ARCADA

Refaktorisering av föråldrad källkod

Tommy Pettersson

Lärdomsprov

Informationsteknik

2023

Lärdomsprov

(Författare) Tommy, Pettersson

Refaktorisering av föråldrar källkod

Yrkeshögskolan Arcada: Informationsteknik, 2023.

Identifikationsnummer:

26875

Sammandrag:

Detta lärdomsprov är ett beställningsarbete för refaktorisering av en legacy kodbas. Den tidigare kodbasen var på AngularJS och .NET 4.0.1 ramverk. Eftersom att AngularJS är utdaterat och inte längre stöds från och med januari 2022 av skaparna av biblioteket, behövde applikationens kodbas skrivas om på. Detta delvis på grund av säkerhetsskäl som uppkommer då bibliotek och annat inte längre stöds och uppdateras av skaparna. För detta arbete har flertal olika verktyg använts. Som ett exempel är verktyget Jira, som använts för att hålla reda på de olika modulerna och kravspecifikationerna som har gjorts för detta arbete samt kommunikationsverktyg för ändringsförslag. Även Bitbucket har använts för versionshantering för kodbasen och de olika versionerna som har framställts under utvecklingen för att hjälpa ifall en bugg har uppstått och identifiera i vilken version det hände.

Programmeringen görs huvudsakligen i C# samt React med typescript. Unit-tests har gjorts i back-enden för att säkerhetsställa att dataflödet skickar och tar emot rätt data i det format som förväntas av applikationen. Frågeställningen för detta arbete är gjort på basis av beställarens målsättning av applikationen. Slutresultatet för detta arbete uppföljer de kraven som beställaren har satt. Dessutom visade slutresultatet lovande resultat på frågeställningarna som ställdes för detta arbete.

Nyckelord:

Refaktorisering, React, .NET, React-Hook-Forms, AngularJS, TypeScript

Degree Thesis

(Author) Tommy, Pettersson

Refaktorisering av föråldrar källkod

Arcada University of Applied Sciences: Degree Information Technology, 2023.

Identification number:

26875

Abstract:

This degree project is a commissioned project for the refactoring of a legacy codebase that is based on AngularJS and the .NET 4.0.1 framework. Since AngularJS is outdated and no longer supported from January 2022 by the creators of the library, the application's codebase needed to be re-written due to security reasons that arise when libraries and other things are no longer supported and updated by the creators. For this work, several different tools have been used, such as Jira, which has been used to keep track of the various modules and requirement specifications that have been made for this work as well as communication tools for change proposals. Bitbucket has been used for version management of the codebase and the various versions produced during development to help in case a bug has occurred and identify which version it happened in.

The programming is mainly done in C# and React with typescript. Unit tests have been done in the back end to ensure that the data flow sends and receives the correct data in the format expected by the application.

The questions for this work are made on the basis of the client's objectives for the application. The result for this work followed up the requirements that the client has set for the writer and the result showed promising results on the questions that were asked for this work.

Keywords:

Refactoring, React, .NET, React-Hook-Forms, AngularJS, TypeScript

Innehåll

1 Inledning

Det här arbetet är ett beställningsarbete. Arbetets beställare har en applikation som används internt för låneansökningar. Den ena delen av applikationen är gjord i AngularJS(V.1.0.0) för frontend och .NET 4.0.1. för back-end.

Inspirationen för beställningsarbetet kommer från att AngularJS har blivit legacykod, och inte längre underhålls av utvecklare (AngularJs, 2022). Därav gjordes beslutet att uppgradera applikationen till nyare teknologier och skriva om en kodbas som är betydligt nyare. Genom att skriva om och förbättra både back-end och front-end till de nyare teknologierna och förbättra strukturen av koden, kan arbetet förbättra framtida prestanda för applikationen. Kodbasen har även översatts från finska/svenska/engelska till enbart engelska variabel namn samt funktions namn, då den tidigare kodbasen har varit en blandning av dessa eftersom flera olika utvecklare jobbat på applikationen.

Teknologierna utvaldes av beställaren, och förblev React 17, Typescript, Redux, React-Hook-Forms, som hädanefter hänvisas som RHF, samt MUI(Material-ui) v.5 som frontend. För back-end valdes .NET core 3.1, AutoMapper samt Entity ramverk, som hädanefter hänvisas som EF-work. Arbetet har använt sig av en ny struktur där man förminskar kontrollern till det enbart nödvändigaste, varefter den sedan kallar på olika service layers från business logiken där logiken för back-enden körs.

2 Centrala begrepp

Ramverk

I programmering är som en standardisering och template för specifika kodspråk, detta används för att göra programmeringen lättare och mer läsbar för grund koncept och metoder som används för att förenkla för andra utvecklare som jobbar på applikationen

Legacy

Betyder då applikationer eller bibliotek eller ramverk är äldre kodbaser som inte stöds eller utvecklas mera av tillverkarna.

Refaktorisering

Refaktorisering är då man skriver om en kodbas som existerar men är av legacy kod eller då man vill byta kodbas till något nytt språk, bibliotek eller skriva om det för att uppfylla nya funktioner som den gamla kodbasen inte stödjer men bibehålla grund funktionerna.

Loadtimes

Loadtimes i detta arbete är laddningstiden för applikationen utifrån en webb-applikations laddning på ens webbläsare.

Ligthweight

Är i arbetets sammanhang lättviktiga applikationer, de har inte stor storlek för biblioteket, det har bra loadtimes och är associerat att vara lättanvänt.

Re-render

Är då en webbapplikation skrivs om och framställer hemsidan igen. Då en användare fyller i ett dynamiskt formulär så framställer applikationen formuläret på nytt för de nya fälten eller liknande som läggs till.

Kontroller

Är i detta arbetes sammanhang för Microsofts .NET ramverk som använder sig av dem för att kontrollera hur applikationen hanterar förfrågningar från användarna.

3 Syfte och frågeställning

Syftet med detta arbete är att utforska hur applikationens prestanda och läslighet kan förbättras via ny teknologi. Då den tidigare teknologin använt för applikationen inte längre underhålls av utvecklarna är det kritiskt för företaget att refaktoriseria applikationen, då det med tiden kunde ha medfört säkerhetsrisker och buggar.

Frågeställning för arbetet var:
Vilka förbättringar i prestandan(loadtimes) har åstadkommits?
Vilka förbättringar i läslighet och livslängden har åstadkommits?

4 Planering av utveckling

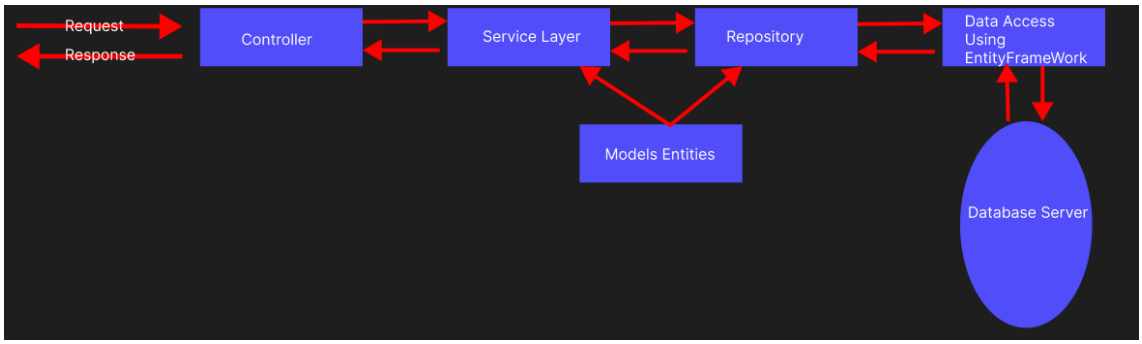
Planeringen av detta arbete började med att först gå genom den gamla kodbasen och strukturen.

Därefter delades sidan upp i segment i olika tickets på Jira där det listades upp kraven, vad som skall finnas på sidan och segmenten.

4.1 Planering av moduler

Då kraven var utlagda började jag med forskning över de olika teknologierna, vad som är den nya standarden och hur det går att förbättra de metoderna som har använts tidigare. Detta gjordes för att applikationen skall ha snabbare respons tid, vara mer lättläslig för nya utvecklare och ha längre livslängd.

Back-enden var den första modulen som planerades. Jag valde att använda mig av service layers, då det inte längre rekommenderas att sätta databas och data logik i kontrollers, eftersom detta försämrar livslängden på koden och gör den svårläst (Microsoft, 2022). Som illustrerat i grafen nedan, så görs en request till APIn i kontrollern som sedan skickar informationen till den designerade service layern, varefter den skickas till data access pointen som är EF-work, som sedan validerar användaren och gör en förfrågan till databasservern.



Figur 1 Struktur diagram

Det beslöts även att det krävs mera unit-tests för back-enden, då detta inte har varit en standard för applikationen i fråga tidigare.

Unit-tests är viktiga för att kontrollera vad för data skickas vidare, att rätt data kommer fram som har förfrågats ifrån APIn och att data kommer i rätt format och struktur som utvecklaren förväntar sig. Detta är ytterst viktigt då databasen kan ha andra namn i sig än utvecklaren vill ha i back-enden. Det ovannämnda förstärktes i detta projekt, då kravet var att översätta och använda enbart engelska namn på variabler och funktioner. Det beslöts att biblioteket AutoMapper var det lämpligaste för ifrågavarande arbete att använda sig av, eftersom detta bibliotek förenklar processen att konvertera den data som kommer till EF-work från databasen, och som sedan skickas till back-enden.

Frontendens moduler planerades igenom att använda sig av React 17 med TypeScript, Redux, RHF, Zod validation samt MUI v.5 för elements och styling. Valet av detta baserades på beslut gjort av arbetets beställare. Det valdes att använda sig av RHS efter att jag och en annan utvecklare forskade i vilket bibliotek som passade bäst för denna applikation, samt för förbättring av prestandan av applikationen. En vägande faktor var att RHS är ett väldigt ”lightweight” bibliotek samt förbättrar bundling-tid (optimering av laddningstider av webbapplikationer). Dessutom förminskar RHS re-renders och validerings uträkningarna.

Figur 2 Bundling jämförelse tagen ifrån Bundlephobia.com

4.2 Kravspecifikationer

För detta arbete krävdes användning av dynamiska formulär som ska klara av att ta emot data och fylla i information från olika databaser.

Back-enden behövdes för den nya modulens olika funktionaliteter, eftersom dessa måste skrivas om från den tidigare back-enden som använde sig av .NET 4.0.1, då denna hade allt i controllern. Den tidigare back-enden saknade även unit-test helt och hållet.

Beställaren önskade implementera mer unit-test som standard, samt en förbättring av back-endens prestation. Den nya standarden är att inte ha all logik i kontrollen, eftersom detta snabbt blir väldigt oläsligt och kan skapa långsam prestanda av applikationen. På grund av detta valdes det att kontroller enbart skulle sköta API-delen och göra anrop till olika delar av applikationen. En modell skapades i objekts projektet, som är en del av applikationen/solutionen som sköter logiken för modeller samt EF-work, som skapar och har kopplingen till databasen. Detta eftersom databasens fält är på finska och beställaren önskade hålla standarden att kodbasen skall ha engelska namn.

Eftersom det är skillnad på namnen, kommer applikationen inte veta att data som kommer till entityn (objekt som kommer från databasen) skall sättas till modellen på rätt ställen. På grund av detta behövdes det göras mapping (mappning av objekt via modeller) för det. Det valdes att använda biblioteket AutoMapper för detta, då detta bibliotek förenklar processen betydligt och gör koden mer lättläst då mapping kan bli väldigt komplexa.

Efter att modellen var skapad gjordes det en mapping profile i BusinessLogic applikationen. Sedan skapades service layern för logiken som används i API:et. Den skapades genom att göra de tre olika gränssnitt som behövs för denna modul GET/PUT/POST, där

vardera funktionen sköter dess respektive funktioner så som att hämta, uppdatera eller skapa data till databasen.

Då backend-funktionerna var skapade behövdes unit-tests för att kolla att den data som behövs skickas på rätt sätt samt kommer fram i det format som är förväntat i olika scenarion. Dessa skapades i backend.test projektet där det skapades tests för själva service layern med GET/PUT/POSTfunktionerna samt för AutoMapper profilen.

Arbetet i fråga använder sig av RHF samt MUI, för vilka så kallade custom hook components har skapats. Dessa använder sig av baskomponenter från MUI som har byggts om och tar in props och funktionalitet från RHF för att sömlöst integreras i applikationen i denna modul, samt andra existerande och framtida moduler.

```
export default function RadioGroups<
    TOption extends string | number | boolean,
    TFieldValues extends FieldValues,
    TName extends FieldPath<TFieldValues>
>({
    name,
    control,
    defaultOption,
    label,
    options,
    formControlProps,
    ...selectProps
}: RadioProps<TOption, TFieldValues, TName>) {
    const {
        field: { onChange, onBlur, value, ref },
    } = useController({
        name,
        control,
        defaultValue: defaultOption,
    });
    return (
        <FormControl {...formControlProps}>
            <RadioGroup
                defaultValue={defaultOption}
                id={name}
                value={value}
                onChange={onChange}
                onBlur={onBlur}
                ref={ref}
            >
                {options.map(option => <FormControlLabel key={`${name}-${option.value}`} value={option.value} control={<Radio sx={{
                    "&.Mui-checked": {
                        color: "#009430",
                    },
                }} />} label={option.text}></FormControlLabel>)}
            </RadioGroup>
        </FormControl>
    );
}
```

Figur 3 Custom compontent integrated with RHF and MUI

Som illustrerat ovan är denna komponent gjord enskilt i en grafisk mapp för att förenkla strukturen av applikationens läslighet för utvecklare. Det bör även noteras att den tar in props ifrån RHF och MUI via spread funktioner.

Formulären har även ett antal villkorliga regler, som beroende på vad användaren fyller i och väljer kommer visa och göra olika uträkningar. Detta är behövligt då det i formulären används olika formler för att räkna ut de bästa lånevillkoren som användaren i fråga kan ansöka om.

Utöver logiken bakom den olika inputen samt uträkningarna, skall det även finnas funktionalitet för att uppdatera och skicka in formulären (PUT & POST-requests).
Dessa funktionaliteter används i knappar som ”save and continue”. Tanken bakom detta är att save är PUT-requesten som uppdaterar data i databas tables, om det är så att en eller flera fält har ändrats i lånapplikationen.

Continue knappen fungerar som så, att ifall det är en helt ny låneansökan som användaren ansöker om, och deras tidigare ansökan har blivit nekat eller arkiverad på egen förfrågan eller dylikt, så skickar applikationen en POST-request istället.

Jag implementerade en logik för att kontrollera detta; ifall det är en ny applikation eller ifall det är en uppdatering av en nuvarande applikation. Detta löstes genom att i back-end införa i modellen och controllen ett till datafält ”status”, ifall back-enden inte hittar data från de olika tables eller ifall kundens profil returnerar ett datavärde som indikerar att dens tidigare applikation är arkiverad. I detta fall returnerar den till frontenden ett värde mellan 0-1. Ifall värdet är 1 så existerar det data och en applikation för användaren i fråga och då oavsett om den trycker på save eller continue så körs PUT-requesten och skickar data tillbaka till back-enden.

Eftersom denna modul består av 3 formulär som blir sammanslagna då det skickas in, valde jag att följa reglerna för react hooks (reactJs, 2023).

Eftersom de olika formulären behöver samma data, och för att den skall laddas in tidigare, så skapades en initialLoad komponent. Denna sköter GET-requesten som laddar in den behövliga data från back-end APIn och services. När requesten har gjorts och returnerats, så aktiveras en custom hook som sätter ”status” mellan ”idle, loading och loaded”. Beroende på statusen av loading komponenten så render en circularProgress ifall (status ==

”loading”) då datan har hämtats från APIn så sätts den till ”loaded” och då render applikationen huvudformuläret.

Huvudformuläret innehåller controllen och implementationen av RHF, som sätter applikationens state som sparat via redux store och APIn till en resolver via Zod custom schema.

Då validering av inputs är viktigt, har RHF resolvers inbyggts för det här som kan användas med olika bibliotek. I detta projekt har det valts att använda Zod för dess funktionalitet och error hantering, istället för att använda det mer populära Yup som kräver extra funktioner för att sköta error hantering.

Då input fälten är bestämda på förhand för denna modul, var skapandet av ZodSchemat förenklat. Det strukturerades efter att back-end modulens modell var klar och sen imiterade den. På grund av att den skall fungera med Redux måste back-end model, interface i store samt schemat vara identiska för att de skall veta vart data skall fara och var den data hör till.

Då schemat var klart och exporterades som en type, behövde det läggas ut logiken för att få de tre olika formulären att ta in datan och vara kopplade mellan varandra, men ändå render åtskillt.

Detta löstes genom att skapa dem som custom components som används i huvudformuläret. Då huvudformuläret render så körs dess funktioner, varefter den kallar på de olika formulärens custom component och render dem.

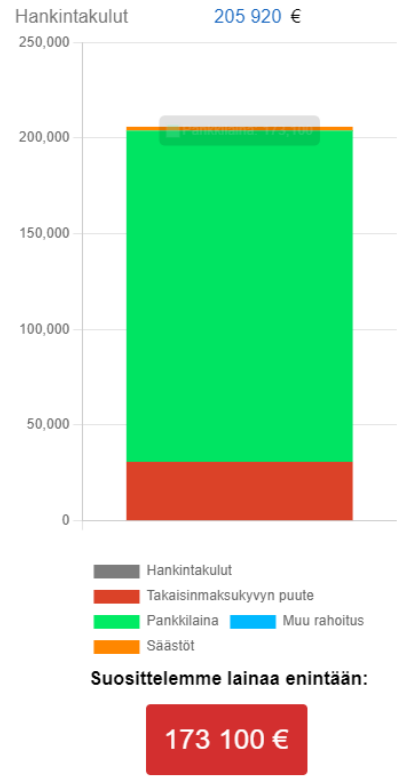
Detta valdes också för hur schemat är strukturerat och för att använda sig av RHF inbyggda funktion av watch() (React-Hook-Forms, 2023).

Figur 4 användning av useWatch()

Den ovannämnda funktionen useWatch() används i de tidigare nämnda RHF MUI komponenterna, för att få rätt värde från store/schema via namn samt för att få rätt värde till uträkningarna för att uppdateras i realtid. Detta på grund av att då användaren ska kunna ändra på fältens värde, så skall uträkningarna köras samtidigt och visa resultatet vid behov utan att behöva ha sidan att re-render. Det ovannämnda görs för att öka prestanda för applikationen.

En graf samt två summor ska visas vid sidan av formulären som skall uppdateras i realtid. För ifrågavarande arbete illusteraras detta i figur 5. Det fanns olika alternativ för att implementera detta, men då applikationens ena modul använde sig av React-CharJs tidigare och det är ett av de mest populära biblioteken för grafer(ReactChartJs, 2023), var det ett utmärkt val för ifrågavarande ändamål då det även är lätt integrerat och inte kräver mycket prestanda av applikationen. Då data för alla uträkningar som skall användas i grafen redan fanns i applikationen, behövde det utvecklas lite mer funktionalitet i applikationen store. Detta är till exempel funktionaliteter som kan hantera de olika uträkningarna som sköttes från formulären, och uppdaterades därav automatiskt i realtid ifall användaren ändrade på input data i formulären. Det ovannämnda sköttes via det tidigare nämnda watch()

funktionen samt stores funktionalitet, som använder sig av custom component som sköter uppdatering ifall något ändras och den ritas i realtid.



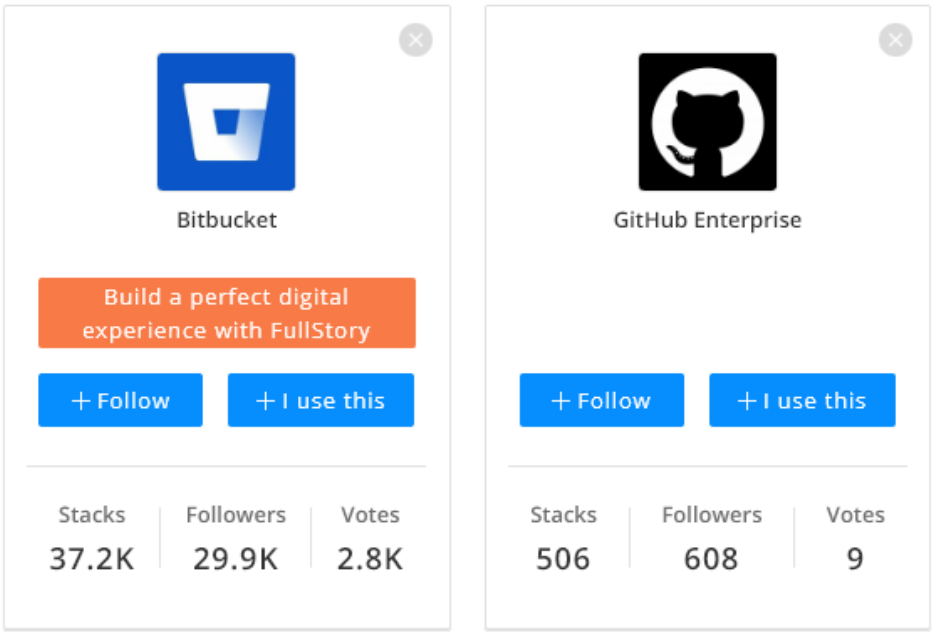
Figur 5 Barchart using ChartJS, Redux and custom hooks

4.3 Verktyg

De olika verktygen som har används i detta projekt är Bitbucket för versionshantering, Jira för projekthantering och felsökningsspårning för att loggföra ifall applikationen har någon bugg i sig och Confluence för att spara och samla dokumentation om projekten.

4.3.1 Versionshantering Bitbucket

Ett av verktygen som användes i detta projekt är Bitbucket. Detta användes för repository och versionshantering. Bitbucket används framför allt av företag då det är väldigt finjusteringsbart för företagen i fråga och har mycket bra Jira integrering, inbyggt CI/CD (continuous integration and continuous delivery). Enligt stackshare.io så används Bitbucket i 98,6% mer stacks än GitHub Enterprise.



Figur 6 Bitbucket vs GitHub Enterprise https://stackshare.io/stackups/bitbucket-vs-github-enterprise 08/03/2023

Bitbucket och andra versionshanterings- och samarbetsprogram är väldigt bra att använda för att hålla koll på de olika versionerna, samt ifall flera utvecklare jobbar på samma applikation men på olika delar av den och de inte skall orsaka konflikter i filerna och applikationen.

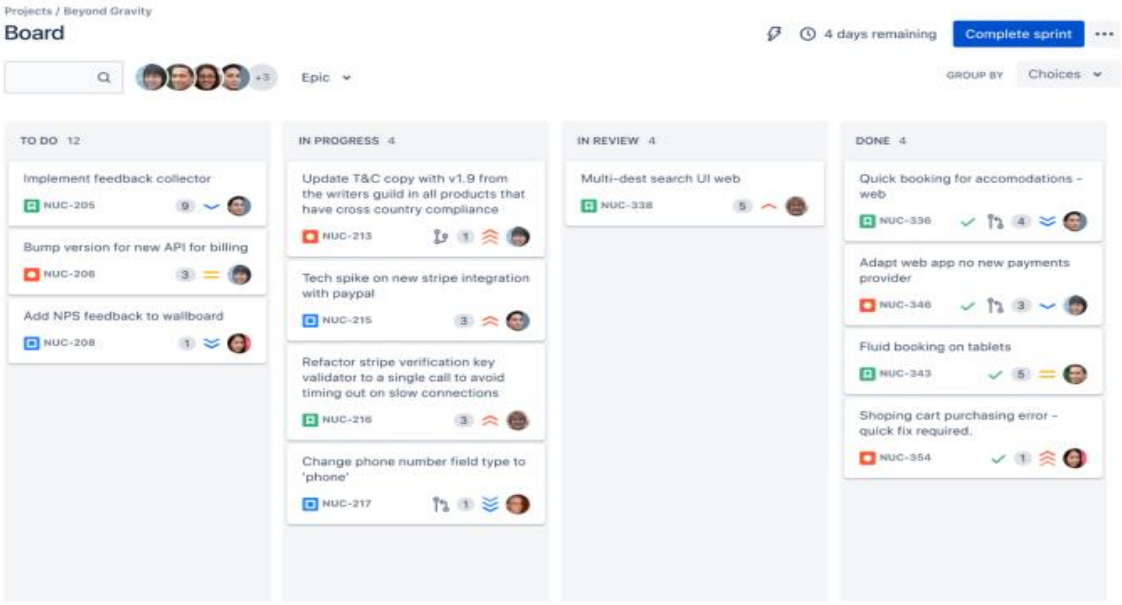
Den största fördelen med att använda Bitbucket är att kunna lagra koden på en plats som är säker och tillgänglig för alla i ditt team, så de kan få tillgång till hela eller delar av koden utan problem. Det är även väldigt lätt att hålla koll på alla ändringar och göra rollbacks vid behov. Då Jira är så bra integrerad kan du se de tickets som är öppna för problemen eller buggarna i Bitbucket som gör felsökning och konversationer lättare angående dem vid en pull request (Polontech, 2019).

4.3.2 Jira

Jira Software är en del av en familj av produkter som är utformade för att hjälpa team av alla typer att hantera arbetet. Ursprungligen designades Jira som en bugg- och problemspårare, men idag har Jira utvecklats till ett kraftfullt arbetshanteringsverktyg för alla typer av användningsfall, från krav och testfallshantering till agil mjukvaruutveckling. (Atlassian, 2023)

Jira lanserads 2002 som ett rent spårningsprogram för felsökning för mjukvaruutvecklare. Under åren har flertal teams även utanför it världen börjat använda Jira för att spåra och logföra vilka som helst för problem som teamen kan möta.

En av de bästa möjligheterna med Jira är hur du kan utnyttja agila arbetsmetoder genom en lista/listor som håller koll på vad behöver göras för de teamet, vem jobbar på vad, vad är under review och vad som är klart och har gått ut till produktion.



Figur 7 Exempel på Jira teams workflow (project-management.com 08/02/2023)

Jira användes i detta projekt då det är standardiserat i företaget som beställde arbetet, för att hålla koll på vilken utvecklare sköter vilka utvecklings-, defect- och bugg-tickets. Detta gör att teamet vet vem som sköter vad och vem de kan fråga ifall de stöter på något liknande fel eller modul som en annan utvecklare kanske har påbörjat den processen re-dan.

Eftersom detta arbete gäller refaktorisering av en äldre kodbas, har funktionaliteten i den äldre koden använts i flera moduler av applikationen. I och med detta har Jira hjälpt med att hålla koll ifall någon redan har skrivit om en del av koden som skall användas i den nya kodbasen.

Jag valde att fördela upp projektet i olika mindre moduler för att kunna skapa mindre uppgifter för att förenkla utvecklingen av projektet, samt för dokumentation och testning då modulerna är redo för det i de olika miljöerna. På så vis vet man vad som har gjorts i vilken branch och version.

Då en modul är klar så görs en pull request och då den har blivit godkänd av någon i teamet så sätts den i en ny miljö för testning. I och med detta så flyttar vi ticketen i Jira från ”in progress” till ”ready for testing”.

Då detta projekt var väldigt omfattande, och andra utvecklingsprojekt och buggfixande gjordes samtidigt, kunde jag inte sätta någon av modulerna till testning förrän alla var redo för det och fick godkännande av ledningen. Ifall det hittades defekter i applikationen efter att den blivit godkänd och testningen inletts, så skapades en ny ticket som var kopplad till den ursprungliga ticketen som en ”defekt ticket” för att hålla reda på vad för defekter som finns, samt vilka som är under bearbetning och vilka som är fixade och invänta testning och som sedan är redo för produktion.

4.3.3 Confluence

Confluence är ett verktyg för samarbete i de olika teamen. I Confluence sparar man konversationer, dokumentation och uppdateringar som berör de olika applikationerna man har under arbete. Den stora fördelen med Confluence är hur det är strukturerat med sidhierarki. En annan betydande fördel är även hur extremt anpassningsbar den är för användarna.

Confluence är gjort av Atlassian och är kompatibelt med integrering av olika applikationer såsom: GitHub, Google Drive, Jira, Gitlab, Bitbucket med mera. Detta hjälper stort då man för dokumentation över applikationer eller listor över olika team som jobbar på projekt, då man kan anpassa vilka som har rättigheter att se de olika dokumenten. I och med detta kan du lätt linka ett stycke i dokumentationen till en ticket som är skapad i Jira för referens.

5 Back-End refaktorisering

I detta projekt så har back-enden för den tidigare kodbasen haft .NET 4.0.1 som ramverk. Den nya applikationen är baserad på .NET core 3.0.1 och därav krävdes omskrivning av koden till nya metoder.

Skillnaden mellan .NET 4.0.1 och .NET core 3.0.1, är att .NET core är en open-source generell utvecklingplattform som är cross-plattform kompatibel. Detta betyder att applikationen som utvecklats kan köras på Windows, Linux och MacOS utan större problem, medan .NET ramverk fungerar enbart på Windows OS. .NET core stöder även flertal programmeringsspråk och microservices medan .NET ramverk inte gör det.

.NET core är även i storskaliga projekt bättre, då skalbarheten är betydligt effektivare jämfört med .NET ramverk. (Scaler Academy, 2022)

Det har även implementerats mer unit-testning under detta arbete för att säkerhetsställa och ge kodbasen en längre livslängd för nya utvecklare. Då något ändras i applikationen eller ny eller annan data läggs till, så finns de där för att kolla att data kommer ut i det formatet man har förväntat sig.

AutoMapper har implementerats i en större grad på grund av att databaserna har finska kolumnnamn, vilket kan skapa förvirring samt blir ett problem för utvecklare som inte kan språket bra. Detta innebär även att det finns flertal olika variabel och funktionsnamn i applikationen i stället för standardiseringen av engelska i kodbaser.

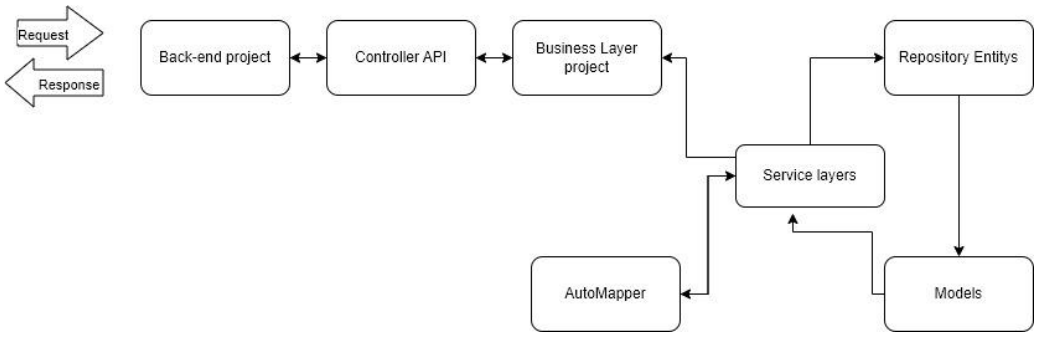
AutoMapper förenklar även mappningen mellan EF och modellerna i back-enden, då namnen har ändrats och är återanvändningsbara.

5.1 Service layers

En stor del av refaktoriseringsarbete för back-enden var att skriva om logiken för data hämtningen från databasen, i stället för att ha både kontroller logiken samt data hämtnings logiken i kontroller, vilket var gammal praxis som inte rekommenderas av Microsoft mera (Microsoft, 2022). I stället skall man dela upp logiken mellan controllen som sköter logiken och flödet av logiken till APIn, medan service layers är byggt i en annan del av applikationen i ett projekt som sköter all businesslogik.

I businesslogik projektet av applikationen så sköter den all datavalidering och AutoMappers profiler. Detta är rekommenderat för prestanda samt livslängden av applikationen, på grund av att ifall man sätter all logik inom kontroller blir funktionen väldigt stor och svår att kontrollera efter fel för validering av data. Även läsbarhet av funktionen blir sämre och därav försvårar det för andra utvecklare som kommer jobba på applikationen i framtiden.

Service layers hjälper även applikationen med återanvändbara funktioner, då delar av den kan måsta användas på mer än ett ställe. Istället för att skriva om liknande funktioner som hämtar och modifierar data, så använder man sig av de existerande funktionerna i service layern för att återanvända dem på de olika platserna. Detta förenklar processen genom att ifall data ändras, så ändras den för alla funktioner samtidigt från service layern. I och med detta behöver man inte leta och ändra varenda funktion i kontrollers som använder sig av den data.



Figur 8 Dataflow diagram over service layers

5.2 AutoMapper

Detta projekt har använt sig av AutoMapper, ett bibliotek som använder sig av en konfigurations-API för att definiera en objekt-objekt mappingsstrategi. AutoMapper använder sig av en konventionsbaserad matchingsalgoritm för att matcha källan till destinationen. AutoMapper är inriktad och används inom modellprojektions scenarier för att förenkla komplexa objektmodeller till DTO:er och andra enkla objekt var design är mer lämpad för serialisering, kommunikation, meddelande eller anti-korruptionslager mellan domänen och applikationslagret. (AutoMapper documentation, 2017)

Applikationen använder sig främst av AutoMapper för att hantera mappingen mellan de olika tabeller i databasen som går till EF-work och sedan ut till frontenden. Då databaserna använder sig av finska namn ibland, försvåras kontinuiteten för applikationen ifall nya utvecklare börjar jobba på den och inte kan finska flytande, då AutoMapper i sina AutoMapper.Profiles kan initieras vid applikations start. Detta i sin tur gör att de olika projekten i applikationen kan ärva och använda sig av dem. Man kan med översättningen automatiskt göra mappings som leder AutoMappers algoritm till vilka objekt och deras variabelnamn som hör till vilken data/DTO-modell.

```
CreateMap<Lainatarves, AcquisitionCosts>()
    .ForMember(dest => dest.PurposeOfLoanList, opt => opt.MapFrom(src => src))
    .ForMember(dest => dest.PurposeOfLoan, opt => opt.MapFrom(src => src.LainanTarkoitus))
    .ForMember(dest => dest.FirstHomePercentage, opt => opt.MapFrom(src => src.EnsiasuntoProsentti))
    .ForMember(dest => dest.OtherExpenses, opt => opt.MapFrom(src => src.MuutKulut))
    .ForMember(dest => dest.TransferTax, opt => opt.MapFrom(src => src.Varainsiirtovero));
```

Figur 9 AutoMapper profile example

Då AutoMappers profiler blir uppsatta och modifierade för applikationen i fråga under business logic projektet, som nämnts tidigare i arbetet, behöver det sättas upp endast en gång. Efter detta kan de sedan återanvändas av utvecklare i framtiden för att minimera dubbel arbete och återupprepade uppgifter, som att göra mappings för en EF-work modell till en data model som används i back-enden som redan existerar och har använts tidigare.

5.3 Minimize kontroller

Att hålla kontrollerna rena och snygga är något vi har lärt oss att vi borde göra första gången vi har snubblat på MVC-mönstret (Model-view-controller). När projektet dock växer och andra teammedlemmar går in i projektet, kan det gå överstyr. Detta gäller speciellt då deadlines måste hållas, vilket tyvärr är så gott som hela tiden. (code-maze, 2022)

Den tidigare applikationens kontrollers hade mycket logik och kod i sig, som utförde funktioner, datahämtning och kallade på andra servicelayers. Det gjorde att koden blev svår för nya utvecklare att förstå, och att de nya utvecklarna inte nödvändigtvis kunde tyda vad som händer och vad som gör vad.

Då refaktorisering till det nya projektet började, fördes diskussioner mellan teamet hur vi skulle gå till väga med detta. Efter att ha kollat på dokumentationen från Microsofts sidor om .NET och MVC mönster, så bestämde teamet att varje ny modul som skulle skrivas om skulle implementera den nya logiken som rekommenderades. Vi skulle följa samma kodstruktur för att hålla det i linje med varandra.

Jag valde att följa det nya mönstret som rekommenderas och gjorde kontrollers för detta projekt så små och lättläsliga som möjligt. Detta eftersom att de sköter all logik i de olika servicelayers som skrevs om i tidigare stycken av detta arbete.

Kontrollerfilen är nu strukturerad så att den tar in serviceinjektions i början av modulen, som sedanan används i de olika funktionerna i kontrollern.

Kontrollern sköter enbart logiken och funktionerna för en modul i taget. För detta projekt så sköter den dvs de olika förfrågningarna (GET, PUT, POST) för lånebehov. Då en requests görs till back-enden, går den till kontrollern som sedan kallar på de service layers som sköter data hämtningen från EF-work. Dessa mappar sedan dem via de anpassade AutoMapper profilerna som mappar data till DTO-modellerna, som går tillbaka till servicelayern som sedan i sin tur sköter undantagshanteringen för att säkerhetsställa ifall något går fel. Efter detta returneras rätt felmeddelande och det loggas, eller ifall requesten

går igenom så loggas det också vad som har hänt. Detta sköts från servicelayern för att hålla kontrollern så liten som möjligt.

5.4 Unit-testing

Unit-testning är väldigt nödvändigt att använda sig av i små som stora expanderande projekt.

Unit-testningen är nödvändig för att säkerhetsställa att alla funktioner, data och mapping fungerar som det skall då det används i back-end. Frontend unit-testning är lite annorlunda och kollar på att events, responsivitet och annat fungerar som det skall då användaren är på applikationen eller hemsidan.

Arbetet valde att utöka och förbättra unit-testningen då refaktorisering gjordes till det nya projektet. Det valdes att använda sig av XUnit samt Moq för att bygga upp unit-testerna där man skapar skenmiljöer, för att försäkra sig om att de olika funktionerna i modulerna fungerar. Varje test är separat per modul och innehåller enbart den logik som kontrollern har. På det viset så kan man testa kontroller, servicelayer, mappingen samt undantagshanteringen.

Unit-testningen sköttes i ett separat projekt inom applikationen, där testerna byggdes och utfördes sen via IDEns integrerade testningsverktyg (Visual Studio 2022).

```
using Moq;
using UnitTest_Mock.Controllers;
using UnitTest_Mock.Model;
using UnitTest_Mock.Services;
using Xunit;

namespace UnitTesting
{
   public class EmployeeTest
    {
        #region Property
        public Mock<IEmployeeService> mock = new Mock<IEmployeeService>();
        #endregion

        [Fact]
        public async void GetEmployeebyId()
        {
            mock.Setup(p => p.GetEmployeebyId(1)).ReturnsAsync("JK");
            EmployeeController emp = new EmployeeController(mock.Object);
            string result = await emp.GetEmployeeById(1);
            Assert.Equal("JK", result);
        }
        [Fact]
        public async void GetEmployeeDetails()
        {
            var employeeDTO = new Employee()
            {
                Id = 1,
                Name = "JK",
                Desgination = "SDE"
            };
            mock.Setup(p => p.GetEmployeeDetails(1)).ReturnsAsync(employeeDTO);
            EmployeeController emp = new EmployeeController(mock.Object);
            var result = await emp.GetEmployeeDetails(1);
            Assert.True(employeeDTO.Equals(result));
        }
    }
}
```

Figur 10 Exempel unit-test c-sharpcorner

I figur 10 visas ett exempel på hur man sätter upp unit-tests via XUnit och Moq. Detta projekts tester har struktureras på liknande sätt och sätter upp en skenmiljö av miljön samt databasmiljön, för att sen utföra ett flertal olika tester på de olika funktionerna GET, POST, PUT för att säkerhetsställa att det inte uppstår några fel.

Jag införde även unit-testning på AutoMapper profilerna för att säkerhetsställa att mappingen mellan EF-work och DTO-modellerna (data transfer object) utförs rätt och att rätt datavärde blir satt till rätt ställe. Detta gjordes eftersom mappingen kan gå igenom utan fel, trots att datavärdet sätts på fel ställe eller har ignorerats helt och inte kommer till DTO-modellen. Detta leder i sin tur till att det då inte returnerar rätt då de behövs i de olika funktionaliteterna som använder sig av den modellen.

Unit-tests är dock inte felsäkra och används som en referenspunkt vid utveckling då ny eller ändrad funktionalitet utvecklas. Detta är då för ökad läslighet, utveckligsmöjlighet samt livslängden på applikationen. Testning vid ändrad funktionalitet heter

regressionstestning och detta görs då utvecklaren eller en ny utvecklare utökar eller ändrar på den existerande funktionaliteten för att säkerhetsställa att ändringen inte har introducerat nya defekter i kodbasen och därmed förstör andra delar av koden.

6 Frontend refaktorisering

Jag har i detta projekt skrivit om en applikation vars frontend tidigare varit skrivet i AngularJS V.1.0.0, och som nu i och med detta beställningsarbete skrivits över till ReactJS V.17, RHF, MUI V.5, TypeScript och Redux.

Valen av dessa bibliotek är gjorda för funktionerna de medför till arbetet och de fungerar bra att integrera med varandra.

Valet av ReactJS kombinerat med TypeScript gjordes av beställarna då AngularJS inte mera stöds och ReactJS med TypeScript erbjuder bra integrering. De sistnämnda är även båda väldigt lättviktiga bibliotek och kan använda sig av återanvändningsbara komponenter som gör applikationens livstid och utveckling lättare och ökar prestandan av den.

React Hook Form är ett bibliotek som hjälper till att hantera komplexa formulär. Den har utmärkt prestanda, är superlätt, har noll beroenden, kan enkelt integreras med olika React UI-ramverk som Material, Antd, och ger ett intuitivt API och en utmärkt utvecklarupplevelse. (Hygraph, 2022)

Valet att använda sig av Redux gjordes då det biblioteket erbjuder bra integrering med Reacts egna states och som en samlingsplats för alla olika datalager som används i frontenden.

De flesta av modulerna i projektet påverkar inte varandra direkt, men gör deras data det så kommer Redux till bra användning för att hålla kolla ifall staten har ändrats eller ifall ett formulär har sparats och skickat data till databasen som sedan används i en annan del av applikationen.

Redux är väldigt lämpligt för en storskalig applikation som flera utvecklare arbetar på samtidigt, ifall datalagren används på flera ställen av applikationen samtidigt. MUI V.5 är ett bibliotek som är uppbyggt för att använda React UI komponenter som implementerar Googles Material Design (Material-ui, 2022).

```
<RadioGroups
    label={t("LoanNeed.AcquisitionCostsForm.FirstHomeBuyer")}
    formControlProps={{
        sx: {
            root: {
                color: 'green',
            },
            colorSecondary: {
                '&$checked': {
                    color: 'green',
                },
            },

        },
        variant: "standard",
    }}
    control={control}
    name="acquisitionCosts.purposeOfLoanList.firstHomeBuyer"
    options={firstHomeRadio}
    disabled={purposeOfLoan == 5}
    error={Boolean(
        errors.acquisitionCosts &&
        errors.acquisitionCosts.purposeOfLoanList?.firstHomeBuyer
    )}
/>
```

Figur 11 Anpassad MUI hook

Valet för att använda sig av MUI V.5 baserade sig på hur lätt det är att integrera i de andra biblioteken så som React-hook-forms. En vägande faktor var även hur lätt det är att skapa återanvändbara UI komponenter som används över hela applikationen som följer ett globalt färgschema samt design val.

I figur 11 visas ett exempel på detta där en återanvändningsbar komponent har gjorts som implementerar MUI, React Hooks och RHF. Denna komponent kan återanvändas genom hela applikationen med egna variabler och data. Istället för att skriva om den och designa den varje gång som blir mer utfyllnadskod och gör applikationen större i onödan, så är den strukturerad så att den går att återanvända.

6.1 React hooks med TypeScript

Hooks är ett nytt tillägg i React 16.8. De låter dig använda tillstånd och andra React-funktioner utan att skriva en klass. (React, 2022)

React hooks är bakåtkompatibel och kan användas i redan existerande kodbas eller som i detta arbete refaktorisering.

React hooks kan användas för att spara och sätta states inom modulerna. React hooks kan även implementeras med andra bibliotek för att medföra ny funktionalitet som använder sig av de olika biblioteken.

```
const [
    purposeOfLoan,
    apartmentType,
    firstHomeBuyer,
    purchasePrice,
    debtShare,
    purchasePricePlot,
    firstHomePercentage,
    otherExpenses,
    quotation,
    transferTaxAttention,
    showBarChart,
] = useWatch({
    control,
    name: [
        "acquisitionCosts.purposeOfLoan",
        "acquisitionCosts.purposeOfLoanList.apartmentType",
        "acquisitionCosts.purposeOfLoanList.firstHomeBuyer",
        "acquisitionCosts.purposeOfLoanList.purchasePrice",
        "acquisitionCosts.purposeOfLoanList.debtShare",
        "acquisitionCosts.purposeOfLoanList.purchasePricePlot",
        "acquisitionCosts.firstHomePercentage",
        "acquisitionCosts.otherExpenses",
        "acquisitionCosts.purposeOfLoanList.quotation",
        "acquisitionCosts.purposeOfLoanList.transferTaxAttention",
        "showBarChart"

    ],
});
```

Figur 12 Hook med Zod schema validation

Ett exempel där anpassade hooks har använts i detta arbete är vid hämtning av data. För ifrågavarande syfte gäller att då det skapas en hook, som är en state av en string som ändras vid datahämtning via APIn, initieras den som IDLE då funktionen inte är i användning. Då funktionen kallas för att hämta data, så sätts den vid GET-requesten till FETCHING. Då data har hämtats från requesten och blivit godkänd via

undantagshanteringen så sätts statusen till FETCHED, för att indikera att den data är hämtad och sparad i den state som behövs. Detta är väldigt användbart geom hela applikationen då olika states behövs men inte är sparade i schemat, eller då de behövs för validering att en handling har skett och är klar.

Det tidigare exemplet har använts i applikationen för de custom komponenter som innehåller de olika formulären och UI-komponent som skall visas. Ifall till exempel datahämtningen och sparningen inte är klar, så visas en laddningssymbol som indicerar för användaren att det laddas. Då statusen ändras från FETCHING till FETCHED så vet applikationen via regler som är insatta, att den då skall render ut de komponenter som behövs.

```
function LoanNeedInitialLoading() {
    const { classes } = useStyles();
    // take profile id from route
    const match = useRouteMatch<{ profileId: string }>(
    );
    const profileId = toNum(match && match.params.profileId, -1);

    const { InitLoanNeedLoadingStatus, fetchLoanNeed } =
        useInitExistingLoanNeed(profileId);

    useEffect(() => {
        fetchLoanNeed();
    }, [profileId]);

    if (InitLoanNeedLoadingStatus !== "fetched") {
        return (
            <div className={classes.container}>
                <div className={classes.contentContainer}>
                    <CircularProgress />
                </div>
            </div>
        );
    }
    return <LoanNeedForms />;
}

export default (LoanNeedInitialLoading);
```

Figur 13 Custom loading av component

TypeScript har använts i detta projekt och integrerats med Reacts kodbas för att förbättra säkerheten och utvecklingen. TypeScript är ett hårt typat programmeringsspråk som bygger på JavaScript, vilket ger bättre verktyg i alla skalor (TypeScript, 2022).

TypeScript lägger till ytterligare syntax till en kodbas, eftersom TypeScript integrerar väldigt bra med kodeditors som Visual Studio Code och dylikt. TypeScript fungerar även väldigt bra med intelliSense, som hjälper under utvecklingen med att säkerhetsställa att funktioner och klasserna får de datatyper som är förväntade att användas där.

6.2 React Hook Forms

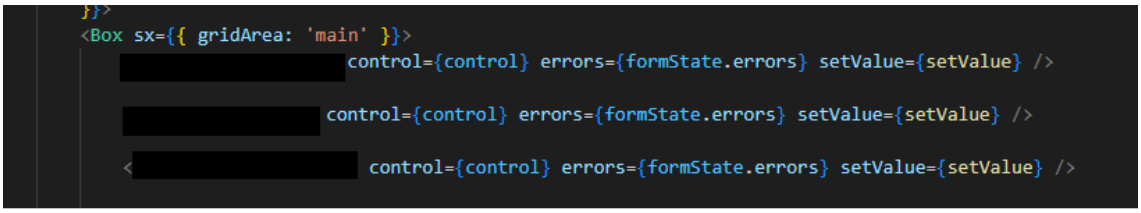
En av de bättre funktionerna med RHF, är att det går att implementera olika validationsscheman, Redux och andra bibliotek, vilket gör implementeringen mer sömlös. Genom att applikationen använder sig av RHF, ökar även dess skalbarhet eftersom man kan återanvända flera komponenter.

Genom användning av schemavalideringar, Redux och custom komponenter så känner RHF och IntelliSense igen variabler som finns i schemat. Implementationen av detta kan ses i Figur 12. I denna figur kan man under Name, som pekar på vilket dataobjekt som denna komponent menas att användas för, se hur funktionen gör en validering via error hanteringen som kollar via schemat ifall användarinput är korrekt eller strider mot schemareglerna.

En funktion som användes i detta projekt är att RHF kan hantera mer komplexa och flera formulär samtidigt från custom komponenter.

```
const { control, formState, handleSubmit, reset, setValue } = useForm<LoanNeedSchema>({
    resolver: zodResolver(loanNeedSchema),
    defaultValues,
    shouldFocusError: false
})
```

Figur 14 React-hook-form create



Figur 15 Flertal form kontrol med custom forms

Som illustrerat i Figur 15 ovan, så sköter denna del tre olika forms som skickar tillbaka informationen och data till huvud formuläret. Detta i sin tur sköter valideringen och sedan PUT/POST-request, som skickas till backenden för att hanteras och sparas i databasen.

7 Metod

Metoden som användes i detta arbete är kvalitativ undersökning. Detta tillämpades genom att samla data för de två olika versionerna av applikationen, samt teknologin de använder sig av.

Den data som samlades och jämfördes är requests, transffered, resources, finish, DOMContentLoaded och Load. Den har samlats in igenom att i de olika miljöerna.

Datainsamlingen inleddes med att gå in i den miljö som användes, och använda funktionen empty cache and hard reset för att säkerhetsställa att inget är sparat i cache, vilket kunde ha förvrängt den initiala data vi samlar in. Funktionerna är inbyggda i webbläsarna och för ifrågavarande arbete gjordes detta i Chrome. Jag har sedan samlat in data för varje refresh på hemsidan, för att efter det kunna göra en statistisk jämförelse med de olika variablerna. Detta för att få fram ett resultat mellan de olika teknologierna som är använda i applikationerna.

Datainsamlingen gjordes manuellt genom att efter varje refresh på applikationen kolla värden som kan ses utifrån webbläsarens konsol under nätverks tabben. I dessa fastställs informationen över när varje request och rendering har färdigställts.

Verktygen som användes var Firefox webbläsare, för tillgång till applikationen pga att det är en webbapplikation, Firefoxs inbyggda konsol och nätverk tab för att få se data för varje refresh samt Microsoft Excel för att spara och göra uträkningar av insamlingen. Internetuppkopplingen har varit 1000/100 via Ethernet kabel då datainsamlingen skedde.

Statistiken är utförd genom att jämföra den data som blev insamlad, och som sedan blev inmatad manuellt in i Excel. Efter detta ställdes den insamlade informationen upp och det uträknades medeltal, minsta värde och största värde utifrån de 46 datapunkterna som insamlades för varje miljö. Det var totalt sex olika miljöer, som fick en total på 276 olika datapunkter genom de olika miljöerna som sammanställdes i detta arbete.

Resultatet är sedan uppställt i en tabell där man kan se jämförelsen mellan de olika miljöerna, samt i en graf för att visualisera skillnaden mellan de olika miljöernas högsta-, lägsta- och medelvärde.

Denna data är självständigt samlad och används för att se ifall det är en förbättring mellan de olika teknologierna som använts i de två olika applikationerna. Detta eftersom arbetet handlar om refaktorisering en gammal kodbas, som använder sig av gammal teknologi som inte längre stöds av utvecklarna av den teknologin, till en ny teknologi som valts av beställarna av detta arbete.

8 Resultat

Resultatet för frågeställningen i detta arbete har fåtts fram genom samlad data över laddningstiderna för applikationerna i de olika miljöerna som använts (lokalt, internal testning och systemtestning).

Resultatet för prestandaändringen efter att refaktorisering har gjorts kan ses i tabell 1. Request har sänkts markant, vilket kan ses utifrån data som samlades in via applikationen som använde sig av .NET 4.0.1 och AngularJS hade ett medelvärde på 66, 17 och 20 medan data som samlades in via den nya applikationen som använde sig av React 17 och .NET core 3.1 hade ett medelvärde på 14. Resultatet påverkas av att React använder sig av bundling, medan AngularJS inte gör det, så en stor del av logiken i frontenden blir

minimized och bundlad i stället för att det skulle skickas flertal requests för de olika funktionerna.

Resultatet påverkas även av förbättringen av back-enden, där det tidigare under Back-end refaktorisering kaplitet diskuterades över minimeringen av kontrollers och användningen av service layers. Det diskuterades även om använding av AutoMapper för att kombinera all data till en modell som används och skickas till frontenden, vilket förminskar request antalen som skickas och tas emot.

För *Transferred* och *Resource* har den nya applikationen högre värden, då Reacts bundling metod fungerar bra för att öka prestandan. Dess bundling skickar dock alla funktioner och bibliotek till frontenden, vilket gör att överföringen blir stor i storleken. På grund av detta ökar även *finish* värderna.

För *DOMContentLoaded* och *Load* kan man se att i nästan alla aspekter är den nya applikationen med React och .NET core snabbare. Detta beror på användningen av React som skapar en virtuell DOM som hanterar Javascript koden och render ut det som HTML kod, som försnabbar och minskar tiden innan allt har laddat och render ut på applikationens frontend.

Som resultat för frågeställningen över hur prestandan har ändrats, så har applikationen blivit snabbare och därav ökat prestandan. Prestandan har även ändrats mot det positiva genom implementationen av mer Unit-testning. Genom ändringen av back-enden mot att använda fler service layers och service modeller, har applikationen som en helhet ökat i prestanda då mycket onödig kod kan tas bort, så att det inte behöver skrivas om för varje modul som använder sig av samma funktionalitet eller datahämtning.

För frågeställningen angående läslighet och livslängd har jag i detta arbete gått igenom varför ändringar har gjorts och hur det har ändrats för läsligheten och livslängden av denna applikation. En av de större ändringarna och förbättringarna är att den nya applikationen har skrivits om till att hålla en standardisering av hur back-enden samt frontenden är strukturerad och koden skall skrivas, samt att kodbasen nu är skriven på engelska.

En annan förbättring angående läslighet och livslängd är att modellerna för datahämtningen nu använder sig av AutoMapper för att översätta datavariablerna från finska namn till engelska namn, för att förenkla och hålla standarden på engelska namn i kod.

En av de övriga förbättringarna som gjord är omskrivningen till React, då AngularJS, som tidigare nämnt, är utdaterad och inte längre stöds av skaparna av det biblioteket. Sålunda har livslängded automatiskt ökat markant då React är ett av de mest populära biblioteken att använda inom frontend och inte kommer försvinna inom en snar framtid.

9 Diskussion

Detta arbete gjordes som ett beställningsarbete för en webbapplikation som används internt av företaget. Arbetet har varit väldigt givande och svårt emellanåt då applikationen redan var fungerande men utdaterad då det var skrivet i AngularJS och .NET 4.0.0.

Mycket av grundkoden och funktionerna kunde återanvändas men då beställarna önskade att det skulle överföras till den ny applikationen och samtidigt skrivas om i ReactJS och .NET core så bestämdes det att den skulle förbättras samtidigt. Detta utfördes igenom att implementera nya funktioner och bibliotek för att förbättra prestandan av applikationen som resultatet reflekterar. Då både back-enden samt front-enden skrevs om förbättrades även procedurerna för hämtningen av data via EF-work och kombinationen av nya DTO modeller och automapper. Även front-endens laddningstid och responstid ökade markant som kan ses utifrån resultatet samlat i detta arbete som kan ses i tabell 1.

Förbättringen av loadtimes kan ses utifrån tabell 1 och i graferna 20 och 21 där i de tre olika testmiljöerna har den nya applikationen presterat bättre i alla tre medelvärdet i resultatet. Detta bestyrks även då man tar i åtanke att det inte är stora extremvärden i största och lägsta värden i de olika miljöerna, förutom i systemtestnings miljön för den äldre applikationen som använder sig av AngularJS och .NET 4.0.0 vars *DOMContentLoade* hamnade på 11 290, då det andra närmaste i de andra miljöerna var på 2120.

Förbättringen på läslighet och livslängden av applikationen har ökat i min åsikt då de nya rekommendationerna, så som att minimera kontrollers och användingen av ReactJS nya

hooks är lättare att uppehålla för framtiden då AngularJS inte mera stöds och får uppdateringar. Det jag lärde mig mest igenom att göra detta refaktoriserings arbete och besvara denna fråga var vikten att skriva lättläslig kod som håller sig inom reglerna. Jag lärde mig även vikten över att alla håller en någorlunda rödtråd hur de skriver funktioner och klasser samt användning av samma språk för namnen på funktioner, variabler och liknande då en stor del av backend delen av arbetet gick ut på att skriva om och göra nya mappings profiler för DTO modellerna från de finska namnen till en mer standardisering av engelska som automatiskt ökar livslängden då nya utvecklare kan vara från andra länder där de inte pratar finska.

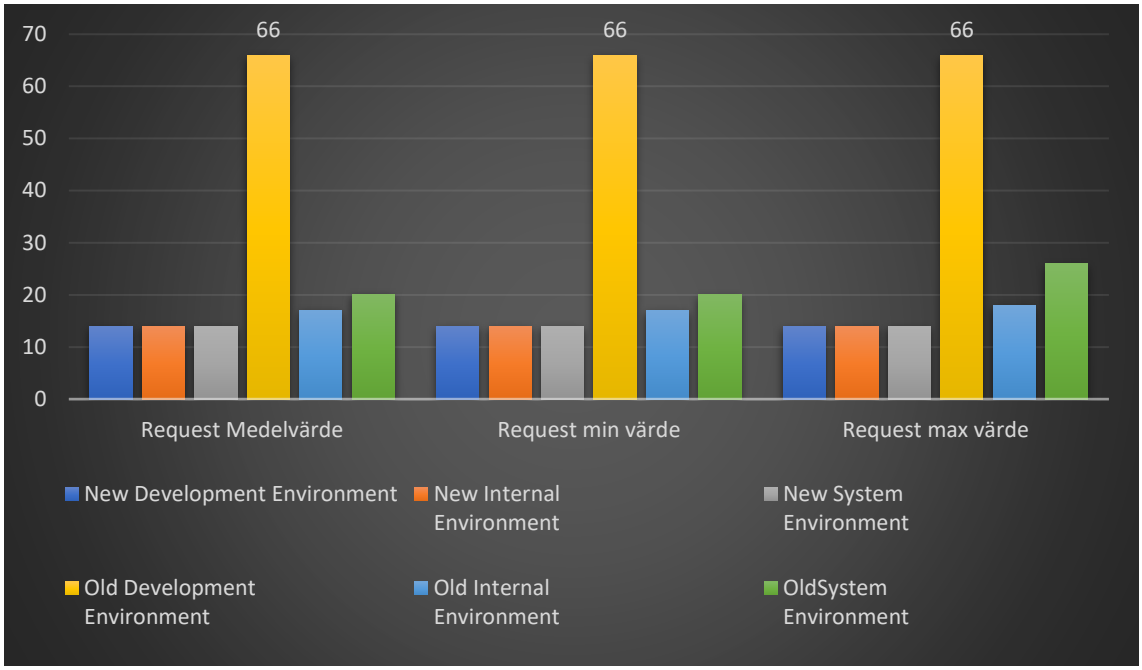
Under arbetes gång har det varit möten varje vecka med uppdateringar och statuskoll hur man ligger till och vad som har gjorts och vad planeras att göras tills näst. Under arbetets gång har jag följt min initiala plan och tidslinje för arbetet och hållit tidsramen som jag satt på mig själv och som beställaren satt på mig. Beställarna är väldigt nöjda med resultatet av arbetet då detta arbete har varit en storskalig applikations ändring och andra utvecklare har samtidigt överfört andra moduler av applikationen då jag gjorde detta arbete.

9.1 Tabeller

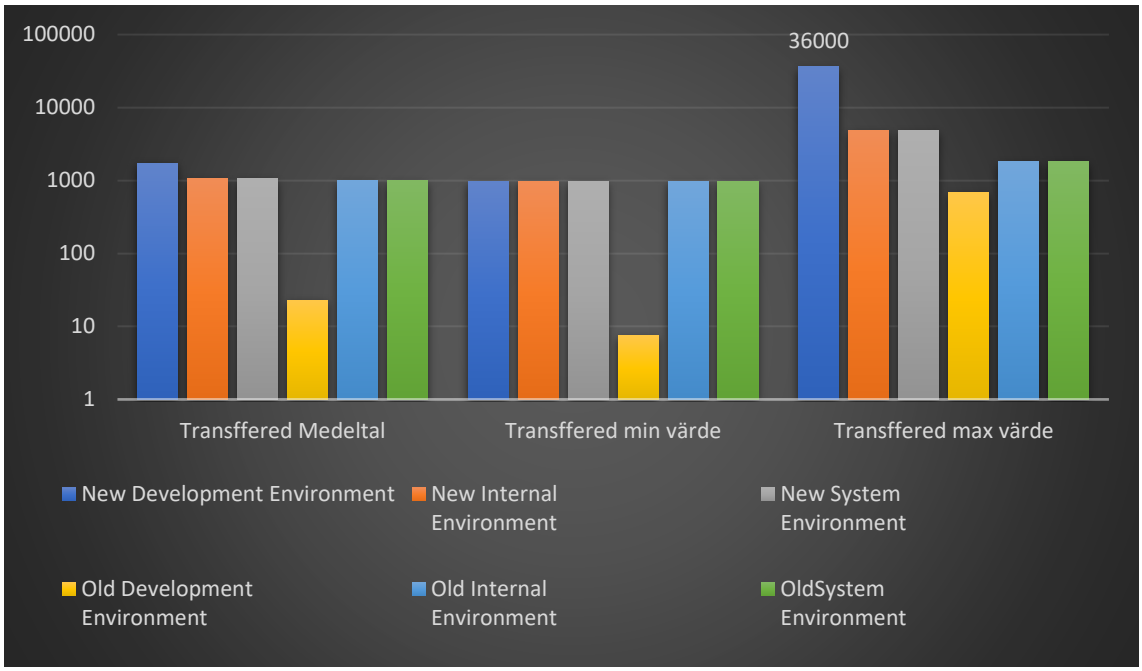
Tabell 1. Frontend load speed AngularJs med .NET 4.0.1 backend Vs React 17 .NET core 3.1 backend

	React + .NET core Development Environment	React + .NET core Internal Environment	React + .NET core System Environment	AngularJs v.1.0.0 .NET 4.0.1 Development Environment	AngularJs v.1.0.0 .NET 4.0.1 Internal Environment	AngularJs v.1.0.0 .NET 4.0.1 System Environment
Request Medeltal	14	14	14	66	17,02	20,13
Request Minsta värde	14	14	14	66	17	20
Request Största Värde	14	14	14	66	18	26
Transferred avg(KiloByte)	1728,59	1056,07	1056,13	23	993,91	993,91
Transferredmin(KiloByte)	967	970	970	7,5	976	976
Transferred max(KiloByte)	36000	4800	4800	683	1800	1800
Resource avg(MegaByte)	16,7	2,45	4,8	1,8	1,8	1,90
Resource min(MegaByte)	16,7	2,45	4,8	1,8	1,8	1,80
Resource max(MegaByte)	16,7	2,45	4,8	1,8	1,8	1,90
Finish avg(Seconds)	1,44	0,964	1,05	1,31	0,65	1,36
Finish min(Seconds)	0,88	0,548	0,81	0,85	0,50	0,50
Finish max(Seconds)	3,36	2,59	2,26	2,73	2,1	31,64
DOMContentLoaded avg Miliseconds	613,22	345,11	391,54	880,80	487,56	682,96
DOMContentLoaded min Miliseconds	381	236	253	467	324	344
DOMContentLoaded max Miliseconds	2120	1910	1600	2120	1930	11290
Load milisecond avg	712,24	489,91	512	878,78	673,67	832,41
Load min milisecond	445	328	357	465	499	484
Load max milisecond	2150	1930	1630	2120	2060	11300

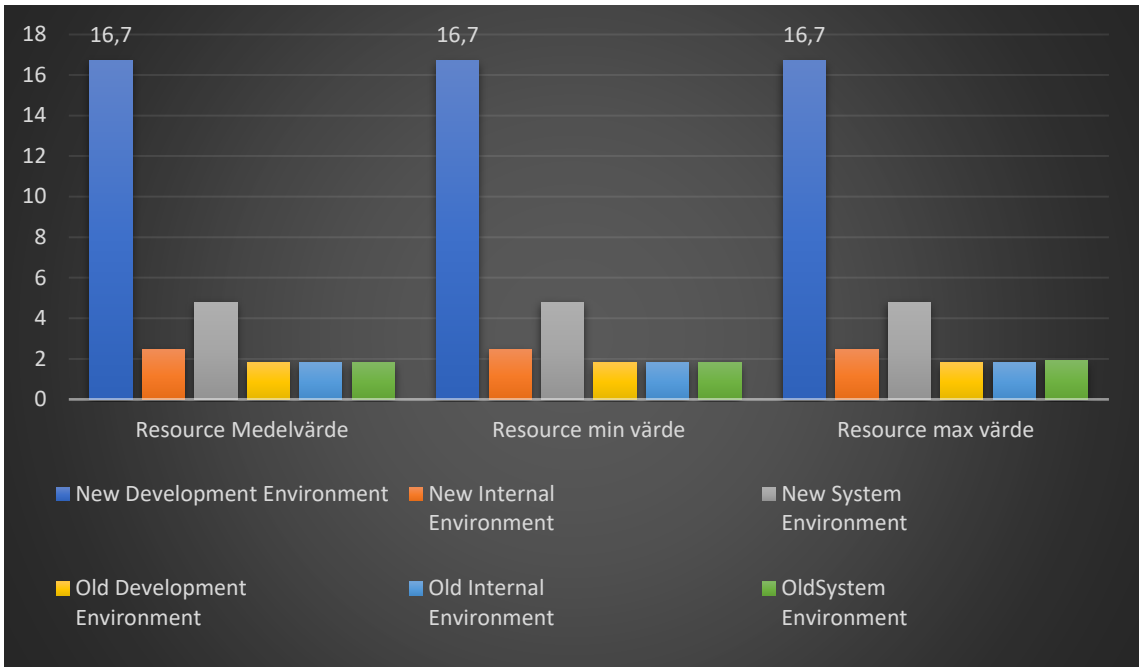
9.2 Figurer



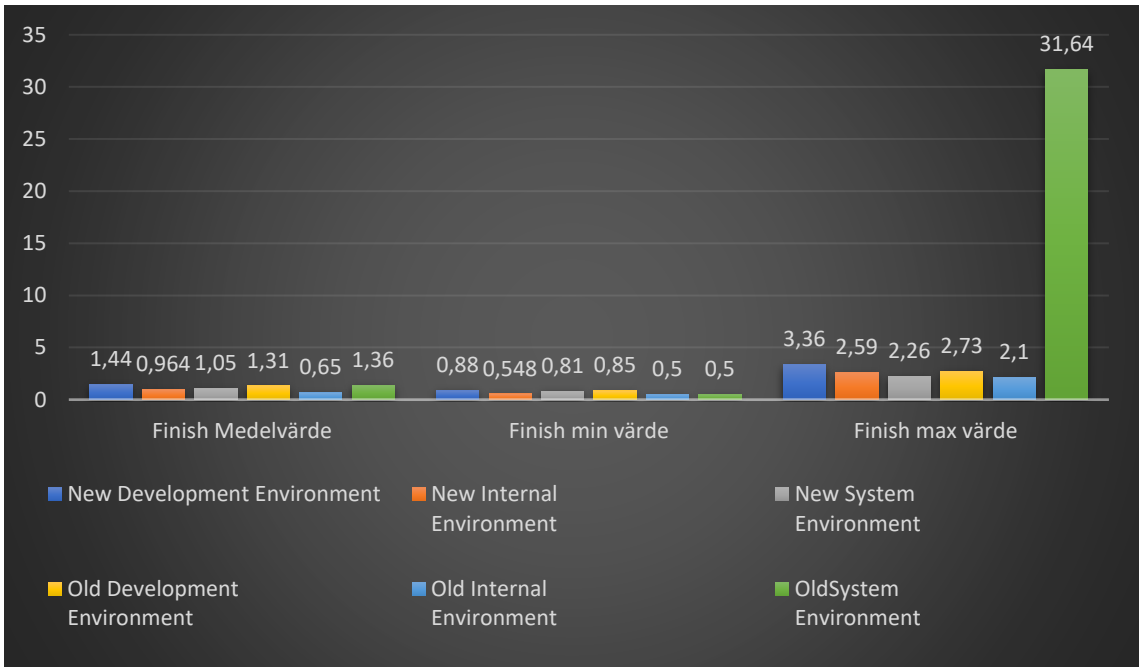
Figur 16 Jämförelse requests



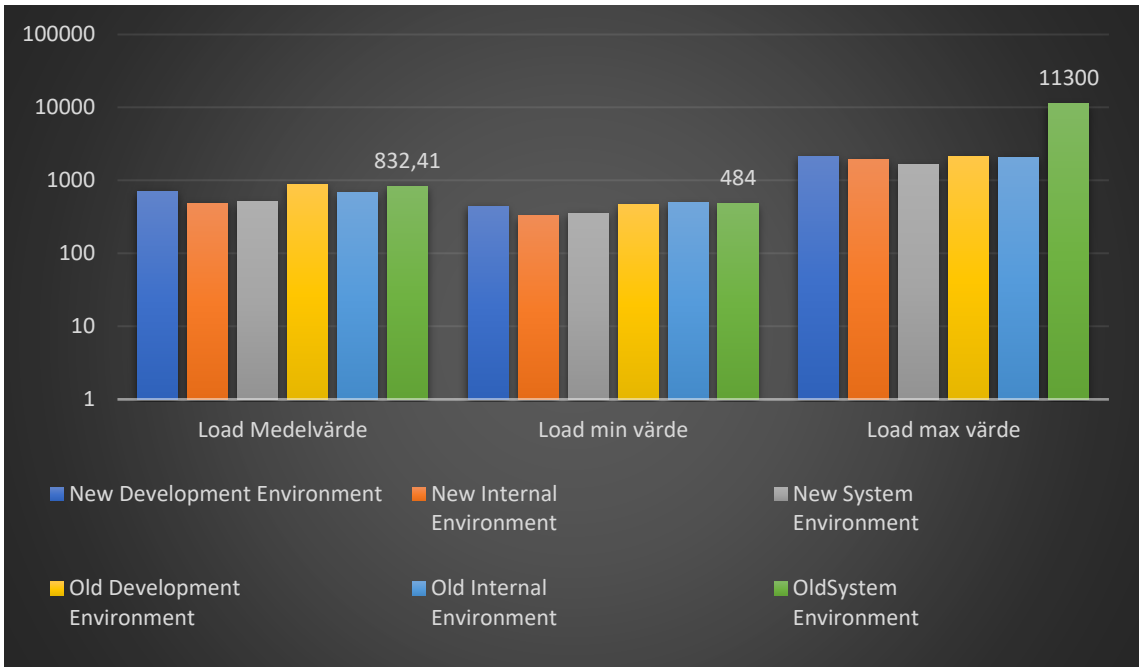
Figur 17 Jämförelse Transferred



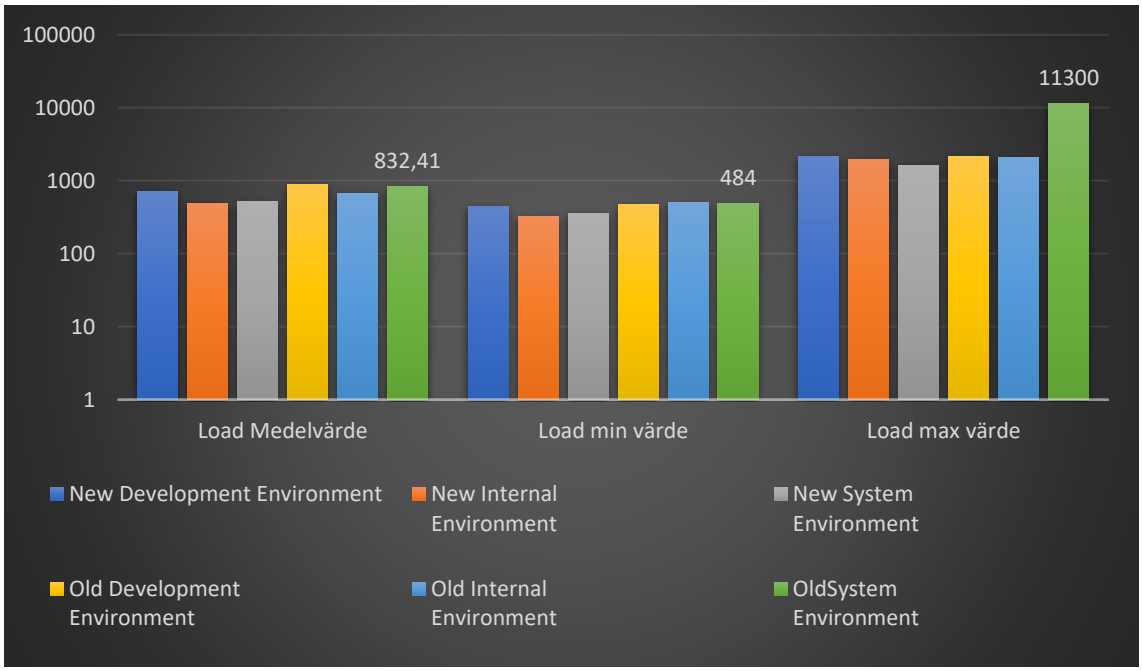
Figur 18 Jämförelse Resource



Figur 19 Jämförelse finish



Figur 20 Jämförelse DOMContentLoaded



Figur 21 Jämförelse Load

10 Källor

AngularJS. (Januari 2022). *End of support*.
https://docs.angularjs.org/misc/version-support-status
Hämtad: 17/01/2023

Atlassian. (u.å.) *Who uses Jira*
https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for#Jira-for-requirements-&-test-case-management
Hämtad: 04 Februari 2023

AutoMapper. (u.å.) *About AutoMapper*.
https://docs.automapper.org/en/stable/
Hämtad: 15 Februari 2023

Bundlephobia. *Bundlesize React-Hook-Forms 3.29.4*.
https://bundlephobia.com/package/react-hook-form@3.29.4
Hämtad: 19 Januari 2023

Bundlephobia, *Bundlesize React-forms 4.0.1*.
https://bundlephobia.com/package/react-form@4.0.1
Hämtad: 19 Januari 2023

Bundlephobia, *Bundlesize Formik 2.2.9*.
https://bundlephobia.com/package/formik@2.2.9
Hämtad: 19 Januari 2023

Burcu Bayhan, B. (25 Januari 2022). *React Form Libraries Comparison: Formik Vs React Hook Form*. Apiumhub
https://apiumhub.com/tech-blog-barcelona/react-form-libraries-comparison-formik-vs-react-hook-form/
Hämtad: 23 Januari 2023

CONFLUENCE
Confluence. (u.å.) *Confluence*. Atlassian
https://www.atlassian.com/software/confluence
Hämtad: 12 Februari 2023

Hygraph Team. (27 September 2022). *React Hook Form – A Complete Guide*. Hygraph.
https://hygraph.com/blog/react-hook-form
Hämtad: 16 Februari 2023

InterviewBit. (29 November 2022). *.NET Core vs .NET Framework*.
https://www.interviewbit.com/blog/net-core-vs-net-framework/
Hämtad: 15 Februari 2023

Kisliak, A. (10 Juli 2019). *Bitbucket*. Polontech.

https://polontech.com/blog/bitbucket-team/#:~:text=The%20main%20benefit%20of%20Bitbucket,changes%20and%20roll-back%20if%20needed.
Hämtad: 04 Februari 2023

MaterialUI. (u.å.) *Material UI – Overview*.
https://mui.com/material-ui/getting-started/overview/
Hämtad: 20 Februari 2023

Pecanac, V. (7 September 2022). *10 Things You Should Avoid in Your ASP.NET Core Controllers*. CodeMaze
https://code-maze.com/ten-things-avoid-aspnetcore-controllers/
Hämtad: 16 Februari 2023

ReactJS. (u.å.). *Introducing Hooks*.
https://reactjs.org/docs/hooks-intro.html
Hämtad: 19 Februari 2023

Santos, J. (u.å.) *The pros and cons of using Jira software*. Project management
https://project-management.com/the-pros-and-cons-of-using-jira-software/
Hämtad: 10 Februari 2023

Stackshare. (20 Januari 2023). *Bitbucket vs Github Enterprise*,
https://stackshare.io/stackups/bitbucket-vs-github-enterprise
Hämtad: 02 Februari 2023

TypeScript, (u.å.). *What is TypeScript*.
https://www.typescriptlang.org
Hämtad: 20 Februari 2023

Walther, S. Mabee, D. Schonning, N. Andersson, R. Addie, S. Pasic, A. Dykstra, T. (07 September 2022). *Validating with a service layer*. Microsoft.
https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/models-data/validating-with-a-service-layer-cs
Hämtad: 10 Februari 2023