToken» authtype, så sender brukeren ikke inn e-post eller passord. Men en «refershToken». Denne matches opp mot en RefreshToken tabell i databasen. Hvis ett tilsvarende token blir funnet, genereres en ny RefreshToken og AuthToken som returneres til brukeren. Tanken bak denne logikken er at frontenden kan lagre RefreshToken ved pålogging, og gjenbruke denne for re-autentifisering når det normale tokenet går ut på dato. Da slipper brukeren måtte logge seg på igjen hver time, og vil da heller forbli pålogget så lenge brukeren benytter seg av appen innenfor RefreshTokenet sin levetid. Dette styrker sterkt brukeropplevelsen.

Register endepunktet tar in FormData:

A screenshot of a computer

AI-generated content may be incorrect.

Registrerings endepunktet fungerer enkelt og greit ved at dataen som brukeren sender inn blir sendt til databasen etter litt enkel databehandling. En «salt» blir generert for passordet. Dette er en tilfeldig streng som genereres og legges på passordet for å passe på at to potensielt like passord ikke genererer en lik hash. Etterpå blir så passordet hashet, før det lagres i databasen. Ved posting til databasen sjekkes det også om noen brukere med lik e-post allerede er registrert, om dette er tilfellet så vil registreringen faile. Ved en fullført registrering returneres så en TokenResponse slik som i login endepunktet, slik at man slipper å måtte registrere seg for å så logge inn.

ListsController

ListsControlleren har ansvar for å hente ut, og opprette handleliste-data. Dette gjøres via disse endepunktene definert i Swagger dokumentasjonen:

A colorful lines on a white background

AI-generated content may be incorrect.

**/Lists (POST)** endepunktet oppretter en ny handleliste sammen med varer ved at brukeren sender inn en JSON body med følgende data.

A black rectangular object with a white background

AI-generated content may be incorrect.

Denne dataen blir opprettet i databasen og linkes til brukerens brukerId, for å så senere kunne bli hentet ut.

**/Lists (GET)** endepunktet henter ut handleliste data for brukeren, dette gjøres vet at brukerens Id hentes ut fra Bearer tokenen som blir brukt i kallet. Denne Iden brukes for å hente ut alle lister knyttet opp til brukeren. Disse returneres med navn og id som brukes i frontenden.

**/Lists (PUT)** endepunktet endrer på data i lister og varer. Dette endepunktet er designet for å endre navn på listen, legge til / endre varer eller fjerne varer fra listen. Dette gjøres ved at det sendes inn en modell med navn og varer som ønskes å ha i listen. Man må altså da legge til alle varer som skal **forbli** eller legges til i listen. Alt annet blir slettet. Modellen som brukes er:

A computer screen shot of code

AI-generated content may be incorrect.

**/Lists/{id}** endepunktet henter ut all lagret data om en liste, og varene i listen ved bruk av listens Id. Dette brukes f.eks i frontenden når man trykker inn på en spesifikk liste. Dataen som returneres er definert i swaggeren:

A computer screen shot of a code

AI-generated content may be incorrect.

**/Lists/{id} (DELETE)** endepunkter er ansvarlig for å arkivere lister. Tanken her er at når en bruker markerer alle varene som «handlet» eller i kurven, så får brukeren en prompt om å slette listen ved ønske. Dette gjøres ved at listen blir satt som «completed» i databasen, og vises da ikke i andre stored procedures. Jeg valgte å ikke slette listen permanent, for å kunne ha potensiale for å implementere en arkivliste i fremtiden.

**/Lists/items** endepunktet er ansvarlig for å oppdatere status for handlevarer. Dette endepunktet oppererer med ett «batch» prinsipp, hvor tanken er at frontenden grupperer sammen flere varer som skal oppdateres samtidig, for å så sende dem inn i en liste med id og status. Statusen som blir oppdatert i dette tilfelle representerer om varen er «lagt i kurven» / handlet. Slik at brukeren lett kan ha oversikt på hva som resterer på handleturen. Dette endepunktet returnerer bare en status, ettersom at dataen allerede ligger i frontend når dette endepunktet brukes.

**/Lists-templates** endepunktet henter ut en liste med maler som kan brukes av brukeren for å lett opprette handlelister raskt. Eksempelvis så kan en mal være «Tacokveld», hvor listen inneholder lefser, salsa osv. Da kan brukeren velge vekk enkelte varer, endre antall eller legge til nye varer. Dette forenkler prosessen for brukeren ved repeterende handleturer.

### Database

I dette prosjektet benyttes en **MSSQL-Dataase.** Akkurat i dette tilfellet kjører databasen i en docker container, dette er grunnet naturen av oppgaven. Men i en reell produksjon så kunne denne databasen lett settes opp i Azure. Jeg bruker Microsoft SQL Server Management Studio til å operere databasen, hvor jeg har installert RedGate verktøyet (bedrift standard), dette tillater meg å lett benytte git som versjonskontroll system i databasen. Alt av tabeller, brukere, roller og stored procedures er derfor lagret i git. Og det å flytte database strukturen vil derfor være veldig enkelt.

Databasen har opprettet ett eget schema «ShopApp», med en dedikert bruker «shop\_app\_user» og to roller «AuthManager» og «DataManager». Akkurat nå så er den eneste brukeren «shop\_app\_user» og den har tilgang til begge rollene, men tanken her er at i en fremtidig oppskalert versjon av prosjektet så kan forskjellige brukere tilegnes disse rollene. Slik at bruker A bare har tilgang til operasjoner beskyttet av «AuthManager» og bruker B bare har tilgang til «DataManager». Dette er rett og slett bare lagt inn for fremtidssikring og oppskaleringsmuligheter.

Databasen er designet for å støtte kjernefunksjonaliteten i handleliste-appen. Den består av flere tabeller som lagrer relevant data.

A screenshot of a computer

AI-generated content may be incorrect.

User tabellen er ansvarlig for å holde brukerinformasjon, og brukes ved registrering og pålogging. Her lagres også passordsalten som brukes ved hashing.

RefreshToken tabellen er ansvarlig for å lagre RefreshTokens som genereres ved autentifisering, for å bruke ved fornying av token senere.

Lists databasen inneholder data om handlelistene, som hvem listen tilhører, og navn på listen. Denne knyttes tett opp mot Items databasen, som inneholder handlevarer. Disse knyttes opp med en foreign key via list\_id.

ListsTemplates og ItemsTemplate er mer minimale versjoner av Lists og Items, som bare inneholder manuelt innfylt data som brukes i visningen av «maler» i frontenden.

All tilgang til databasen skjer gjennom Stored Procedures, noe som bidrar til sikkerhet og ryddighet. Eks. Ved registrering eller innlogging blir data sendt til en prosedyre som enten oppretter en ny bruker eller henter frem en eksisterende. Deretter genereres nye tokens (refresh), og disse returnerest til API-et. Jeg har valgt å ha all database logikk i Stored Procedures for å igjen splitte opp ansvarsområder. Databasen er ansvarlig for dataen, og innhenting og skriving av data skal da skje av databasen. Dette bidrar også til at om database logikk behøver endring så kan dette skje uten at ett API må endres på og redeployes. Derfor er det null SQL queries i APIet. Dette gjør det også lettere å dele datase logikk inn i roller (DataManager, AuthManager) for fremtidig bruk av disse rollene.

Følgene Stored Procedures finnes i databasen:

A screenshot of a computer program

AI-generated content may be incorrect.

Her finnes det prosedyrer for å opprette handlelister, hente dem ut etter brukerId, hente ut varer fra en handleliste via listeId, hente ut template lister, sette status (lagt i kurv) på varer, opprette brukere, logge på, og fornye refresh tokens. Disse blir alle brukt i APIet for å danne kjernefunksjonaliteten i appen.

Som nevnet tidligere brukes RedGate for git funksjonalitet i databasen, dette vises på skjermbilden (se blå sirkler), hvor ny/endret data får ett ikon. Jeg har da muligheten til å høyreklikke på databasen for å commite endringer til ett git repo. Man har også mulighet til å opprette database fra ett repo, eller gjenopprette ved rollback. Dette er veldig nyttig for database utvikling og har vært veldig greit under dette prosjektet for å holde styr på endringer.

## Frontend

Til frontend valgte jeg å bruke **Blazor ASP.NET Core .NET 8**. Blazor er ett morderne rammeverk fra Microsoft som gjør det mulig å bygge interaktive webapplikasjoner med C# i stedet for det mer typsike valget JavaScript (evt. TypeScript). Dette gir en mer helhetlig utviklingsopplevelse ettersom både frontend og backend deler samme språk og kan gjenbruke mye av den samme logikken og modellene.

Dette var ett valg jeg først var skeptisk til, ettersom at det for meg er ett helt nytt rammeverk. Min frontend web-app erfaring er hovedsakelig innenfor React, React-native og Angular, primært i JS/TS. Men for å matche backend valgte jeg å forsøke meg på dette rammeverket. Som har gitt mye lærdom, og det er ett rammeverk jeg har sett resultater i. Hovedgrunnen bak dette valget var å prøve noe nytt, ettersom at lærdom er den viktigste egenskapen i dette yrket, og fordelen ved dette valget er da helheten med frontend og backend. Dette førte til ett par vanskeligheter og problemer. Mye tid ble benyttet på å implementere autentifisering i frontenden, hvor jeg støtte på mange problemer og mange bugs. Dette blir beskrevet mer i detaljer senere.

### Oversikt

Frontend-applikasjonen består av flere **Razor Pages** og **komponenter** som hver har tydelige ansvarsområder og funksjonalitet. For å skape en ryddig struktur og unngå duplisering av kode, er hovedsiden wrappet inn i et eget layout kalt **MainLayout**. Her plasseres felles elementer som skal gjenbrukes på tvers av sidene – for eksempel **navigasjonsbar**, **autentiseringslogikk**, og en **error-boks** for feilmeldinger. Dette gjør at nye sider kan lages raskt, uten at man må gjenta disse elementene manuelt.

I tillegg finnes et **EmptyLayout**, som er et minimalistisk layout uten navigasjon eller autentiseringskrav. Dette layoutet benyttes på **Login**- og **Register**-sidene, hvor det er ønskelig at brukeren ikke distraheres av menyelementer eller funksjoner som kun er tilgjengelig etter innlogging. På den måten holdes innloggings- og registreringsprosessen enkel og fokusert.

Når brukeren logger inn, lagres tokenet lokalt i frontend via localstorage, og applikasjonen bytter automatisk til å bruke MainLayout. Her møtes brukeren først av **ListCreate**-siden hvor det er mulig å opprette nye handlelister. Til venstre ligger en **ListBar**-komponent (integrert i MainLayout) som dynamisk henter inn alle eksisterende handlelister fra API-et. Hver handleliste representeres med en knapp i denne sidebaren, og ved å trykke på en knapp navigeres brukeren til **ListDetails**-siden. Her får man oversikt over innholdet i en spesifikk handleliste, samt mulighet til å legge til eller fjerne varer.

For autentisering har frontend egne sider for **Login**, **Register** og **Logout**. Login og Register samhandler direkte med API-et for å sende innloggingsdata eller opprette nye brukere, og mottar et **JWT-token** samt et **refresh-token** som deretter brukes i videre kommunikasjon med API-et. Logout-siden sørger for å fjerne tokens fra lagring i frontend, og brukeren blir automatisk sendt tilbake til en side med EmptyLayout.

Dette designet gir en tydelig oppdeling mellom innloggede og ikke-innloggede brukere, samtidig som det bidrar til en **ren og brukervennlig opplevelse**. Brukeren møter kun de komponentene som er relevante for gjeldende tilstand, og navigasjonen er bygget opp slik at det alltid er enkelt å bytte mellom ulike handlelister eller opprette nye.

A screenshot of a computer screen

AI-generated content may be incorrect.

### Detaljert oppbygning

Web-Appen består av mange forskjellige komponenter, layouts, sider og state managers. Dette er en mer detaljert oversikt og forklaring for hver enkelt komponent.

**Layouts**

* **MainLayout**

MainLayout brukes som standard for sider som krever at brukeren er innlogget. Her ligger komponenter som navigasjonsbar (ListBar), error-boks, og eventuelle statusmeldinger. Layouten fungerer som en ramme rundt sidene og sikrer at alle brukeropplevelser for innloggede brukere er konsistente. MainLayout er også ansvarlig for å verifisere om brukeren er autentifisert, og eventuelt navigere til Login siden om det trengs. Dette gjørs ved å sjekke AuthState (forklares senere), og expiration på token som lagres i LocalStorage.

* **EmptyLayout**

EmptyLayout brukes for sider der det ikke er behov for navigasjon eller autentiseringskrav, i dette tilfelle Login, Logout og Register. Denne layouten er enkel og minimalistisk, og bidrar til at brukeren ikke får tilgang til funksjoner før vedkommede er autentisert.

**Komponenter og Sider**

* **Login**

Her møter brukeren ett «form», hvor e-post og passord må fylles inn. Man har også muligheten til å navigere seg til Register-skjermen om man ikke har en bruker. Logikken her fungerer ved at en AuthService kalles, hvor token hentes ut. Denne blir så verifisert med standard null og empty checks, hvor den så blir servert til AuthStateProvideren. Om alt er vellykket så navigeres brukeren til MainLayout.

* **Logout**

Denne siden inneholder minimalt med logikk, og har ingen view. Den sender en forespørsel til AuthService om å bli logget ut som sletter tokens fra localStorage, og sender en oppdatering til AuthStateProvideren. Brukeren blir så navigert til Login siden.

* **Register**

Register siden er så å si identisk til Login, med unntak av at man også må fylle inn navn her. Det bes også om å skrive inn passordet to ganger for å bekrefte at man har skrevet inn korrekt passord. Logikken her er også ganske lik. Først blir passordfeltene sjekket opp mot hverandre, og om de er ulike vil dette vises i viewet. Etter dette blir en forespørsel sendt til AuthService, token blir validert, AuthStateProvider blir varslet og brukeren blir navigert til MainLayout.

* **ListCreate**

ListCreate er hovedsiden, og den første siden som brukeren møter etter innlogging. Her har man litt forskjellig funksjonalitet, men alt bidrar til å opprette en ny handleliste. Man blir møtt med en dropdown, hvor man har mulighet til å velge fra diverse maler. Ved valgt mal vil varer automatisk legges til i listen, og navn på listen vil automatisk settes. Dette kan så bli redigert etter brukerens behov. Dette gjøres via inputfelt for navn og inputfelt for varer. Inputfeltet for varer har også funksjonalitet for å sette **antall** varer. Varer som blir lagt til i listen kan også fjernes ved behov før brukeren til slutt kan velge å opprette listen. Ved opprettet liste blir man automatisk navigert til listens detaljeside. Mesteparten av logikken her går via en ApiClient som kommuniserer med backend.

* **ListBar**

ListBar er en komponent i MainLayout som alltid vises på venstre side. Ved initialisering av komponenten hentes det ut lister fra backend. For hver liste som blir hentet ut lages det en knapp på baren hvor det vises ett navn, ved å trykke på knappen navigeres man til listens detaljeside ved bruk av listeId. Eks. på url: /lists/12. Man har også en knapp her for å navigere tilbake til ListCreate siden om det trengs.

* **ListDetails**

Viser innholdet i en valgt handleliste. Listen blir lastet inn ved bruk av listeId som fås som parameter fra URLen. Denne blir brukt i ett kall til APIet via APIClienten. Her vises alle varene i handlelisten, hvor brukeren har mulighet til å sjekke av på om varer er lagt i kurven eller kjøpt. Dette bidrar til å gjøre handleturen mer systematisk og lett for brukeren. Varer vises via **ShoppingItem** komponenten.

* **ShoppingItem**

ShoppingItem komponeten er bare ett enkelt «kort», som inneholder detaljer om varen. Her står det navn på vare, når den ble lagt til, antall varer og status på om den er avsjekket. Komponenten inneholder en event metode for når man krysser av sjekkboksen. Denne metoden blir brukt i ListDetails skjermen for å oppdatere statusen i APIet.

* **ErrorMessageBox**

ErrorMessageBox er en komponent jeg utviklet for å kunne vise feilmeldinger globalt i webappen. Denne er i grunnen en kopi av Blazor sin innebygde errorbox, men utvidet til å bruke en global ErrorNotifier state klasse. Dette lar meg trigge feilmeldinger i andre tjenester og komponenter, som kan vises uansett hvor brukeren er i webappen.

**State management, kommunikasjon og Servicer.**

* **CustomAuthStateProvider**

Denne klassen er en utvidelse av Blazor sin innebygde **AuthenticationStateProvider**, som er hovedmåten Blazor definerer autentifikasjon på. Klassen er ansvaring for å logge innloggingstilstanden til brukeren basert på et lagert JWT-token. Tokenet lagres i nettleseren via ProtectedLocalStorage. Denne kan sjekkes ut via å inspisere nettsiden (F12) -> Application -> Local Storage. Ved oppstart via GetAuthenticationStateAsync metoden hentes det ut en ClaimsIdentity fra tokenet, som representerer brukeren. Denne sjekkes for å se om brukeren er autentisert. Ved pålogging og avlogging kalles MarkUserAsAuthenticated og MarkUserAsLoggedOut, for å endre staten i nettleseren. Denne klassen bidrar til en dynamisk reaksjon på påloggings-status.

* **AuthService**

AuthService er en tjeneste som håndterer autentisering i frontend-applikasjonen. Den kommuniserer med backend via ShopAppApiClient for å logge inn, registrere og fornye refresh-tokens. Når en bruker logger inn eller registrerer seg mottas tokenet fra backend. TryRefreshTokenAsync sjekker om refresh-token finnes i localstorage, og henter nye tokens ved nødvendighet. Denne blir brukt når brukeren har tokens lagret i storage, og åpner nettsiden. For å forhindre å måtte logge på hver gang siden åpnes.

* **ShopAppApiClient**

ShopAppApiClient er en klientklasse som kommuniserer med backend APIet. Den håndterer autentisering, innhenting og skriving av data. Clienten tar inn en HTTPClient, hvor jeg har lagt til en **ApiExceptionHandler.** Denne klassen brukes for API kall i hele frontend applikasjonen.

* **ApiExceptionHandler**

Dette er en klasse som er til for å fange http exceptions, dette gjør at nettsiden ikke kræsjer om APIet returnerer f.eks 404, 401 osv. Optimalt sett så burde denne klassen utvides i fremtiden til å håndtere forskejllige feilmeldinger på forskjellige måter, med logging, visning og eventuelt retry mekanismer. Som er implementert til en viss grad nå, men jeg fikk ikke tid til å utvide dette etter ønske. Nå håndterers 400, og 404 feilmeldinger her, og vises i ein feilmeldingsboks. Andre feil logges, og vises som feil uten ekstra logikk.

* **ErrorNotifier**

ErrorNotifier er en simpel klasse med ett parameter definert som Action, denne Actionen tar inn en streng som inneholder en feilmelding. ErrorMessageBox komponenten bruker denne Actionen til å kjøre logikk når ErrorNotifier oppdateres. Oppdateringer på denne Actionen skjer ved at en hvilken som helst klasse i prosjektet kan instansiere klassen og kalle NotifyError(string message), da vil den tidligere nevnt Actionen trigges overalt hvor denne er brukt.

* **ListState**

ListState er en state klasse som brukes for å oppdatere listedataen, denne brukes i sidebaren hvor handlelistene vises. Denne klassen inneholder handlelistene og likt som ErrorNotifier lar komponetner abonnere på en Action. Denne blir brukt når f.eks brukeren lager eller redigerer en liste, da kalles ListState.RefreshAsync som oppdaterer handlelistene i hele prosjektet.

### Problemer som oppstod under Frontend

Som nevnet innledningsvis så hadde jeg ett par problemer med valget av frontend rammeverk. Dette var primært grunnet manglende kunnskap, men mesteparten av dette ble løst. For ordensskyld så vil følgende problemer bli beskrevet her. Disse har jeg tatt lærdom av, og problemene ble løst.

* **Autentisering**

Jeg hadde store problemer med å implementere autentifisering, jeg var ikke klar over Blazor sitt oppsett med AuthenticationStateProvider, AuthenticateView osv. Her stevet jeg lenge med at jeg bare lagret token i localstorage og hentet denne ut. Denne løsningen førte også til andre problemer, men etter hvert ved å lese igjennom Blazor dokumentasjon og mange StackOverflow posts så fant jeg løsningen via en custom Provider. Dette brukte jeg store deler av tiden til frontend utviklingen på.

* **Dependency Injection**

Jeg hadde problemer med DI, hvor det ble en del nested servicer som førte til kræsjing og at ingenting fungerte. Dette var tett opp knyttet til problemene med autentisering. For å løse autentiserings problemene, endte jeg opp med mange servicer (APIClient, AuthService, TokenStorage) som importerte hverandre. Til slutt slettet jeg disse servicene og startet over. Og har nå ingen direkte dependencies mellom disse. APIClient får tokens fra ProtectedLocalStorage, som er innebygget i Blazor. Disse blir definert i HTTPClienten manuelt, i stedet for at en annen klasse har ansvar for dette.

* **Kode som kjøres før render engine**

Jeg hadde også en del problemer hvor jeg hadde kode som påvirket ting i razor komponentene. F.eks så var første forsøket mitt på å logge brukeren ut ved «ugyldig» token å ha denne logikken i ApiExceptionHandler. Hvor jeg ved en Unauthorized status logget brukeren ut og navigerte dem til login skjermen. Dette fungerte ikke i Blazor, ettersom at denne koden kjørte før komponentene var rendret. Jeg lærte raskt at navigerings logikk og annen lik logikk må skje i komponentene selv, helst i en OnAfterRenderAsync metode. Denne logikken ble da altså flyttet til MainLayout.

* **Rendermode**

Jeg hadde flere problemer hvor jeg hadde interaktive komponenter (tekstfelt, knapper osv.) som selv om de hadde definert logikk ikke gjorde noe. Dette viste seg etter lesing i dokumentasjon å være grunnet «render mode». Sider i Blazor som har interaktive komponenter må definere @rendermode InteractiveServer øverst i dokumentet.

## Standarer

I løpet av utviklingen har jeg fulgt diverse standarer som jeg har lært i min tid i bedriften. Som tidligere nevnt i dokumentasjonen har vi en del verktøy som RedGate og ReSharper som hjelper til med kodeformatering, og diverse «regler». Eksempler på dette er unødvendig casting av variabler, at en variabel er definert med «var» i stedet for med typen sin (eks. string), unødvendig tegnsetting eller definering. ( Eks. MockModel testVar = new MockModel() | MockModel testVar = new() ). Disse standarne har jeg fulgt i dette prosjektet også.

Jeg har også hatt stort fokus på prinsipper som DRY (Do not repeat), hvor fokuset er at man skal lage tjenester eller metoder for repeterende kode. CQRS, hvor man splitter operasjoner opp i commands og queries (lese og skrive operasjoner), pattern og responsibility som definerer ansvarsområder for logikk.

Dette er standarer jeg som utvikler benytter meg av i arbeid for bedriften, som jeg mener styrker koden, lesbarheten og vedlikehold.

## Hosting

Den første tanken var at løsningen skulle hostes i skyen via Azure, men dette var ikke praktisk utførbart grunnet kostnader, og rettigheter. Jeg gikk derfor heller for hosting i Docker, hvor jeg «latet som» at det ble hostet i Azure. Prosjektet er altså da utviklet slik at det kan fungere i ett Azure miljø også. Ved en Azure deployement så hadde det vært relevant å definere repositories for API, Frontend og Database i Azure DevOps, her kan man definere pipelines per repo hvor man kan automatisk bygge API og Frontend til App Servicer i Azure ved push til master.

Det er veldig praktisk å lagre database strukturen i ett repo også, ved nøvendigheter for å bygge ny database eller rulle tilbake endringer. Dette kan også være praktisk om man skal sette opp ett Dev miljø med identisk database.

Ved hosting så har man også muligheten til å kjøre ApplicationInsights på f.eks APIet, hvor man kan logge feilmeldinger og lignende. Man har også da en enkel oversikt over loggene generert av Serilog.

## Testplan

Hensikten med testplanen er å verifisere at Frontend appen fungerer etter kravene, og at brukeren kan utføre nødvendige handlinger.

* Registrering/Pålogging
* En eller flere lister med varer
* En lett måte for brukeren å ha kontroll på hva som er handlet, og hva som gjenstår.
* En «template» funksjon som kan kopieres fra eller bruke som grunnlag for ny liste.

Testingen skal dekke

* **Autentisering (login, logout, registrering, refreshToken)** – og at autentiserte endepunkter bare fungerer ved innsendt og gyldig token.
* **Brukerinteraksjoner (opprette, hente lister og sette status på varer)**
* **Dataflyt** – mellom frontend, API og database. Henter korrekt data uten feil osv.
* **Feilhåndtering** – feil passord, utløpt token

Testingen blir utført ved bruk av

* **Manuell testing** – End-to-end testing av hele web-appens funksjonalitet.
  + Opprette bruker
  + Logout
  + Login
  + Opprette ny liste manuelt
  + Vise liste
  + Sette status på varer
  + Opprette ny liste fra mal
  + Vise liste, og bytte mellom lister
  + Logout og logge inn på annen bruker, verifisere at ikke «feil» lister vises.
* **Enhetstester** – APIet vil kjøre NUnit tester på enkelte områder, for å verifisere at kjernefunksjonaltiet fungerer. Dette vil være mest relevant for vedlikehold og eventuell videreutvikling.

Ut ifra dette har jeg definert meg ett par test caser jeg skal gå igjennom.

* Registrere ny bruker, forventes at ny token returneres og man blir navigert til hovedsiden
* Login med gyldig bruker, forventes at ny token returneres og man blir navigert til hovedsiden.
* Login med ugyldig passord. Forventes feilmelding.
* Registrering med allerede eksisterende e-post. Forventes feilmelding.
* Opprette handleliste. Forventes å navigere til ny liste.
* Hente handlelister. Forventes at handlelister hentes inn for innlogget bruker og vises i listebaren.
* Krysse av vare. Forventes at status oppdateres i Viewet, og at status vedvarer ved refresh av siden.
* Token utløper. Forventes at bruker navigeres til Login side, hvor forsøk ved refreshtoken vil utføres. Om dette mislykkes må brukeren logge inn på nytt manuelt.
* APIet kjører ikke. Forventes feilmelding.

### Testlogg

Jeg har manuelt testet disse test-casene flere ganger, og utifra kriteriene så er testene bestått. Testene som kjører på APIet kjører også fint. Manuell testing av nettside flowen fungerer som forventet. Gjør obs. på at det sannsynligvis eksisterer bugs, som kan finnes om man gjør feil ting. Men ut ifra mine tester så viser det ingen funksjons-ødeleggende bugs. Unit testene passerte også.

A screenshot of a computer program

AI-generated content may be incorrect.

## Brukermanual

I prosjektet mitt så har jeg utviklet en web-applikasjon som gjør det enkelt å administrere handlelister digitalt. Når du åpner applikasjonen i nettleseren, møtes du først av en innloggingsside. Dersom du allerede har en konto, skriver du inn e-post og passord, og trykker på *Logg inn*. Dersom du ikke har en konto fra før, kan du opprette en ved å velge *Registrer*. Da fyller du inn fullt navn, e-postadresse og passord, og etter at registreringen er gjennomført blir du automatisk logget inn i systemet.

A screenshot of a login screen

AI-generated content may be incorrect.A screenshot of a login form

AI-generated content may be incorrect.

Når du er logget inn kommer du til hovedsiden, hvor alle handlelistene dine er samlet. Til venstre ser du en listevisning med knappene til de ulike handlelistene du har laget. For å opprette en ny liste trykker du på *Ny Liste* (denne siden vises som standard når man logger inn), skriver inn navn, legger til varer og lagrer. Den nye handlelisten vil da dukke opp i oversikten din. Her har man også mulighet til å lage en handleliste ut ifra en mal, som velges ifra dropdown menyen øverst på siden. Dette vil da sette navn på listen og legge til varer fra malen, disse kan fjernes om ønsket og nye varer kan legges til frem til du er fornøyd.

A green and white screen

AI-generated content may be incorrect.

Man navigeres automatisk til den nye listen ved opprettet liste, her vises detaljene over handlelisten. Og handlevarer vil være listet nedover siden. Her vil det stå navn på vare, antall av varen og når den ble lagt til. Underveis i løpet av handleturen så kan det være greit å «sjekke av» varer som man har lagt i kurven. Dette gjøres enkelt ved å trykke på sjekkboksen til venstre for varenavnet. Da vil nettsiden huske at denne varen er sjekket av når du åpner siden igjen senere. Her har man også muligheten til å redigere lister om det ønskes, ved å slette varer, legge til nye varer eller endre navn på listen. Når alle varene er «sjekket ut», får man også ett varsel om man vil arkivere listen.

A screenshot of a computer

AI-generated content may be incorrect.

Ønsker man å navigere til en annen handleliste så kan dette gjøres enkelt ved å velge handlelisten ut ifra navn i sidebaren til venstre. Da vil man navigeres til detaljene for denne listen.

Ønsker man å logge ut av nettsiden så kan dette gjøres øverst til venstre hvor det står *Logg ut*. Man blir da navigert tilbake til logg inn siden, og alt av cachet bruker data vil tømmes fra lokal lagring.

## Kilder

Underveis i prosjektet har jeg brukt diverse kilder for hjelp med problemer som har oppstått, løsningsforslag til diverse logikk og for å lære om komponenter og rammeverk. Disse kildene har bestått av:

* StackOverflow
* Microsoft Docs (ASP.NET, Blazor, MSSQL)
* ReSharper sine innebygde hjelper funksjoner for formattering, warnings osv.
* RedGate sine innebygde hjelpere.
* GitHub forum

Jeg har også lastet min egen kildekode, skisser og dokumentasjon opp til Github på følgende repo. (<https://github.com/RasmusLNetland/Fagprove-Rasmus/tree/main>)

## Egenvurdering

### Oppsummering

Databasen ble satt opp som **MSSQL** database for å etterligne en reell deployement hvor alt hadde blitt hostet i Azure. Dette er også en database jeg er godt kjent med via jobb. Her startet jeg med å definere en bruker for APIet, roller og ett schema. Dette gjorde at alt grunnleggende var på plass. Deretter opprettet jeg alle tabellene jeg visste jeg trengte. Stored Procedures ble utviklet underveis som de ble trengt av APIet. Disse kunne jeg nok kanskje laget samtidig før jeg startet på APIet om jeg hadde kartlagt behovet fra databasen bedre.

Backenden ble utviklet i **ASP.NET Core 8**, hovedsakelig begrunnet god kjennskap med rammeverket, men også grunnet stabiliteten og effektiviteten. Det finnes også god dokumentasjon på rammeverket. Fremgangsmåten her var å først sette opp en del extensions klasser for dependency injection, exceptions og oppsett av prosjektet. Utilities for SQL og credentials ble også satt opp tidlig. Etter dette så var resten av prosessen ganske repeterende, ved at jeg opprettet en MediatR handler, integrerte database logikk i database servicen, og koblet det opp mot ett endepunkt i controlleren. Dette mener jeg var en hensiktsmessig fremgangsmåte som gjør det lett å utvide APIet fremover.

Jeg valgte å bygge frontend-løsningen i **Blazor .NET 8** fordi rammeverket gir god støtte for moderne webapplikasjoner, og det integrerer godt med backend rammeverket. Fremgangsmåten var å starte med kjernefunksjonaliteten, autentifisering og innlogging/registrering – før jeg bygde videre med token-håndtering, lagring i nettleser og integrasjon mot API klienten. For å så få på plass komponenter, sider og view. Jeg mener denne fremgangsmåten var hensiktsmessig fordi jeg sikret at det grunnleggende fungerte tidlig, og kunne utvide løsningen stegvis. Dette gjorde også at jeg fikk løst problemer jeg hadde med autentisering og Dependency Injection (nevnt i dokumentasjon) tidlig.

### Utført arbeid og utfordringer

Jeg har utviklet en frontend med autentiseringsflyt basert på JWT-tokens, som henter data og autentiserings token fra ett backend API. Dataen lagres i en database. Jeg er fornøyd med hvordan løsningen ble og mener den dekker oppgavens krav samt endringen som kom fra prøvenemda.

Underveis i prosessen oppstod det en del problemer, for det meste i Blazor rammeverket. Jeg slet lenge med å få til en skikkelig autentiseringsflyt, hvor token ble lagret korrekt og refreshtoken fungerte. Navigering til login ved manglende autentisering og lignende. Dette ble løst etter hvert, men ble brukt mye tid på dette.

Samt var det også probelmer med Dependency Injection, som hovdesakelig kom fra nested dependencies i sammenheng med autentiseringsproblemet. Etter hvert så fant jeg en løsning hvor jeg slap å neste avhengigher, som førte til at frontenden kræsjet.

Et eksempel på dette som jeg slet lenge med var integreringen av ApiClienten, hvor jeg trengte autentiserings token for de fleste endepunktene. Dette forsøkte jeg å løse ved å injisere en AuthService. Eneste problemet var at denne Authservicen var avhengig av ApiClienten for å hente ut token. Dette skapte en kræsj i flyten. Løsningen min her ble å la ApiClient hente tokenet selv fra localstorage, mens AuthService lagrer det.

### Avvik i løsning, i forhold til plan eller oppdrag

Løsningen min ble utviklet uten avvik i forhold til oppdraget, men jeg hadde større ambisjoner ved oppstart enn det jeg klarte å fullføre. Jeg hadde tanker om ett mer utvidet og grunding testmiljø i form av enhets-tester (NUnit), men dette rakk jeg ikke. Jeg fikk laget enkelte sjekker som passer på at endepunktene returnerer Ok status, men jeg hadde egentlig ønsket å dekke flere statuser enn dette (BadRequest osv.).

Det var også ønskelig å implementere ett healthcheck endepunkt, som kunne benyttes av frontend ved api exceptions for å verifisere om APIet kjører, men dette rakk jeg heller ikke. Så i situasjoner hvor for eksempel APIet er nede, eller databasen er nede vil brukeren få varsel om at noe er galt. Men så mye mer informasjon enn dette har ikke systemet å levere.

Jeg er heller ikke 100% fornøyd med design eller UI, og mener at dette hadde trengt mer arbeid om det hadde vært tid til det, men brukergrensesnittet er funksjonelt og oversiktlig nok til at det er brukbart.

Ellers føler jeg at løsningen ble utført som planlagt og til oppgavens krav, og jeg er fornøyd med løsningen utifra tiden jeg hadde på meg.

### Hva kunne jeg gjort bedre

Jeg tror de fleste problemene mine oppstod grunnet at planen ikke var nøye nok, jeg kunne brukt litt ekstra tid på å f.eks gjøre research om Blazor, i form av å f.eks se ett par kurs på pluralsight eller youtube om basic oppsett. Så kunne spart meg for de fleste Blazor relaterte problemene jeg møtte på. Jeg kunne nok også vært enda mer strukturert i timeplanen og fordelingen av oppgaver, ettersom at jeg følte at jeg datt litt av etter hvert og gjorde ting litt mer etter behov, som for meg tyder på at ting ikke ble kartlagt godt nok.

Ellers så kunne jeg ha valgt ett rammeverk til frontend jeg hadde vært mer kjent med, som f.eks React. Dette hadde nok økt effektiviteten for frontend utviklingen, samtidig som at design og lignende nok hadde vært bedre, men jeg føler at jeg har lært masse av å benytte meg av Blazor og syntes at resultatet er bra og definitivt brukbart med tanke på tidsforbruket på en uke.

For min egen skyld så kunne jeg også jevnet ut arbeidet med dokumentasjonen litt, dette sparte jeg til ganske sent og endte opp med mye skriving på veldig kort tid. Å skrive litt og litt underveis hadde forenklet denne prosessen.