Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

VIDAREUTVECKLING AV PROTOTYP

- refaktorering och förbättring av CO_2-portal

Sofia Södergårdh, Christian Syväluoma

<2022:16>

Datum för godkännande: 18.05.2022
Handledare: Björn-Erik Zetterman

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Sofia Södergårdh, Christian Syväluoma
Arbetets namn:	Vidareutveckling av prototyp - refaktorering och förbättring av CO_2-portal
Handledare:	Björn-Erik Zetterman
Uppdragsgivare:	Hållbarhetsgruppen Högskolan på Åland

Abstrakt

Syftet med det här arbetet är att vidareutveckla en prototyp genom bl.a. refaktorering. Vidareutvecklingen är en påbyggnad av den funktionalitet som redan fanns. Genom refaktoreringen vill vi få ner antalet duplicerade funktioner och komponenter för att göra framtida utveckling mer lättnavigerad samt minska risken för fel.

Refaktoreringen av frontend och backend har skett parallellt med hjälp av bland annat postman under processen. Mycket diskussioner har ägt rum under utvecklingen för att tillsammans komma på det bästa sättet att implementera en del av ett krav eller hela kravet.

Alla krav vi hade på vår kravspecifikation har vi lyckats implementera inom tidsramen för arbetet. En del ny funktionalitet har tillkommit och mängden duplicerad kod och antal filer har minskat markant. Det är lättare att förstå projektet och det kommer gå snabbare för de som i framtiden ska utveckla denna applikation att sätta sig in i det.

Nyckelord (sökord)
Refaktorering, Java, Angular, TypeScript, koldioxid

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
2022:16	1458-1531	Svenska	85 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
11.04.2022	13.05.2022	18.05.2022

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Sofia Södergårdh, Christian Syväluoma
Title:	Further development of Prototype - refactoring and improvement of CO_2 Portal
Academic Supervisor:	Björn-Erik Zetterman
Commissioned by:	Sustainability Group Åland University of Applied Sciences

Abstract

The purpose of this work is to further develop a prototype through refactoring among others. The continued development is an extension of the functionality that already existed. This will cut down on the amount of duplicated functions and components so that future development would become more easily navigated and reduce the risk of errors.

The refactoring of the frontend and the backend has been done in parallel with the help of Postman during the process among others. A lot of discussions have taken place during the development to find the best way to implement a specific requirement or a part of it.

All requirements that were in the specification have been successfully implemented within the allotted time. Some new functionality has been added. The amount of duplicated code and amount of files have been reduced significantly. It is easier to understand and will be faster to become acquainted with the project for future developers.

Keywords

Refactorization, Java, Angular, TypeScript, carbon dioxide

Serial number:	ISSN:	Language:	Number of pages:
2022:16	1458-1531	Swedish	85 pages

Handed in:	Date of presentation:	Approved:
11.04.2022	13.05.2022	18.05.2022

INNEHÅLLSFÖRTECKNING

FIGURFÖRTECKNING

1. INLEDNING

1.1 Syfte

Syftet med det här arbetet är att vidareutveckla en prototyp genom bl.a. refaktorering. Vidareutvecklingen är en påbyggnad av den funktionalitet som redan fanns. Genom refaktoreringen vill vi få ner antalet duplicerade funktioner och komponenter för att göra framtida utveckling mer lättnavigerad samt minska risken för fel. Webbplatsen/portalen utvecklas åt hållbarhetsgruppen vid Högskolan på Åland och kommer produktionssättas och tas i bruk direkt efter avslutat arbete. Målen med portalen beskrivet av projektkoordinator Paula Linderbäck (*Utveckling av CO2-portal för Högskolan på Åland*, 2021):

> *Målsättningarna med projektet är att*
> - *skapa en central databas som innehåller data om högskolans alla koldioxidutsläppskällor (skolans lokaler, tjänsteresor, skolfartyget etc.)*
> - *skapa en hemsida som presenterar data om högskolans koldioxidutsläpp i form av olika diagram, så alla på ett lättillgängligt sätt kan följa skolans hållbarhetsutveckling*
> - *så långt som möjligt automatisera datainsamlingen. Berättigade användare av hemsidan ska dock kunna uppdatera databasen via olika formulär.*

Portalen kommer användas av bl.a. Högskolan på Åland för att redovisa koldioxidutsläpp och av all personal på skolan för att redovisa sina tjänsteresor.

1.2 Metod

Vi började med att sammanställa en kravspecifikation avseende vad som behövde förbättras och vilka funktioner som skulle läggas till. Detta gjorde vi i samförstånd med uppdragsgivaren genom möten och diskussioner. Eftersom vi jobbat med att ta fram själva prototypen hade vi en väldigt tydlig bild om vad som kunde förbättras. Dessa krav fick sedan en prioritet med skalan: hög, medium, låg.

Vi har sedan tagit ett krav i taget och påbörjat implementationen. Under tiden implementationen skett har stödord skrivits för dokumentationen. Det har även hjälpt när allt ska sammanställas till detta arbete. När all implementation för ett krav är klart testas allt igenom för att kolla att det fungerar som det är planerat, om det gör det så gör vi kravet

grönt(färdigt) kravspecifikationen. När kravet är grönt tar vi stödorden och skriver det till löpande text direkt. Se fig 1.

Vi började med kraven med högst prioritet och jobbade oss ner till det med lägst. Ofta jobbade vi med två olika krav parallellt då vi är två skribenter/utvecklare. Mycket diskussioner har ägt rum under utvecklingen för att tillsammans komma på det bästa sättet att implementera en del av ett krav eller hela kravet.

Portalen är en *fullstack* applikation där vi som *frontend*-teknik har använt ramverket Angular[1] , *backend* är implementerat i Java spring[2] och som databas har vi använt MariaDB.
Teknikerna är väl kända för oss sedan tidigare men vi har fått fördjupade insikter/kunskaper.

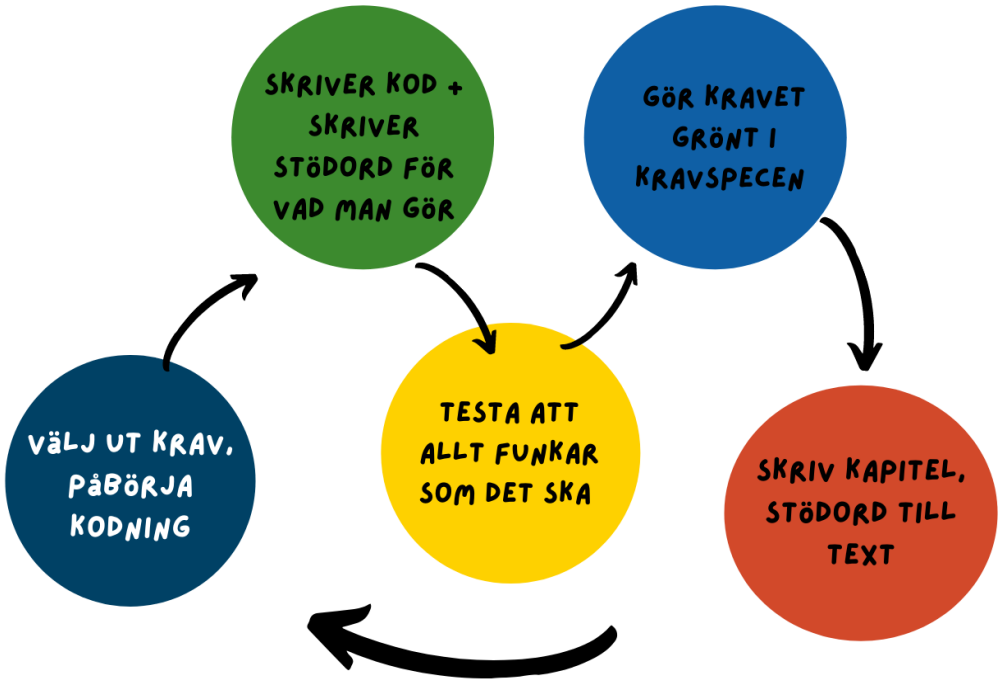


Fig 1, Diagram på arbetsprocessen för arbetet

[1] https://angular.io/
[2] https://spring.io/why-spring

1.3 Avgränsningar

Vi valde redan när vi skrev kravspecifikationen att göra vår första avgränsning och det var att inte undersöka andra ramverk. Orsaken till den initiala avgränsningen är att det är ett otroligt tidskrävande moment. Angular och Java spring var redan valt sedan tidigare och vi var väl insatta i det. Att börja undersöka andra ramverk och utvärdera de olika alternativen för att bestämma vad som är bäst för både frontend och backend ansåg vi att skulle ta väldigt mycket tid. Eftersom arbetet i sig redan var stort så valde vi alltså att jobba med de ramverk som redan fanns och bara förbättra själva koden.

En annan avgränsning uppkom under början av implementationen av krav fem och sex, automatik av förbrukningar, se kapitel 6. Det visade sig att det vi tänkt inte gick att göra p.g.a. tredje parts problem (elbolaget). Problemen bestod av äldre utrustning, komponentbrist som försvårade byte av utrustning och avsaknaden av ett API[3]. Automatiken blev alltså semi-automatisk istället för helt automatiskt, läs mer om detta i kapitel 7.4. Vi skulle haft tid att göra en hel automatisering om det skulle funnits komponenter och förutsättningar att göra det men nu fick vi lösa det på ett annat sätt istället.

1.4 Uppdragsgivare

Det finns ett nationellt program som skapades 2020 och heter: Ansvarsfulla, Hållbara och CO_2-neutrala högskolor. Högskolan på Ålands program styrs av FN:s program för hållbar utveckling, Agenda 2030 och hållbarhets- och utvecklingsagendan 2030 för Åland. Målet med det nationella programmet är att minska fotavtrycket i samhället genom att utveckla verksamheten. Istället vill man lämna ett positivt handavtryck som ska förändra samhället och inspirera andra. Vid Högskolan på Åland finns en grupp, hållbarhetsgruppen, som i samråd med ledningen planerar, koordinerar, kommunicerar och samlar information om skolans arbete för att bli koldioxidneutrala 2030. Denna grupp är uppdragsgivare till detta arbete (*Hållbar utveckling*, 2022).

[3] API = Application Programming Interface

2. BAKGRUND

Vi har jobbat med denna portal tidigare i en annan kurs, programkonstruktion och projekthantering (POP) oktober 2021 - januari 2022. Det fanns då en grund från en tidigare grupp som gått samma kurs under våren 2021. Vi jobbade med andra ord ca. fyra månader med portalen och fortsatte utveckla det som redan fanns tillsammans med samma beställare vi har för detta examensarbete, hållbarhetsgruppen vid Högskolan på Åland.

Det som fanns när vi tog över portalen var en grund för frontend och backend samt ett gränssnitt med en del brister. Vi fortsatte utveckla detta efter samma struktur de hade men vi har vidareutvecklat alla områden. Gränssnittet valde vi att helt göra om för att göra det mer lättnavigerat.

De kategorier som redovisas på portalen är definierade av rådet ARENE. Det är ett råd för de finska yrkeshögskolornas rektorer, där huvudsakliga målet är att främja skolornas gemensamma intressen (*Rådet för yrkeshögskolornas rektorer Arene rf*, 2018). De kategorier som definierats av ARENE är:

1. El
2. Fjärrvärme
3. Vatten
4. Sopor
5. Avlopp
6. Investeringar
7. Resor
8. Michael Sars (unikt för Högskolan på Åland)

För de fem första av dessa kategorier rapporteras datat skilt för högskolans fyra byggnader: Högskolan norra, Högskolan södra, Högskolan västra och Högskolebiblioteket. För investeringar finns det sedan åtta underkategorier den delas in i, olika sorters investeringar helt enkelt. Resor sorteras enligt olika färdmedel. Denna indelning gör att det enkelt går att se vilken byggnad som genererar mest utsläpp och vilken kategori, i vissa fall subtyp, som

genererar mest. För alla dessa kategorier renderas/visas både tabeller och grafer på hemsidan för att visa koldioxidutsläppen.

Det finns tre olika sorters användare som är definierat av tidigare grupp:

- besökare
- inloggad användare
- administratör

En **besökare** kommer bara åt vissa funktioner. Besökaren kan först och främst logga in för att komma åt flera funktioner och byter då till inloggad användare/administratör. För inloggningen används en tredjepart som verifierar inloggning och det är Google då alla vid Högskolan på Åland har en sådan mailadress. Detta fanns implementerat sedan tidigare. Besökare kan använda resekalkylatorn, ett verktyg där man väljer ett färdmedel och antal km och sedan räknar den ut hur mycket co_2 det genererar. Det går också att se en informationssida om själva portalen och om hållbarhetsarbetet på skolan. Sedan kan man som besökare även se CO_2-data för ett år, antingen för alla kategorier sammanslagna eller för en specifik kategori.

Som **inloggad användare** får du tillgång till ytterligare funktioner. Du kan söka data för en kategori för ett eget bestämt intervall. Sedan kan man besöka sin egen profil och ändra uppgifterna om sig själv. Det finns även två blanketter/formulär på sidan som du måste vara inloggad för att kunna fylla i, en för reseförordnande och en för reseräkning. M.h.a. informationen från dessa blanketter räknas CO_2 ut automatiskt och syns sedan i kategorin resor.

Som **administratör** finns det sedan ännu fler funktioner. Det är de mest avancerade funktionerna som alla inte ska kunna göra utan enbart vissa utvalda personer. Det första man kan göra är att rapportera/skriva in ny data för alla kategorier utom resor. Resor räknas automatiskt från de data som användarna ger via blanketterna. En administratör kan i efterhand även redigera/ta bort data ifall något blivit fel. Genom adminfliken kan man även godkänna/ta bort användare genom att ange mailadress. Till sist kan man även ändra faktorn för dessa kategorier, som systemet sedan räknar ut utsläppen med.

I fig 2 nedan kan man se ett *use case diagram* över funktionaliteten för portalen efter POP-kursen och innan examensarbetet påbörjades. På grund av sidans omfattning finns det en användarguide för portalen. Denna guide kan läsas i bilaga 1.

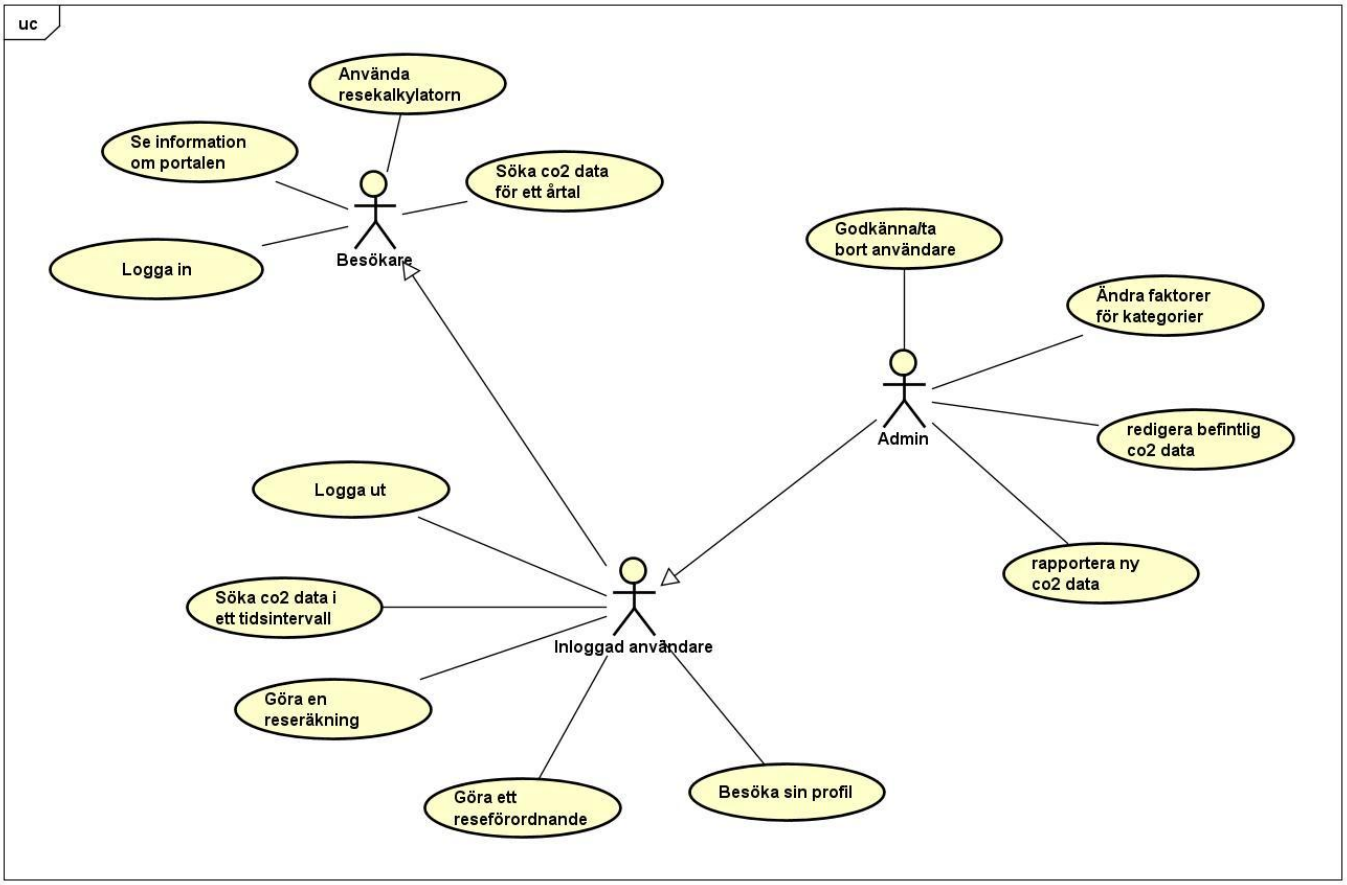


Fig 2, Use case diagram för vad som fanns innan detta arbete påbörjades

3. DEFINITIONER

3.1 TypeScript

TypeScript (TS) är ett programmeringsspråk gjort av Microsoft som släpptes 2012. Skaparen av språket heter Anders Hejlsberg (Wikipedia contributors, 2022c). Språket har vuxit med åren, har stort stöd från en mängd IDEs[4] och räknas idag till de mest älskade språken just nu (Fain & Moiseev, 2020). Den senaste versionen, 4.6.4, släpptes i april 2022. Om man enkelt ska sammanfatta vad det är så är det en uppgradering av JavaScript (JS) som var efterfrågad. JS gick inte direkt att ersätta med tanke på hur stort språket är så då valde man att utveckla och utgå från det som fanns och sätta till det man tyckte saknades.

JavaScript är världens största programmeringsspråk och är enkelt att lära sig (*JavaScript Tutorial*, n.d.). Det är ett språk som används för webbutveckling och jobbar tillsammans med HTML[5] och i de allra flesta fallen även CSS[6]. JS är gjort av Brendan Eich, släppt 1995, och hette från början Mocha men ändrades under tidens gång och kallades till sist Javascript (Wikipedia contributors, n.d.-a). Koden sköter alltså all funktionalitet för en webbplats, exempelvis knapptryckningar och sökningar. Typescript är då en utveckling av hur koden kan skrivas och hanteras.

Typescript används i både vid utveckling av backend[7] och frontend[8]. De största webramverken idag är React[9], Vue[10] samt Angular[11]. Det vi använder för utveckling av frontend, Angular, har stöd för typescript och är till och med det primära språket (*Angular*, n.d.-a).

[4] IDE = integrerad utvecklingsmiljö
[5] HTML = Hypertext Markup Language, märkspråk för hypertext
[6] CSS = Cascading Style Sheet, beskriver utseendet för en webbplats
[7] Backend = en del av ett system som användaren inte har direkt tillgång till
[8] Frontend = grafiskt användargränssnitt för en webbplats
[9] https://reactjs.org/
[10] https://vuejs.org/
[11] https://angular.io/

Allt som är skrivet i JS går alltså att köra som TS-filer också. Faktum är att typescriptfiler, med ändelse .ts, transpileras till javascriptfiler och körs som det också. En skillnad mellan språken är alltså att TS måste transpileras till JS för att kunna köras medan det andra startas direkt.

I figur 3 ses hur det går till när man kör en typescriptapplikation då man börjar med a.ts, b.ts och c.ts som transpileras till a.js, b.js och c.js, för att sedan paketeras ihop till main.js.

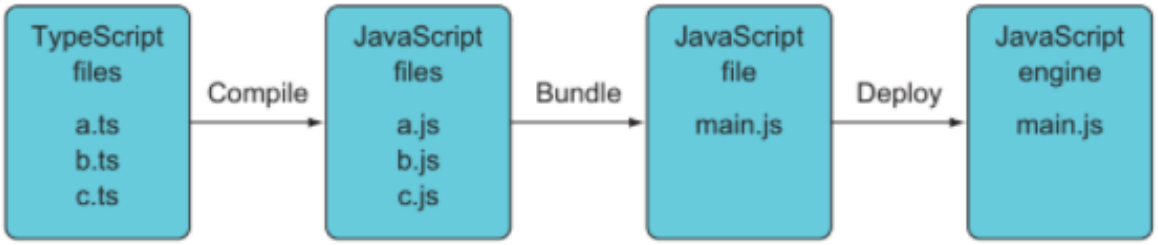


Fig 3, Beskrivning hur en Typescriptapplikation byggs och körs (Fain & Moiseev, 2020)

En av de stora förbättringarna från JS som var efterfrågat var "static checking", någonting som TS stödjer. Det innebär att du inte behöver köra programmet för att få errors. De upptäcks i transpileringen, vilket skiljer sig från JS där man måste köra programmet för att få felen. Finns det något fel med koden, upptäcks det direkt i transpileringen. Felet överförs alltså inte till applikationen så att en användare/testare skall upptäcka felet. Denna uppgradering gör arbetsprocessen smidigare då felen ofta hittas mycket snabbare. Utvecklare kan välja att inte använda denna funktion om man önskar.

Sedan finns även "static type checking" i TS. Det handlar som namnet säger om variabler och vilka typer de har. En annan skillnad från JS är nämligen de utökade variabeltyperna. Alla variabler måste deklareras med en typ som man sedan måste hålla sig till. Deklarerar man exempelvis någonting till en textsträng så måste man hålla det. Man kan inte senare sätta den till en siffra för få kommer du få ett fel. Detta är alltså "static type checking". Den kollar att man hela tiden håller dig till typen man satte för variabeln (*Documentation - The Basics*, n.d.).

En ytterligare sak som skiljer sig är att TS liknar objektorienterad programmering och har stöd för det. Det stödjer exempelvis klasser, interface och enumerationer. Men eftersom det bygger på JS så är det inte helt klassisk objektorienterad programmering där allting är helt bundet till klasser utan skiljer sig lite. Men det stödjer ändå en del av de vanligaste mönstren såsom arv, statiska metoder och implementation av gränssnitt.

3.2 Angular

Angular är ett av de största webbramverken idag och används av cirka 1,7 miljoner utvecklare (*Angular*, n.d.-b). Det släpptes 2016 och utvecklas av Google och deras utvecklare. Ramverket är som tidigare nämnt byggt på ett annat språk, TypeScript, som är beskrivet i kapitel 3.1.

3.2.1 Moduler

Angular har en del grundkomponenter som är huvudidén bakom ramverket. En av de viktigaste är moduler. Utan dem kommer inte applikationen att starta över huvud taget (Bors, 2018). Det går att ha flera moduler men alla måste ha en som beskriver själva applikationen och dess delar, en så kallad rotmodul (Bors, 2018). Denna definierar ett set av komponenter, som beskrivs mer ingående i kommande kapitel.

Denna del använder sig av annotationen *@NgModule( )* när de deklareras. Innanför parenteserna sätts olika objekt till för att beskriva modulen, totalt finns fem olika (Bors, 2018):

1. Deklarationer, där sätter vi till alla komponenter som vi själva skapat och vad som är våra olika vyer.
2. Exporter, där deklareras sådant som ska användas i andra moduler.
3. Importer, det deklareras utomstående moduler och klasser som sedan används i våra egna moduler.
4. Leverantörer, de services/leverantörer som deklareras här kommer att bli tillgängliga genom hela applikationen.
5. Bootstrap, där säger man vad som ska vara huvudvy/rotkomponent och då vilken sida som ska synas när man startar applikationen.

3.2.2 Komponenter

Sedan har vi en annan viktig del, komponenter, som deklareras med annotationen *@Component( )*. En komponent som skapas innehåller fyra filer:

- HTML fil
- SCSS/CSS fil för styling
- TypeScript fil för all logik
- spec fil för tester.

Tillsammans skapar dessa fyra filer en vy. I exemplet här under i fig 4, blir dessa delar en adminsida (*Angular*, n.d.-b). Alla komponenter deklareras alltså som tidigare nämnt enligt fig 5. Det är även här man kopplar ihop själva logiken i Typescriptfilen med HTML-strukturen, templateUrl och även med stylingen styleUrls. I detta projekt använder vi oss av komponenter för alla våra vyer. Det är ett väldigt enkelt sätt att hålla reda på allt som tillhör en vy då det blir strukturerat och överskodligtt.

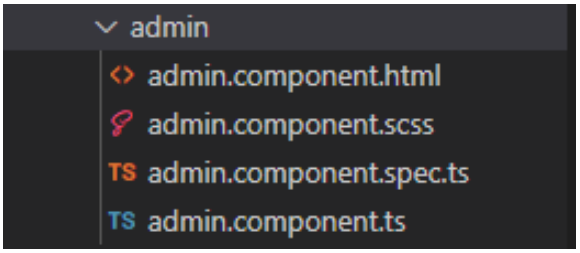


Fig 4, Filstruktur för en komponent

```
@Component({
  selector: 'app-admin',
  templateUrl: './admin.component.html',
  styleUrls: ['./admin.component.scss']
})
export class AdminComponent implements OnInit {
```

Fig 5, Deklaration och sammankoppling för komponent i Typescript filen

3.2.3 Angular router

För att navigera mellan dessa olika vyer så behövs någon form av routing. Det har vi i vårt projekt löst med Angular router. Detta läggs lättast till när man skapar ett projekt. För att

sedan lägga till olika länkar måste de först importeras i rotmodulen och sedan kan man skapa en väg till komponenten. Ett exempel på detta finns i fig 6.

Man skriver in namnet på sin länk och vilken komponent den hör till. Sedan finns även möjligheten som vi gjort och lägga på andra saker. T.ex. finns möjligheten att skicka med data till komponenten som den sedan kan använda. I fig 6 skickar vi med vad det är för kategori. Sedan har vi även lagt på en boolesk parameter, "canActivate". Det gör att vi kan dölja vissa saker för "vanliga" användare och visas då bara för de som är admin (*Angular*, n.d.-c).

```
{ path: 'michaelsarsnew', component: ReportReportComponent, data: {type:"michaelSars"}, canActivate: [isAdmin] },
```

Fig 6, Definition av en sökväg

3.2.4 Services

En annan del som är väldigt användbar är services. Detta är en klass som förmodligen kunde få plats i en komponent men som ändå flyttas ut till en egen del för att få en tydligare struktur och bättre möjligheter att återanvända koden. Detta har vi använt oss av. Dessa delar är singletons, det finns alltså bara en instans av varje service. De är även *stateless* och kan anropas från vilken komponent som helst bara de importeras. Kod i dessa filer är oftast anrop till *backend* när något behöver hämtas från databasen eller liknande (Bors, 2018).

3.2.5 Angular CLI

Angular har även en annan väldigt användbar funktion, Angular CLI. CLI står för *command line interface* och används bl.a. för att skapa komponenter, services, interface och underhålla applikationer direkt från kommandotolken. För att skapa en komponent kan man antingen skapa mapp, lägga till filer, koppla ihop dessa filer och sedan importera och lägga till dem i rootmodulen. Eller så använder man CLI och skriver då: *ng generate component my-component*. Den skapar då allt man annars måste göra själv helt automatiskt. Det är ett väldigt praktiskt verktyg att använda (*Angular*, n.d.-d).

3.2.6 Dependency injection

Dependency injection är ett stort ämne i Angular. Detta är ett designmönster som gör att vi kan dela upp kod, implementeringar tas bort från objekt och plockas ut till en egen del som lättare går att testa och återanvända. En eller flera services injiceras alltså i ett beroende objekt m.h.a. konstruktorn. Servicen använder annotationen *@Injectable* för att markera att den kan användas för *dependency injection*. Värdet på *providedIn* talar om för applikationen i vilken eller vilka delar den är tillgänglig i (*Angular*, n.d.-e).

I fig 7 och 8 ses ett exempel från vårt arbete. Vi skapar en service med annotationen och sätter värdet på *providedIn* till "root" vilket betyder att den blir tillgänglig i hela applikationen. Denna service injicerar vi sedan i en komponentkonstruktör som en privat inparameter med typen *ServiceServiceService*. Det går även att göra detta i andra services.

```
@Injectable({
  providedIn: 'root'
})
export class ServiceServiceService{}
```

Fig 7, Deklaration av en service

```
export class GraphGraphComponent implements OnInit {
     constructor(private service: ServiceServiceService,) {}
}
```

Fig 8, Servicen injiceras i komponenten

3.2.7 Formulär

De flesta applikationer måste hantera användarinput på något sätt. Exempelvis för att logga in, uppdatera profil eller fylla i en blankett. För detta används formulär som tar in det användaren skrivit in, validerar det, skapar en formulärmodell och en datamodell som uppdateras för att sedan kunna följa ändringarna. Angular har två sätt att hantera denna input genom formulär: reaktiv eller malldriven (*Angular*, n.d.-f).

Malldrivna formulär passar sig för formulär med låga krav och enkel logik. Detta kan exempelvis vara registrering för nyhetsbrev. Den är inte lika skalbar men är enkelt att lägga

till och all data hanteras direkt i mallen. Är formuläret däremot en viktig del av ens applikation passar reaktiva bättre. Dessa är skalbara och enklare att testa. Det ger en dessutom direkt åtkomst till formulärobjektmodellen. Reaktiva formulär skapas i en komponent medan malldrivna skapas av direktiv.

Båda formulärtyper bygger på fyra basklasser.

- FormControl
- FormGroup
- FormArray
- ControlValueAccessor

För att lättare skapa formulär med dessa basklasser har vi använt oss av ett API, *formbuilder*. Det är en sorts reaktivt formulär och gör att komplexa formulär kan skrivas lättare med mindre kod med en del genvägar för att skapa just dessa basklasser (*Angular*, n.d.-g).

I fig 9 ses ett av våra formulär. En *formbuilder.group* skapas och i denna sätts en nyckel in (ett fält i formuläret). Denna får sedan ett default värde (en tom sträng i vårt fall) och sedan sätts valideringar för nyckeln/fältet. På nyckeln *otherCost* syns även hur *formArray* används med builder.

```
fourthFormGroup = this._formBuilder.group({
    travelSpecId: ['', [Validators.required]],
    ownCarKm: ['', [Validators.min(0)]],
    ownCarFactor: ['', [Validators.min(0)]],
    ownCarReinbursement: ['', [Validators.min(0)]],
    dailyAllowances: ['', [Validators.min(0)]],
    dailyAllowancesSum: ['', [Validators.min(0)]],
    otherCosts: this._formBuilder.array([this.newOtherCost()]),
    advance: ['', [Validators.min(0)]],
    files: [],
  });

  newOtherCost(): FormGroup {
    return this._formBuilder.group({
      description: ['', [Validators.required]],
      vat: [this.vats[2].name, [Validators.required]],
      cost: ['', [Validators.min(0)]]
    })
  }
```

Fig 9, Formulär som skapats med formbuilder

3.2.8 Internationalisering

Internationalisering handlar enkelt sagt om att en applikation ska finnas tillgänglig på olika språk, s.k. *locales*. Detta kallas ibland också i18n, processen att designa och förbereda ditt projekt på olika språk (*Angular*, n.d.-h). Det är inte bara texten som skiljer mellan olika språk utan också datum kan skilja mellan olika länder, allt sådant måste man tänka på. Vilka språk man vill använda är upp till en själv. Dock måste man göra översättningen själv.

Som utvecklare behöver man göra några moment för att detta ska fungera, närmast märka upp alla ställen där det finns text som ska översättas. Angular kommer sedan leta efter dessa ställen och byta ut text, datum m.m. utifrån vilken *locale* (språkinställning i webbläsaren) som användaren valt. Olika sätt för att göra dessa översättningar med olika språkfiler nämns närmare i kapitle 7.5 (*Angular*, n.d.-h).

3.3 Ramverket Java Spring

Ramverket Spring är en tillämpningsram och container för invertering av kontroll (IoC) för plattformen Java. Spring är det mest populära ramverket för utveckling av Enterprise Java. Ramverket är *open source* och skapades av Rod Johnson och släpptes först under licensen Apache 2.0 i juni 2003. Första produktionsversionen släpptes den 24 mars 2004 (Wikipedia contributors, 2022b).

Kärnfunktionerna i Spring kan användas i utvecklingen av vilket Javaprogram som helst, men det finns tillägg för att bygga webbapplikationer ovanpå plattformen Java EE (Enterprise Edition). Spring förespråkar god programmeringsmetodik genom att möjliggöra en POJO-baserad (Plain Old Java Object) programmeringsmodell (*Spring Framework - Overview*, n.d.).

3.3.1 Dependency Injection

Den teknologi som Spring är mest känd för är beroendeinjektion[12], en sorts invertering av kontroll (IoC), likt Angulars implementation beskrivet i kapitel 3.2.6. IoC är ett generellt koncept och beroendeinjektion är ett konkret exempel. När man skriver ett komplext Javaprogram behöver man hålla klasserna så oberoende av varandra som möjligt för att maximera möjligheten av återanvändning och för att kunna testa dem separat i enhetstester (Crusoveanu, 2016). Enhetstester är automatiska tester som skrivs och körs av utvecklare för att säkerställa att en sektion av ett program beter sig som förväntat (Wikipedia contributors, 2022d). Beroendeinjektion är limmet som håller ihop dessa klasser på samma gång som de hålls oberoende (Crusoveanu, 2016).

Det som beroendeinjektion gör är att injicera objekt in i andra objekt via en montör (assembler) istället för av objekten själva. I exemplet nedan (se fig 10) ses hur objektberoende sköts i traditionell programmering där man behöver instansiera en implementation av gränssnittet Item inne i själva Storeklassen (Crusoveanu, 2016):

[12] Dependency Injection

```
public class Store {
	private Item item;
	public Store() {
		item = new ItemImpl1();
	}
}
```
Fig 10, Implementering i traditionell programmering

Men genom att använda beroendeinjektion (se fig 11) kan man skriva om exemplet utan att specificera någon implementation för Item:

```
public class Store {
	private Item item;
	public Store(Item item) {
		this.item = item;
	}
}
```
Fig 11, Implementering med dependency injection

3.3.2 Spring Boot

Spring Boot är en förlängning av ramverket Spring, som är en lösning för att snabbt skapa självstående applikationer på produktionsnivå som direkt är redo att köras. Det är färdigt konfigurerat med utvecklarna av Spring Boots egensinniga syn på den bästa uppsättningen av konfigurationer och tredje parts bibliotek så att man kan börja med minsta krångel.

Eftersom Spring Boot kommer med den inbyggda förmågan autokonfiguration, kommer både det underliggande ramverket och tredjepartspaket att automatiskt konfigureras baserat på *best practices* och ens egna inställningar (IBM Cloud Education, n.d.).

Vill man ha en inbäddad webbserver tillåter Spring Boot en att enkelt bädda in Tomcat[13], Jetty[14] eller Undertow[15] eller någon annan webbserver. Då Tomcat bäddas in som standard

[13] https://tomcat.apache.org/
[14] https://www.eclipse.org/jetty/
[15] https://undertow.io/

kan man med en enkel konfiguration exkludera Tomcat och välja något annat (se fig 12) (*76. Embedded Web Servers*, n.d.).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
      <!-- Exclude the Tomcat dependency -->
      <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
    </exclusions>
</dependency>
<!-- Use Jetty instead -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Fig 12, XML-konfiguration för Jetty som webbserver

3.4 Generics

Generics är en metod att påtvinga typer för objekt och metoder genom att ge fel om koden bryter mot typsäkerheten. Det är mycket lättare att åtgärda fel och problem vid kompileringen jämfört med under körning av applikationen. Användningen av Generics gör att man inte behöver explicit omvandla objekt till den datatyp man vill använda.

I fig 13 kan man se en lista med rå typ där man måste omvandla objekt till den typ man vill använda, i det här fallet är det en sträng (String) som används (*Why Use Generics?*, n.d.).

```
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0);
```
Fig 13, Explicit omvandling

Samma kod skrivet med Generics gör att man inte behöver omvandla:

```
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0);
```
Fig 14, Ingen explicit omvandling med Generics

Man kan skapa subtyper av Generics genom att utöka (extends) eller implementera (implements) den generiska typen. Relationen mellan typparametrarna av en klass eller ett interface och ett annat beror på vilket av *extends* eller *implements* som används (*Generics, Inheritance, and Subtypes*, n.d.).

I Generics finns *wildcard* betecknat med ett frågetecken (?) som representerar en okänd typ. Detta *wildcard* kan användas i en mängd olika situationer, objekttyper, parameter, lokala variabler eller som retur typ från en metod. *Wildcards* kan också utöka andra klasser för att skapa en relation.

Givet exemplet med de två icke generiska klasserna i fig 15 kan man rimligtvis skapa objekt enligt fig 16.

```
class A {/* ... */}
class B extends A {/* ... */}
```
Fig 15, Utökning av klass.

```
B b = new B();
A a = b;
```
Fig 16, Instansiering av objekt.

Exemplet visar att arv av vanliga klasser följer regeln om subtypning: Klass B är en subtyp av klass A om B utökar A. Denna regel gäller inte för generiska typer (se fig 17).

```
List<B> lb = new ArrayList<>();
List<A> la = lb; // ger fel vid kompilering
```

Fig 17, Kompileringsfel

För att koden skall kunna komma åt de metoder som finns i objekttypen Number via elementen i listan av Integers, behöver man skapa en relation mellan Integer och Number med hjälp av ett övre bundet *wildcard* (*upper bounded wildcard*), se fig 18 för exempel på detta.

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number> numList = intList;
```

Fig 18, Upper bounded wildcard.

Koden i fig 18 fungerar för att List<? extends Integer> är en subtyp av List<? extends Number> (*Wildcards and Subtyping*, n.d.).

3.5 Refaktorering

Refaktorering eller omstrukturering är en process där man ändrar på en mjukvara på ett sådant sätt att det externa beteendet inte förändras, utan istället förbättrar den interna strukturen. Det är ett bra sätt att städa upp koden för att minimera risken med att introducera buggar, det betyder att när man refaktorerar så förbättrar man designen på koden efter att den har skrivits.

"Att förbättra designen efter att den har skrivits" låter kanske lite udda. I skrivande stund är den generella förståelsen av mjukvaruutveckling att man designar systemet först och efter det kodar man. Det vill säga en bra design kommer i första hand, kodning kommer i andra hand. Över tid blir koden ofta modifierad och integriteten på systemet och dess struktur börjar sakta tyna bort. Nivån på koden sjunker alltmer från en högre nivå till hacking (Fowler, 2018).

Refaktorering är det rakt motsatta till just den förståelsen, man tar dålig kod och arbetar om den till en väldesignad kod. Varje steg är inkrementellt och simplistiskt. Orsaken till varför man vill refaktorera kod är till exempel att man upptäcker en så kallad “code smell” det vill säga en metod som kanske är för lång, en duplicering av en annan metod eller att metoden gör för mycket. När problemet väl är upptäckt, kan det åtgärdas med just refaktorering av källkoden eller genom att förvandla den till en ny form som beter sig på samma sätt som tidigare men nu i en bättre design (Fowler, 2018).

Man flyttar exempelvis ett fält från en klass till en annan eller extraherar någon kod ur ett större kodblock till sin egen metod. För de metoder som är för långa kan man plocka ut en eller flera delar av koden till sina egna metoder, för dupliceringar kan man ta bort och ersätta dem med en delad eller generaliserad metod. Och det är alla dessa små ändringar som förbättrar designen avsevärt. Det är den raka motsatsen till mjukvaru degradation (*software decay*) (Fowler, 2018).

Ett exempel på hur en liten refaktorering kan se ut kan ses i fig 19 nedan, där har vi en metod som tar in två nummer och jämför dem och returnerar true om det första är större än det andra. I fig 20 nedan kan man se hur metoden blev efter refaktoreringen, beteendet är fortfarande likadant externt men har nu en mer effektiv intern struktur (Brian, 2022).

```
public boolean max(int a, int b) {
    if(a > b) {
        return true;
    } else if (a == b) {
        return false;
    } else {
        return false;
    }
}
```

Fig 19, En jämförelsemetod

```java
public boolean max(int a, int b) {
    return a > b;
}
```
Fig 20, Optimerad jämförelsemetod

Dessa två exempel visar på hur refaktorering kan göra mjukvaran lättare förstådd, inte bara för en själv utan även för andra som eventuellt kommer att använda och ändra på samma kod. När man först skriver sitt program försöker man i första hand att få det att fungera, man har ingen tanke på framtida utvecklare. Refaktorering hjälper en att göra sin kod mer läslig då ens kod fungerar men inte är strukturerad på ett idealiskt sätt. Att programmera på det här sättet får en att skriva koden tydligt på ett sådant sätt att man kan förstå vad koden har för syfte.

En annan positiv aspekt från refaktorering är att det hjälper en att utveckla kod snabbare. Detta låter kanske lite motsägelsefullt men om man tittar på hur det går när man inte har en bra design är att det i början går fort att skriva kod, men allt eftersom behöver spendera mer och mer tid på att hitta och fixa buggar istället för att lägga till nya funktioner. Förändringar tar längre tid att implementera då man försöker förstå systemet och hitta kod dupliceringar. Varje ny funktion tar längre tid att implementera då man ofta måste skriva om tidigare kod som redan har blivit omskriven sedan originalkoden skrevs. Har man däremot en bra design redan från början och regelbundet refaktorerar koden, förhindrar det systemet från att degraderas och man hålla en hög utvecklingsfart (Fowler, 2018).

För att verkligen försäkra sig om att den kod man refaktorerar fortfarande har samma externa beteende som förut bör man ha välskrivna automatiska enhetstester. Dessa tester ger ökad säkerhet även vid större refaktoreringar där de utförs mot en enskild aspekt av systemet. När man väl har enhetstesterna på plats kan man sedan iterativt refaktorera koden i små inkrement medans testerna ser till att systemet fungerar som förväntat. Skulle ett test fallera, gör man om den lilla förändringen på nya sätt tills testet går igenom. Det är samtidigt viktigt att testerna går fort att genomföra så att programmeraren inte behöver spendera onödigt mycket tid i väntan på att testerna är klara (Wikipedia contributors, 2022a).

Det man inte får glömma är att backa tillbaka om man har kört fast. Det är mycket lätt hänt att man gör flera refaktoreringar i rad utan att testa koden emellan. Man är självsäker och har övat på det här förut. Men sen plötsligt fallerar ett test och man vet inte vilken ändring som var orsaken. Att inte stanna upp här och backa tillbaka är en stor risk, det är omöjligt att veta hur lång tid det skulle ta att fixa problemet med debuggning. Så det man istället skall göra är just att backa tillbaka till senaste fungerande konfiguration och sedan tillämpa sina ändringar steg för steg med tester emellan (Fowler, 2018).

3.6 REST

REST är en förkortning av *representational state transfer*, introducerat av Roy Fielding 2000. Detta är en arkitekturstil som har vägledande principer och begränsningar. Följer ett webb API detta så är det ett REST API eller RESTful som det också kan beskrivas som (*What Is REST*, 2018).

Med klient i detta kapitel menas alla former av enheter som använder internet.

REST definierar sex arkitektoniska begränsningar, varav en är frivillig, som måste tillämpas för att ett API ska kunna klassas som just RESTful (*REST Architectural Constraints*, 2018):

1. **Enhetligt gränssnitt** = ens API ska skrivas på samma sätt i samma format, så om någon förstått hur ett av ens API fungerar så ska den personen helt enkelt kunna följa de andra också.
2. **Klient-server** = Klient och server ska utvecklas separat. Klienten ska enbart veta API:ernas endpoints, i övrigt ska de inte ha någon koppling.
3. **Stateless** = Servern sparar ingen information från klienten mellan förfrågningar, klienten ansvarar för att skicka med all information som servern behöver.
4. **Cacheable** = Alla resurser som kan vara cachebara skall vara det, för att minska antalet serveranrop och förbättra prestandan.
5. **Layered system** = Serverdelen är uppdelad på flera servrar som sköter olika saker. Klienten vet inte vilken server i vilket lager den använder för sin förfrågan utan det sker automatiskt.

6. **Code on demand** (inte obligatoriskt) = Ibland behöver servern kunna skicka körbar kod istället för XML eller JSON, koden ska stödja klienten vid tex rendering av komponenter.

När en klient gör en förfrågan till servern kan HTTP metoder användas. Det är ett meddelande som innehåller data och en URL (endpoints), så att servern sedan vet vad klienten frågar efter. Detta data är oftast skapat i antingen XML eller JSON. Det finns flera metoder, vilken som används beror på om man vill skapa, ta bort, uppdatera eller hämta data. Detta finns beskrivet i fig 21. De mest använda metoderna är (*How to Design a REST API*, 2018):

- GET = Hämta data
- POST = Skapa ny data
- PUT = Uppdatera befintlig data
- DELETE = Ta bort data

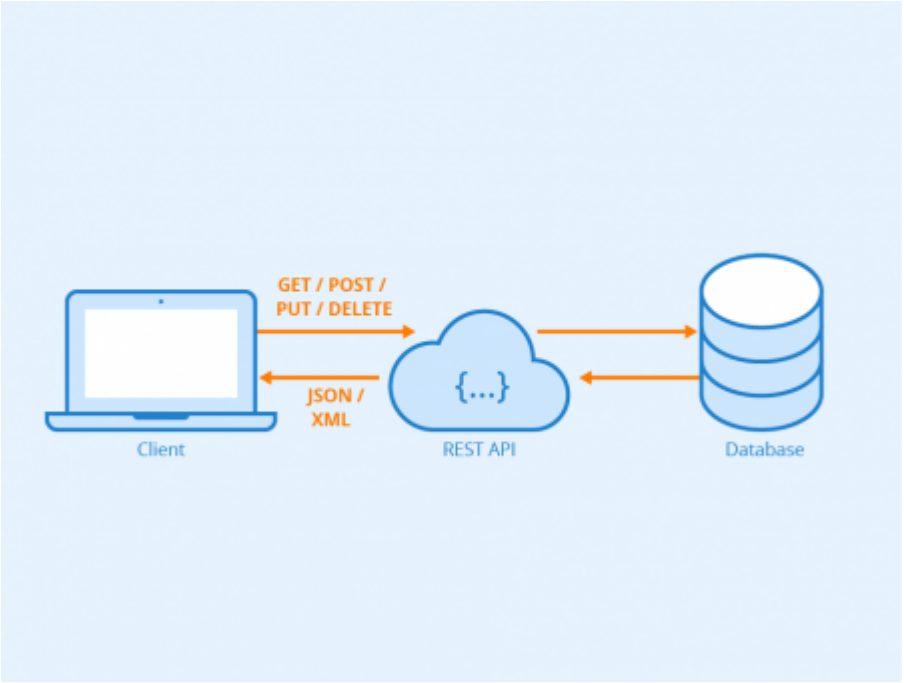


Fig 21, Beskrivning av hur en client/server applikation fungerar (REST API, n.d.)

För att klienten sedan ska veta resultatet av begäran så används statuskoder, i vissa fall innehåller responsen även data. HTTP har ett set standardkoder som är indelade i fem olika kategorier. Det är genom första siffran i koden man ser detta (Wikipedia contributors, n.d.-b):

- 1xx = Information, förfrågan var korrekt och togs emot men kunde inte slutföras.
- 2xx = Lyckad förfrågan, förfrågan utfördes på ett korrekt sätt.
- 3xx = Vidarekoppling, någon ytterligare handling måste göras för att förfrågan ska kunna slutföras.
- 4xx = Klientfel, förfrågan har något syntaxfel.
- 5xx = Serverfel, servern lyckades inte att slutföra en giltig förfrågan.

De vanligaste statuskoderna från dessa kategorier, som sedan används och ingår i REST:s standard, är följande:

- 200 = OK
- 201 = CREATED
- 400 = BAD REQUEST
- 404 = NOT FOUND
- 500 = INTERNAL SERVER ERROR

Vår personliga favorit kod är däremot 418, “I’m a teapot”. Koden var till en början ett aprilskämt 1998 som kortfattat säger att servern inte kan laga kaffe då servern är en tekanna (*418 I’m a Teapot*, n.d.). En rolig kod som vi använt på skoj medan vi testat applikationen.

4. VERKTYG

4.1 Babel Edit

Babel Edit är ett verktyg för översättningar som vi av en slump hittade när vi skulle börja med våra översättningar. Denna tjänst innehåller dessutom stöd för 42 olika språk. Vår språkfil på svenska är runt 860 rader som skulle översättas till två andra språk. När vi insåg hur länge det kommer ta så försökte vi hitta något som kunde hjälpa oss och hittade då detta verktyg och vill därför lyfta fram det. Nedan finns en skärmdump från programmet, fig 22.

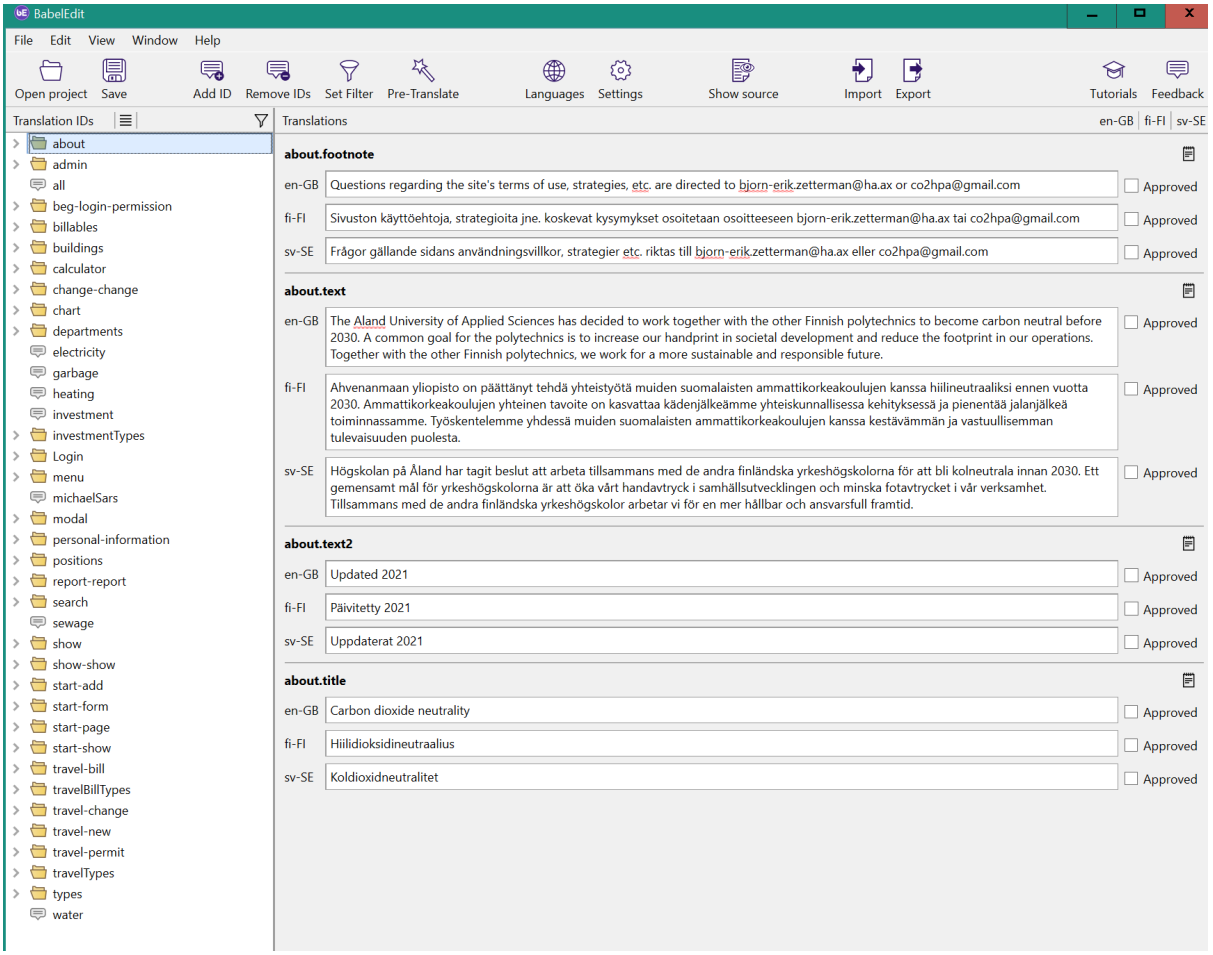


Fig 22, Screenshot från Babel Edit

När man laddat ner och startar Babel väljer man först ett projekt. Vi valde såklart Angular. Där importerade vi sedan vår svenska JSON-fil och genom en *dropdown* valde vi svenska som språk för filen. Sedan adderade vi de två andra språken vi ville ha, engelska och finska.

Babel höll kvar samma struktur vi hade på vår svenska JSON-fil och vi kunde då enkelt hitta bland våra fält. Varje fält hade då tre rader, de olika språken, där svenska redan var ifyllt. Vi kunde då antingen skriva översättningen själva eller välja “pre-translate” och Google Translate gjorde det åt oss.

Vi valde att Google Translate fick översätta alla fält åt oss. Sedan kunde vi gå igenom allt och se till att det lät okej. Vi ändrade något ställe men det mesta såg okej ut. Vi hade väldigt få längre stycken så grammatiken hann aldrig bli fel som det ofta blir vid användande av dylika tjänster. Detta gjorde att vi kunde översätta en hel webbplats på 15 min. Sedan kunde vi ladda ner den engelska och finska JSON-filen och lyfta in dem i projektet och det var klart. Se figur 23 för exempel på hur översättningen kan se ut.

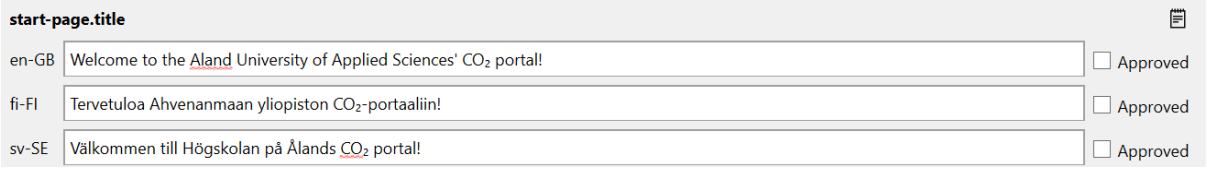


Fig 23, Screenshot på hur ett ID ser ut med de konfigurerade språken

Utöver detta kan man även koppla hela sitt projekt till Babel och den kollar då om det finns översättningar/id som inte används längre. Den innehåller även en stavningskontroll för felstavningar på alla 42 språk. Man kan använda sig av filter för att söka olika ID. Det innehåller även funktioner för att kommentera olika översättningar och det har en godkänn flagga man kan kryssa i när översättningen är godkänd. Man kan sedan filtrera på vad som är klart och godkänt och vad som ännu behöver kollas. Man kan även se statistik på detta (*BabelEdit User Interface*, n.d.).

4.2 Postman

Ett verktyg som har underlättat utvecklingen och felsökningen av vår backend är Postman. Med det här verktyget kan man enkelt skapa anrop till sin server eller någon annan tjänst, istället för att direkt bygga frontendfunktionerna som ska anropa servern.

För om man testar backendlogiken via frontend och någonting är fel eller att man får ett felaktigt eller oväntat svar, blir det svårt att lista ut om felet är i frontend eller i backend. Men med Postman kan man lämna bort frontenddelen och bara fokusera på att logiken i backend fungerar som den skall.

För att göra detta skapar man en ny *request* i Postman och väljer vilken typ av anrop som önskas (GET, POST, DELTE o.s.v.), adressen till servern, eventuella parametrar och en *Authorization header* om det är ett anrop som kräver behörighet samt en *body* om det rör sig om ett anrop som ska skapa eller uppdatera en resurs (POST, PUT). Svaret man får i retur (se fig 24 för exempel med parametrar) kommer med en statuskod och själva responsen kan man t.ex. visa som rådata eller i JSON-struktur om anropet har en svarstyp.

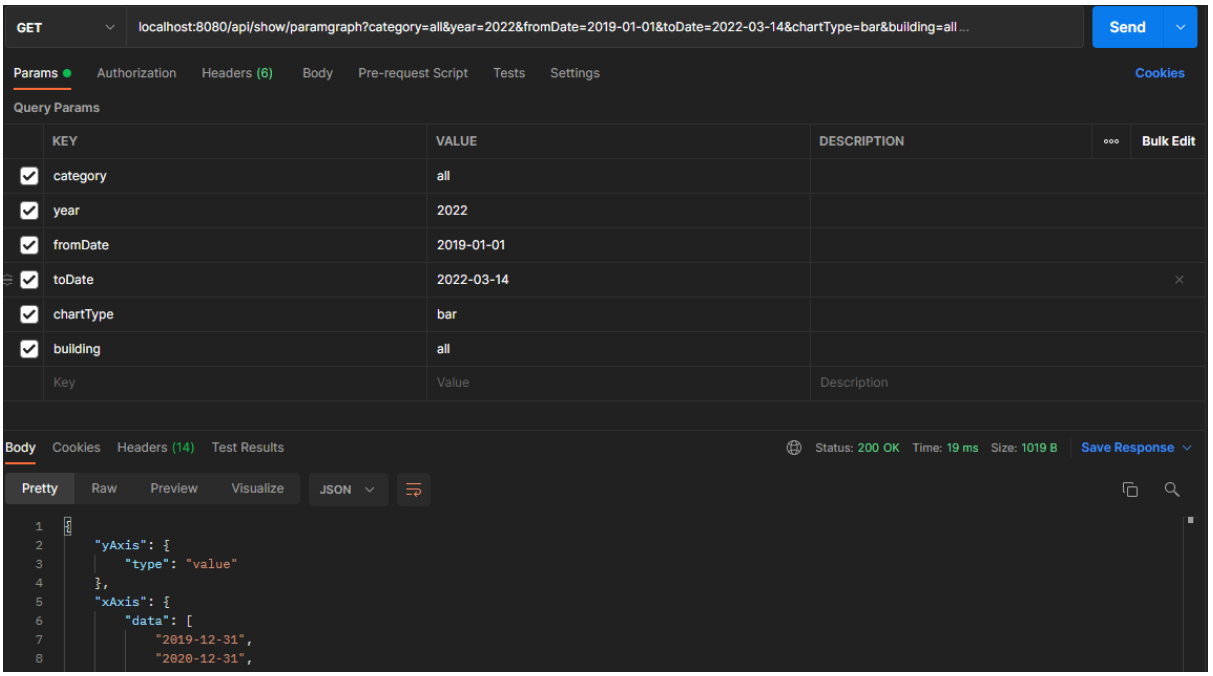


Fig 24, Anrop mot en parameterdriven endpoint

4.3 GitHub

GitHub är ett versionshanteringssystem som lanserades 2008. Det är otroligt populärt och används av mer än 83 miljoner utvecklare och över 4 miljoner organisationer (*Build Software Better, Together*, n.d.). Varje projekt lagras i ett *repository* och kan efter det klonas ner av

vem som helst. Dessa kan vara publika eller privata och de som har tillåtelse kan även *pusha* upp kod med förändringar till samma *repository*.
Det har varit avgörande för att vi skulle kunna jobba två utvecklare med samma projekt. Vi har två stycken delade *repositories*, ett för frontendkoden och ett annat för backendkoden.

Det vi då gjort är att en av oss jobbat i backend och en i frontend. Sedan när man gjort några ändringar så skrivs en "commit" (meddelande om vad man gjort hittills) och sedan *pushas* koden till Github. Då meddelas den andra om att det finns en ändring så den kan göra en *pull* av *repository* och den personen har då samma ändringar som den andra just gjort. Det går även att jobba i samma *repository* så länge man inte jobbar i samma fil.

Detta har alltså gjort det väldigt enkelt för oss att jobba ihop. Vi har båda alltid samma kod och det är enkelt att ta ner den och *pusha* upp sina ändringar utan att behöva maila eller liknande.

4.4 IDE

Som utvecklingsverktyg har vi använt IntelliJ IDEA[16], Spring Tool Suite[17] och Visual Studio Code[18]. Valet av utvecklingsmiljöer är baserat på vad vi föredrar att jobba i. Spring Tool Suite har använts för backend, Visual Studio Code för frontend. Intellij IDEA har använts för både frontend och backend.

[16] https://www.jetbrains.com/idea/
[17] https://spring.io/tools
[18] https://code.visualstudio.com/

5. SYSTEM

5.1 Systemdiagram

I diagrammet nedan, fig 25, finns en överblick på hur alla delar hänger ihop i vårt arbete.

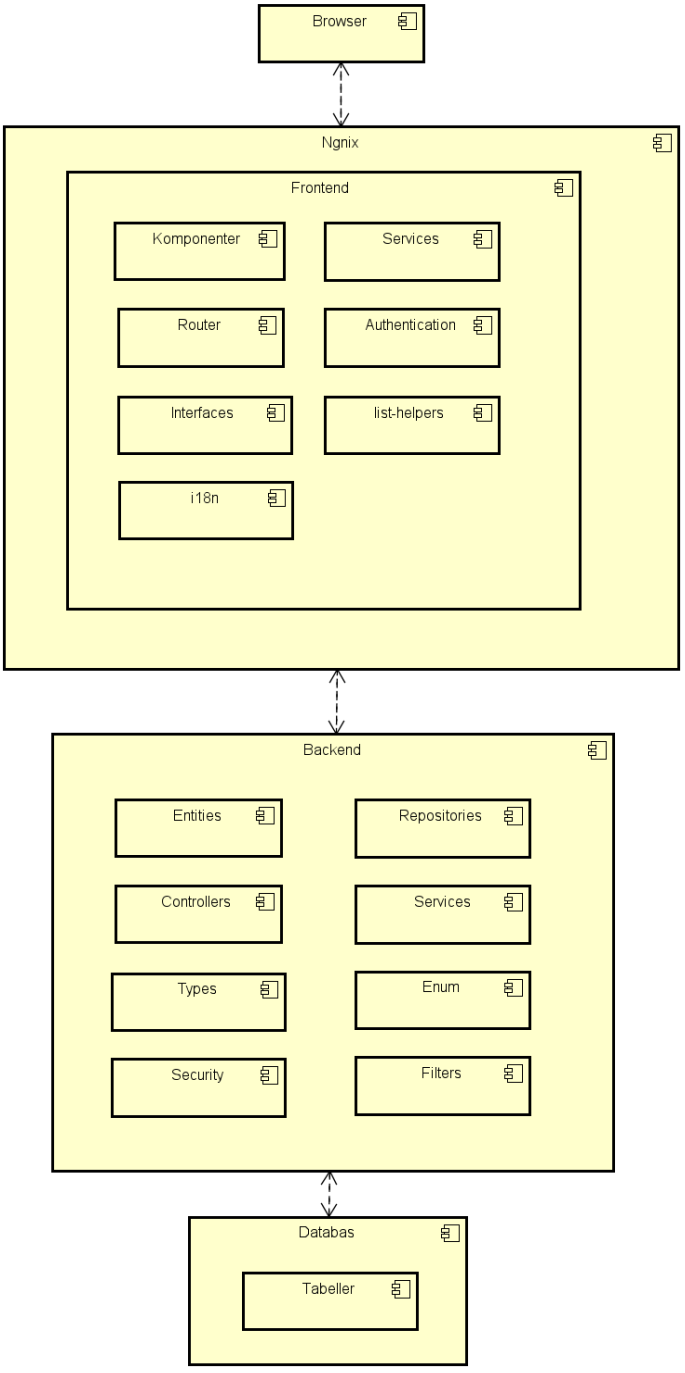


Fig 25, Överblick över systemet

5.2 Produktionssättning

Systemet är produktionssatt i Högskolans IT-labb på en Linuxserver med Ubuntu som operativsystem. Det är denna server som driver alla komponenter i systemet. Databasen drivs av MariaDB. Backenddelen är en JAR-fil med en inbäddad webbserver som heter Apache Tomcat. Frontenddelen drivs av Nginx och till sist är det några *cronjobs* som hanterar backups och regelbundna omstarter. *Cronjobs* är uppgifter som körs automatiskt i bakgrunden i operativsystem.

Konfigurationen av Nginx står för att komprimera webbsidorna till GZIP-format, proxy vidarekoppling[19] till backend samt var SSL[20]-certifikaten finns definieras här. För att bygga själva frontend behöver man köra kommandot “ng build --configuration production”, sedan flytta innehållet från *buildmappen* till Linuxservern. Backend byggs genom att köra kommandot “mvn package” och den resulterande JAR-filen flyttas till Linux-servern.

De *cronjob* som hanterar backups är ett för att dumpa databasen till en fil varje dag kl 01:00 och ett annat jobb kopierar filen till en annan plats och döper om filen till veckodagens namn. Detta sker också varje dag kl 01:15. Det sista jobbet startar om backend-servern varje lördag kl 02:00.

Se bilaga 2 för mer information om denna produktionssättning.

[19] Proxy pass through
[20] Secure Sockets Layer

6. KRAV

Vi har delat upp våra krav i tre kategorier baserat på prioritet. De med högst prioritet är det vi sedan fokuserat på att först bli klara med och sedan jobbat nedåt. Vi tycker denna uppdelning är bra då det blir enkelt att se vad som behöver implementeras i vilken ordning. I figur 26 finns ett *use case diagram* där man ser de nya funktionerna som finns efter arbetet.

6.1 Hög

1. Refaktorering av frontend
2. Refaktorering av backend

6.2 Medium

3. Refaktorering databas
4. Språkval
5. Automatik för förbrukning av el
6. Automatik för förbrukning av fjärrvärme
7. Bilagor ska kunna läggas till i blanketten för reseräkningar

6.3 Låg

8. API-nycklar ska kunna skapas via frontend
9. Hanteringen av SSL-certifikatet ska ske på ett bättre sätt
10. Parameterdrivna infografer ska finnas
11. Diagrammen byts ut till Apache Echarts

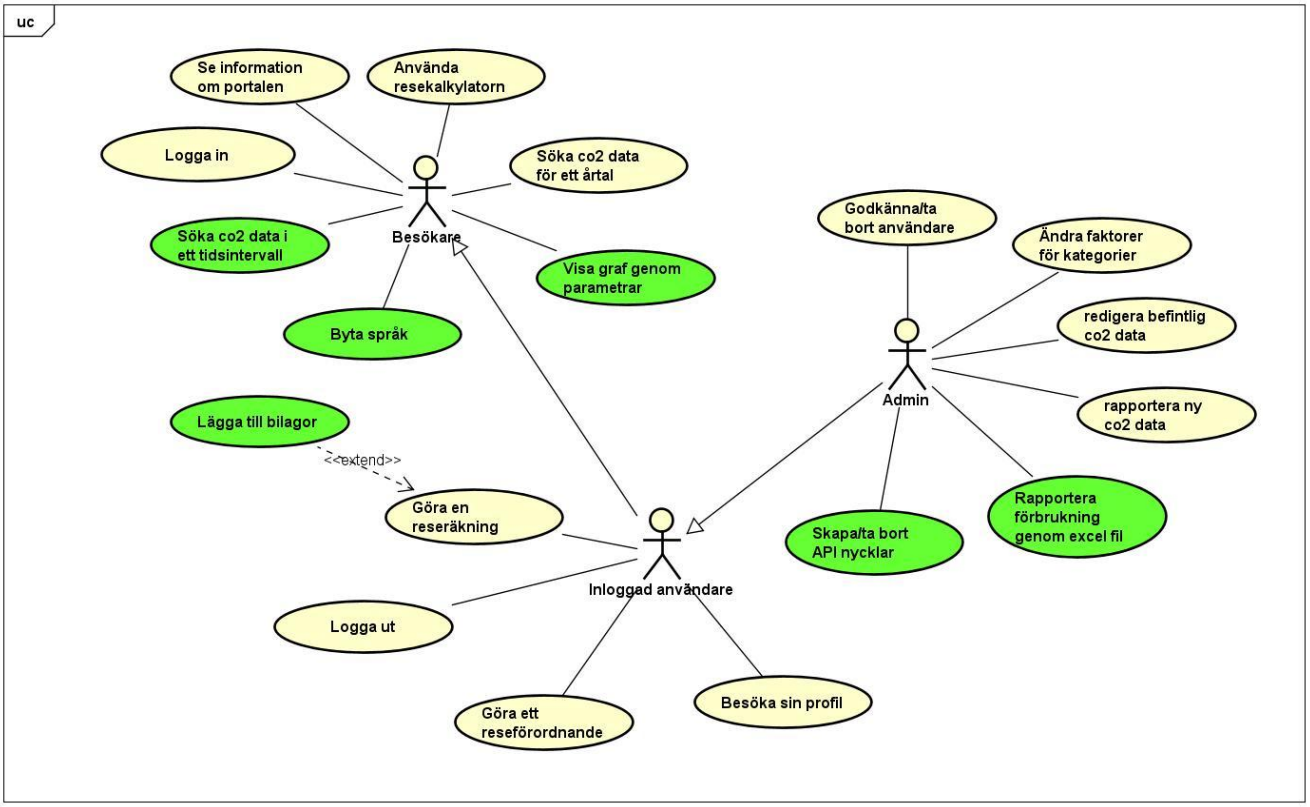


Fig 26, Use case diagram som visar nya funktioner som finns efter arbetet, markerat med grön färg

7. IMPLEMENTATION

7.1 Refaktorering frontend

En av de viktigaste sakerna för oss med hög prioritet detta arbete var refaktoreringen av frontend, som är byggt med ramverket angular. För de olika kategorierna finns det tre olika delar: rapportera ny förbrukning, ändra en befintlig förbrukning och visa alla förbrukningar för ett visst år. Alla dessa kategorier hade sina skilda komponenter för alla dessa delar trots att de inte var mycket skillnad mellan dem, en angular komponent innehåller dessutom fyra filer. Läs mer om Angular i kapitel 3.2. Se fig 27 för hur komponent strukturen såg ut innan refaktorering.

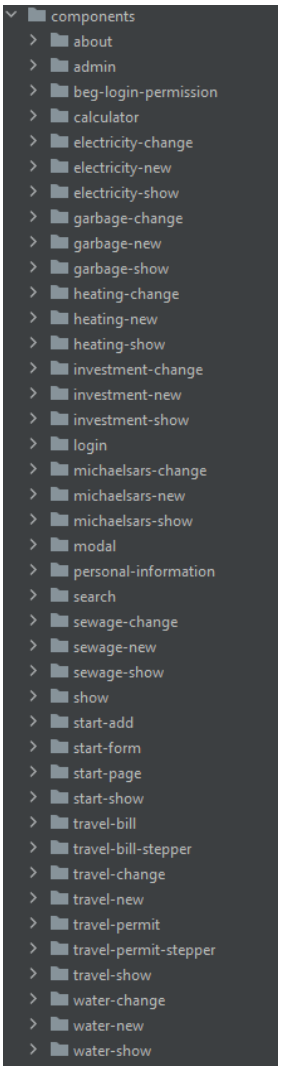


Fig 27, Filstruktur för komponenterna innan refaktorering

För dessa åtta kategorier och tre komponenter vardera för alla landar då på totalt 96 filer. Utöver detta finns även ett interface för varje kategori som i princip innehåller samma sak och dessutom en service, som innehåller två filer, för varje kategori. Allt detta blir alltså väldigt svårnavigerat p.g.a. mängden filer man ska hitta i och gör det nästan omöjligt att snabbt sätta sig in i projektet.

Vi visste alltså att vi måste få ner antal filer för dessa kategorier och kom fram till att första steget var att få ihop komponenterna. Vi skapade tre komponenter: en visnings komponent, en rapporterings komponent och en ändrings komponent. I dessa tre ska alla åtta kategorier in i varje, vilket vi ansåg vara möjligt.

För att göra denna sammanslagning möjlig så använde vi oss av typescripts datatyp “any”. Innan har man använt sig av de olika interfacen för att få en specifik typ för de olika kategorierna. Men nu när vi skulle slå ihop kategorierna så gick inte det och då använde vi oss av “any”. Att frångå strikta variabeltyper är kanske något man annars försöker undvika men i detta fall ansåg vi att det var nödvändigt att använda det. Nackdelen med att använda “any” i utveckling övervägde mängden likadana filer vi annars skulle ha. Detta gör att själva innehållet i de nya komponenterna är i princip identiska med de föregående kategorikomponenterna innan refaktoreringen förutom variebeltyperna.

För att göra det möjligt i routingen att använda samma komponent skickar vi med en datavariabel med vilken typ det är (fig 28) och använde sedan *route.snapshot.data* (fig 29*)* i komponentens typescriptfil i funktionen “*on Init*” för att kunna skriva ut rätt information så som rubriker, inputfält och tabeller.

```
{ path: 'heatingdetail/:id', component: ChangeChangeComponent, data:{type:"heating"}, canActivate: [isAdmin] },
{ path: 'heatingnew', component: ReportReportComponent, data: {type:"heating"}, canActivate: [isAdmin] },
{ path: 'heating', component: ShowShowComponent, data: {type:"heating"}},
```

Fig 28, En data variabel med kategori skickas med till de generella komponenterna

```
this.type = this.route.snapshot.data['type'];
```

Fig 29, Hur kategorin läses in i logikfilen

I HTML-filerna används **ngIf* för att inaktivera fält och taggar som inte behövs för vissa kategorier. Detta används i alla komponenter. Däremot så insåg vi att det skulle bli väldigt rörigt att blanda in resekategorin i dessa komponenter. Den skiljer sig väldigt mycket i rapporteringen och ändringen och höll kvar dem som det var men visningsdelen av resor kunde sättas ihop med de andra. Se fig 30 för hur komponent strukturen ser ut efter refaktorering.

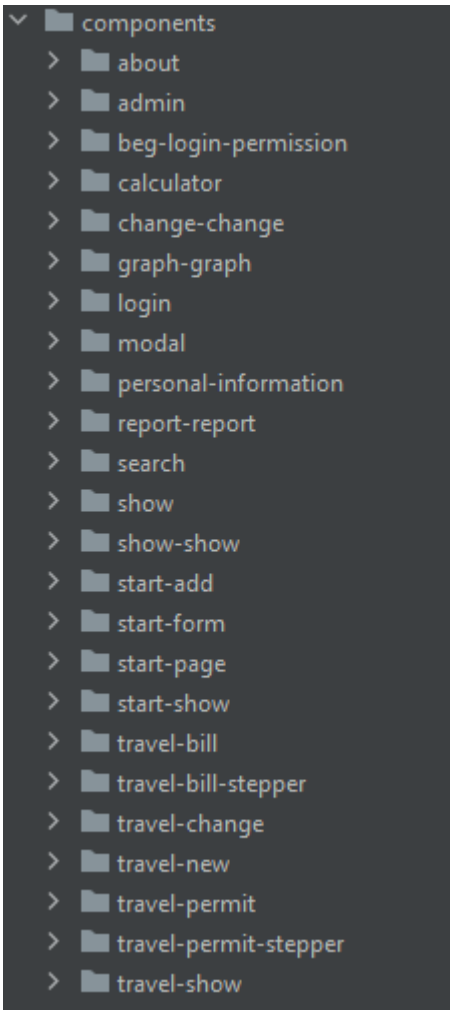


Fig 30, Filstruktur för komponenterna efter refaktorering

Sedan stötte vi på nästa kodduplicering och det var informationshämtningen mot backend, alltså service filerna. Servicefilerna använde sig också av de specifika interfacetyperna och dessutom använde de sig av URL-parametrar för att skicka med år och kategorin. Se fig 31 för hur det såg ut innan refaktorering.

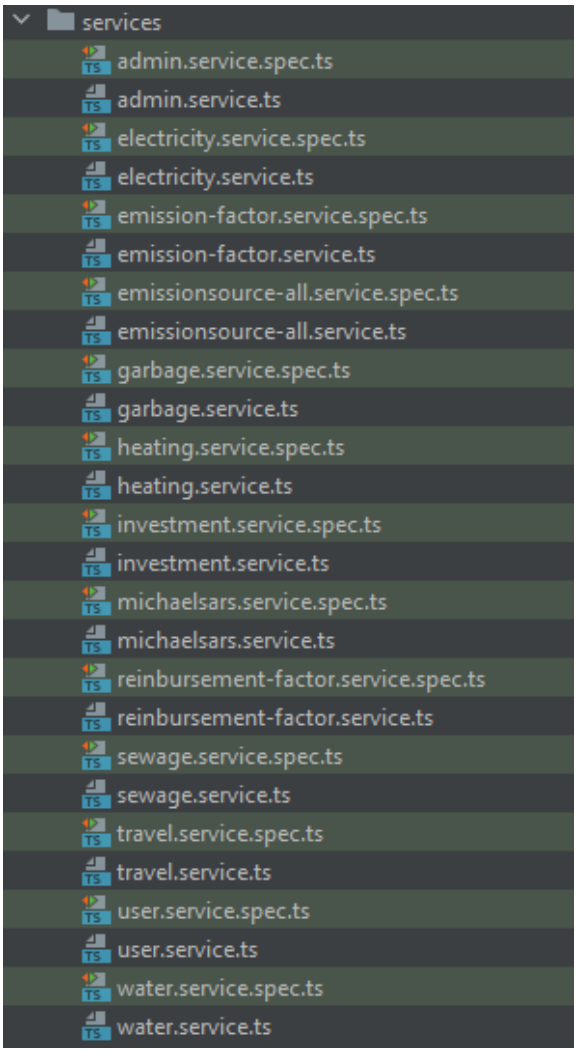


Fig 31, Filstruktur för services innan refaktorering

Detta löste vi genom att istället göra en generell service och skickade med kategori och annan information vi behövde, tex år via *query params*, se fig 32. För att dessutom få ner API anrop skickade vi med en "operation" för att i backend kunna anropa rätt funktioner. Även här använde vi oss av datatypen "any" för att inte binda oss till någon typ.

```
let queryParams = new HttpParams();

queryParams = queryParams.append("type", type);
queryParams = queryParams.append("startDate", startDate);
queryParams = queryParams.append("endDate", endDate);
queryParams = queryParams.append("operation", "searchFromTo");
```

Fig 32, En query param skapas i en service fil

Efter att ha använt “any” som datatyp så användes inte längre interfacefilerna för de olika kategorierna och vi kunde plocka bort dom också. Sju filer till försvann tillsammans med de 14 servicefilerna. Se fig 33 för hur interface mappen såg ut innan refaktorering och fig 34 på hur det ser ut efter.

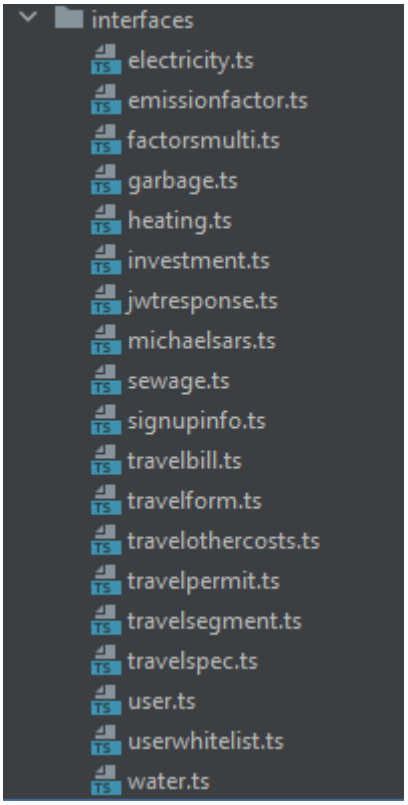


Fig 33, Filstruktur för interface innan refaktorering

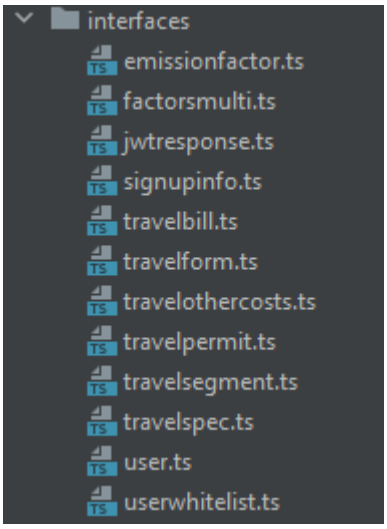


Fig 34, Filstruktur för komponenterna efter refaktorering

Sedan valde vi även att slå ihop två services som hade väldigt liknande funktioner till en gemensam service så att det skulle vara samlat på ett ställe och inte ta upp onödigt mycket plats. Se hur servicestrukturen ser ut efter refaktorering i fig 35.

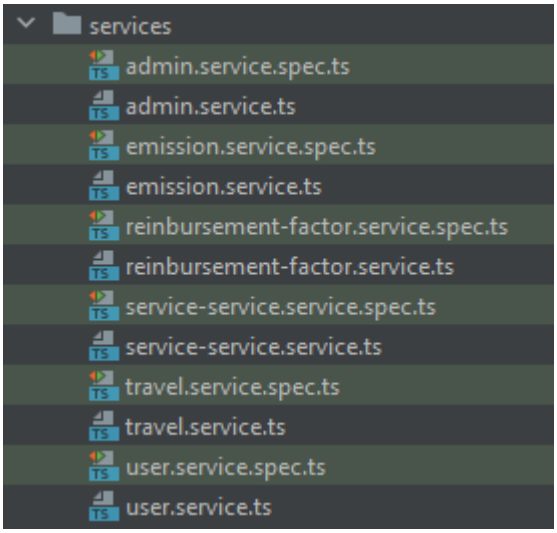


Fig 35, Filstruktur för services efter refaktorering

När vi sedan översatte sidan, se mer i kapitel 7.5, så måste såklart även alla list-hjälpare flyttas dit. De användes i *dropdowns* för att slippa hårdkoda värdena varje gång, därför fanns det i olika filer för de olika ämnena/listorna. När nästan alla dessa listor flyttats till översättningen och informationen hämtades därifrån för att listorna alltid skulle vara på rätt språk så användes list filerna inte längre. Vi kunde därför ta bort dessa åtta filer efter det.

I tabell 1 finns statistik på refaktoreringen av frontend.

Tabell 1, Antal filer och kod som kunnats ta bort efter refaktoreringen sammanställning.

	Då:	***Nu:***	***Förändring:***
Totalt antal filer:	224 st	129 st	-95st
Totalt antal komponenter:	42 st	24 st	-18st
Totalt antal service filer:	26 st	12 st	-14st
Totalt antal interface:	19 st	12 st	-7st
Total antal list-helpers:	11 st	2 st	-9st

7.2 Refaktorering backend

Refaktoreringen av backendservern var den andra stora punkten som vi lade mycket fokus på. *Backend* är skriven i Java med Spring Boot som grund. I fig 36 kan man se hur majoriteten av komponenterna var uppsatta, där varje typ av komponent var duplicerad sinsemellan.

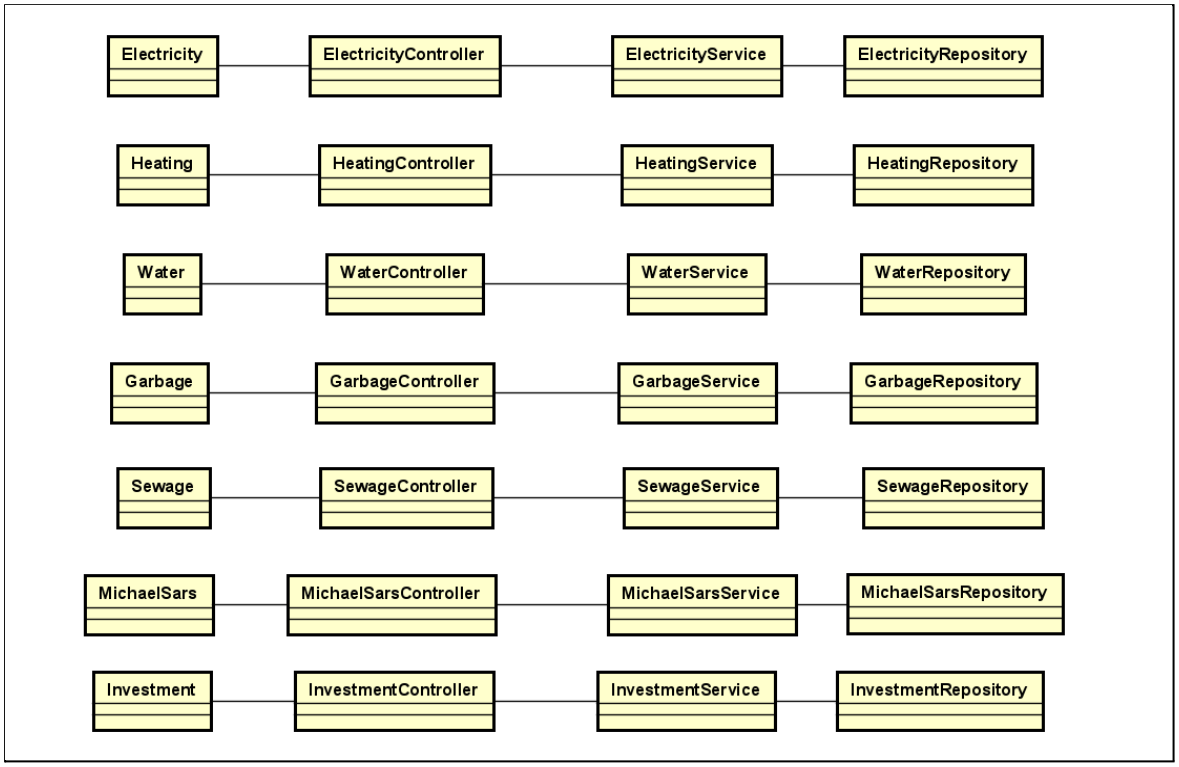


Fig 36, Grov överblick över komponenterna innan refaktorering

Vi började med att abstrahera entitetsklasserna och skapade en mellanklass som håller de fält som är gemensamma mellan alla kategorier samt några unika specialfall för en del av entiteterna. Några av fälten har nu annotationen *@Transient* vilket tillåter oss att ha dessa fält kvar i mellanklassen även om motsvarande fält inte existerar i alla tabeller i databasen. Entiteterna för kategorierna är nu mestadels tomma förutom i de fall där de har sitt eget unika fält samt en jämförelsefunktion. Exempelvis är nu entitets hierarkin för elektricitet: BaseEntity -> MiddleEntity -> Electricity. I figur 37 syns strukturen för mellanentiteten.

```java
@MappedSuperclass
@Getter @Setter @ToString
public class MiddleEntity extends BaseEntity{

    @Basic
    @Temporal(TemporalType.DATE)
    @Column //(unique = true)
    private Date date;

    @Column(nullable = false, precision=8, scale=2)
    private Double co2emission;

    @Column(nullable = false)
    private Integer consumption;
}
```

Fig 37, Strukturen på mellanentiteten

Nästa steg i processen är att ta itu med förråden (repositories) samt dess abstrahering och här steg svårigheten några hack då vi nu måste använda Generics i alla funktioner och i själva interfacet. Eftersom vi inte längre kan ha entitets namnen hårdkodade i SQL- strängarna, måste vi nu använda ett uttryck från SpEL[21] (Spring Expression Language) nämligen "#{#entityName}" och annotationen *@EntityScan* med sökvägen till alla entiteter så att SpEL-uttrycken kan hämta rätt namn till strängen. Se fig 38 för repositoryexempel före refaktorering och fig 39 för exempel på SpEL efter refaktorering.

Annotationen *@NoRepositoryBean* är också ett måste på detta generiska mellan interface, vilket berättar för Spring att detta *repository* inte skall automatiskt instansieras. Det betyder också att alla tjänster som tar in detta generiska interface i sin konstruktor också måste explicit använda "lazy loading" med annotation *@Lazy*. Förvaren för entiteterna är nu tomma men utökar det generiska gränssnittet.

```java
public interface ElectricityRepository extends JpaRepository<Electricity, Long> {

    @Query("SELECT el FROM Electricity el WHERE el.date >= :startDate and el.date <= :endDate")
    List<Electricity> findAllByDateBetween(Date startDate, Date endDate);
```

Fig 38, Repository query innan refaktorering

[21] https://spring.getdocs.org/en-US/spring-framework-docs/docs/spring-core/expressions/expressions.html

```java
@EntityScan("ax.ha.co2.entities")
@NoRepositoryBean
public interface GenericRepository<T, Long> extends JpaRepository<T, Long> {

    @Query("SELECT data FROM #{#entityName} data WHERE data.date >= :startDate and data.date <= :endDate")
    List<T> findAllByDateBetween(Date startDate, Date endDate);
```

Fig 39, Exempel på SpEL

Services visade sig också att vara en utmaning med Generics i bruk, hur en servicefil såg ut innan Generics togs i bruk kan man se i fig 40. En mellanklass skapades här också med alla gemensamma funktioner för alla tjänster. Då vi lagrar vårt repository som GenericRepository<T, Long> i klassen, behöver vi också lägga på annotationen *@Lazy* på konstruktorn så att spring inte försöker koppla ihop tjänsten med en instans av GenericRepository, som vi tidigare satte till *@NoRepositoryBean*, innan tjänsten faktiskt behövs, se fig 41. Tjänsterna för varje entitet har sin egen konstruktor men är annars tomma.

```java
@Service
@Transactional
public class ElectricityService extends BaseService<Electricity> {
    private final ElectricityRepository elRepository;
    private final EmissionFactorService emissionFactorService;

    public ElectricityService(ElectricityRepository elRepository, EmissionFactorService emissionFactorService){
        super(elRepository);
        this.elRepository = elRepository;
        this.emissionFactorService = emissionFactorService;
    }

    //Returns sorted list
    public List<Electricity> findAllByDateBetween(Date startDate, Date endDate){
        List<Electricity> resultList = elRepository.findAllByDateBetween(startDate, endDate);
        Collections.sort(resultList, Collections.reverseOrder());
        return resultList;
    }
```

Fig 40, Service innan refaktorering

```java
@Service
@Transactional
public class MiddleService<T extends MiddleEntity> extends BaseService<T> {
    private final GenericRepository<T, Long> repository;
    private final EmissionFactorService emissionFactorService;
    private final Logger logger = LoggerFactory.getLogger(MiddleService.class);

    @Lazy
    public MiddleService(GenericRepository<T, Long> repository, EmissionFactorService emissionFactorService) {
        super(repository);
        this.repository = repository;
        this.emissionFactorService = emissionFactorService;
    }

    //Returns sorted list
    public List<T> findAllByDateBetween(Date startDate, Date endDate){
        List<T> resultList = repository.findAllByDateBetween(startDate, endDate);
        resultList.sort(Collections.reverseOrder());
        return resultList;
    }
```

Fig 41, MiddleService konstruktör med annotationen Lazy

Controllers är förmodligen där den största skillnaden har skett då vi slog samman alla andra controllers, för de kategorier som finns definierade i kapitel 2 (förutom *travel*), till en och samma controller (MainController.java). Alla åtta metoder som skapades av sammanslagningen i denna controller tar in parametern type som berättar om vilken kategori som efterfrågas. Utöver denna parameter tas även in ett flertal andra parametrar beroende på vilken endpoint som anropas.

Kommunikationen mellan de olika komponenterna i grova drag är nu att MainController använder sig av MiddleEntity istället för de enskilda entiterna och MiddleService används istället för kategoriernas tjänster. MiddleService i sin tur använder sig av MiddleEntity och GenericRepository istället för de enskilda entiteterna och förvaren (se fig 42).

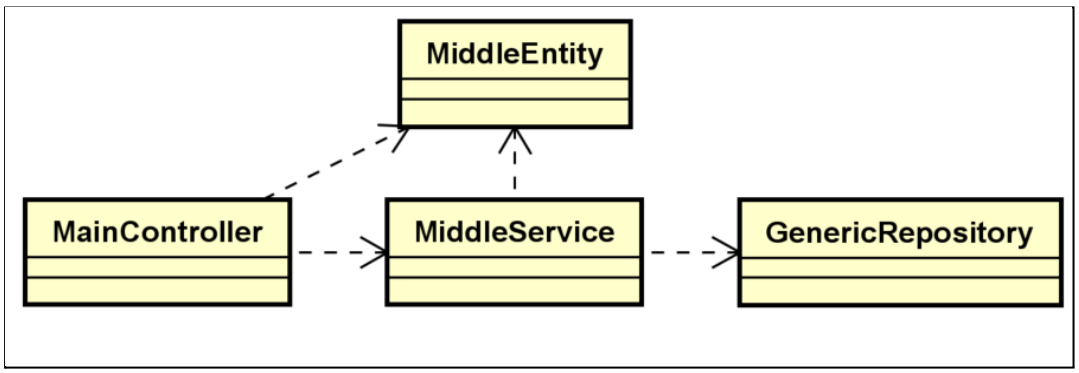


Fig 42, Grov överblick över kommunikationen mellan komponenterna

Då alla tjänster är generaliserade kan vi nu enkelt sätta den aktiva tjänsten baserat på den typ som kommit in med ett anrop istället för att ha en metod för varje kategori, se fig 43. De metoder som behöver hantera entiteter kan nu med hjälp av en hjälpmetod använda den generaliserade entiteten för alla kategorier, se fig 44.

```
private void setActiveService(String type) {
    this.activeService = switch (type) {
        case "electricity" -> (V) elService;
        case "heating" -> (V) heatingService;
        case "garbage" -> (V) garbageService;
        case "investment" -> (V) investmentService;
        case "michaelSars" -> (V) michaelSarsService;
        case "sewage" -> (V) sewageService;
        case "water" -> (V) waterService;
        default -> null;
    };
}
```

Fig 43 Hjälpfunktion för att sätta aktiv service

```
private MiddleEntity getType(String type) {
    return switch (type) {
        case "electricity" -> new Electricity();
        case "heating" -> new Heating();
        case "garbage" -> new Garbage();
        case "investment" -> new Investment();
        case "michaelSars" -> new MichaelSars();
        case "sewage" -> new Sewage();
        case "water" -> new Water();
        default -> new MiddleEntity();
    };
}
```

Fig 44, Hjälpfunktion för att få rätt entitet

De åtta metoder som står för majoriteten av huvudkontrollerns funktionalitet:

1. **getAllData-** Denna metod tar även in parametern “operation” som minimi utöver parametern “type” som tidigare nämnt, dessa två parametrar berättar vad för data och hur datat ska se ut. De fyra övriga parametrarna är valfria och ger mer specifik data i retur (se fig 45).

```
@GetMapping("/show")
public ResponseEntity<Iterable<T>>  getAllData(
        @RequestParam String type, @RequestParam String operation,
        @RequestParam(required = false) Optional<Integer> year,
        @RequestParam(required = false) Optional<String> startDate,
        @RequestParam(required = false) Optional<String> endDate,
        @RequestParam(required = false) Optional<String> investmentType)
```

Fig 45, Parameter input för en endpoint

2. **getDataById**

   Metoden getDataById returnerar en resurs baserat på type och ID, se fig 46.

```
@GetMapping("/data")
public ResponseEntity<Optional<T>> getDataById(@RequestParam String type, @RequestParam Long id) {
    try {
        setActiveService(type);
        return new ResponseEntity<>(this.activeService.findById(id), HttpStatus.OK);
    } catch(Exception e) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Fig 46, Endpoint metod för att hämta en resurs

3. **getDataHistory**

   Denna metod returnerar de fem senaste insättningarna för given kategori.

4. **getGraphByYear**

   Denna metod returnerar grafdata för en given kategori samt årtal eller intervall.

5. **postData**

   Metoden postData tar in en entitet från frontend och sätter in den i rätt tabell för den givna kategorin.

6. **deleteData**

   Metoden tar bort resursen baserat på den givna kategorin och ID:et som kommit in med anropet.

7. **updateData**

   Metoden uppdaterar entiteten på det givna ID:et med den nya entiteten som kommit in med anropet, i fig 47 kan man se hur detta hanteras.

```java
@PutMapping("/data")
public ResponseEntity<T> updateData(@RequestBody T body, @RequestParam String type, @RequestParam Long id) {
    setActiveService(type);
    Optional<T> existingData = activeService.findById(id);
    if (existingData.isPresent()) {
        T _data = existingData.get();
        _data.setDate(body.getDate());
        _data.setConsumption(body.getConsumption());
        switch (type) {
            case "sewage", "water", "garbage":
                _data.setProperty(body.getProperty());
                _data.setCo2emission(activeService.calculateCO2(body.getConsumption()));
                activeService.save(_data);
                return new ResponseEntity<>(_data, HttpStatus.OK);
            case "michaelSars":
                _data.setCo2emission(activeService.calculateCO2(body.getConsumption()));
                activeService.save(_data);
                return new ResponseEntity<>(_data, HttpStatus.OK);
            case "investment":
                _data.setInvestmentType(body.getInvestmentType());
                _data.setCo2emission(activeService.calculateCO2(body.getConsumption(), body.getInvestmentType()));
                activeService.save(_data);
                return new ResponseEntity<>(_data, HttpStatus.OK);
            case "heating", "electricity":
                _data.setProperty(body.getProperty());
                _data.setCo2emission(activeService.calculateCO2(body.getConsumption()));
                int totalMinutes = _data.getTimeIntervalInMinutes();
                double consumptionPerMin = (body.getConsumption() * 1000.0) / totalMinutes;
                _data.setConsumption_Wh_PerMin(consumptionPerMin);
                activeService.save(_data);
                return new ResponseEntity<>(_data, HttpStatus.OK);
            default:
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

Fig 47, Endpointmetod för att uppdatera en resurs

I tabell 2 finns statistik på refaktoreringen av backend.

Tabell 2 Refaktoreringsstatistik för backend

	Då	*Nu*	*Förändring*
Controllers rader kod	3 181	1 230	-1 951
Entities rader kod	944	851	-93
Repositories rader kod	529	433	-96
Services rader kod	1 797	985	-812
Totalt rader kod	6 451	3 499	-2 952
Totalt antal filer	90	90	+/-0

7.3 Refaktorering databas

Vi började med att planera en tänkt struktur på databasen, hur alla tabeller skulle se ut, men vi konstaterade efter ett tag att det inte var nödvändigt att göra en omstrukturering av hela databasen.

Istället lade vi fokus på lite mindre optimeringar av tabellerna. En av dessa var att normalisera tabellen för utsläppsfaktorer till två skilda tabeller, se fig 48. Orsaken till denna normalisering var att det blev mycket dubbletter av attributvärden, i och med denna optimering är det nu lättare att lägga till helt nya faktorer samtidigt som man kan se till att det inte finns dubletter av vissa typer.

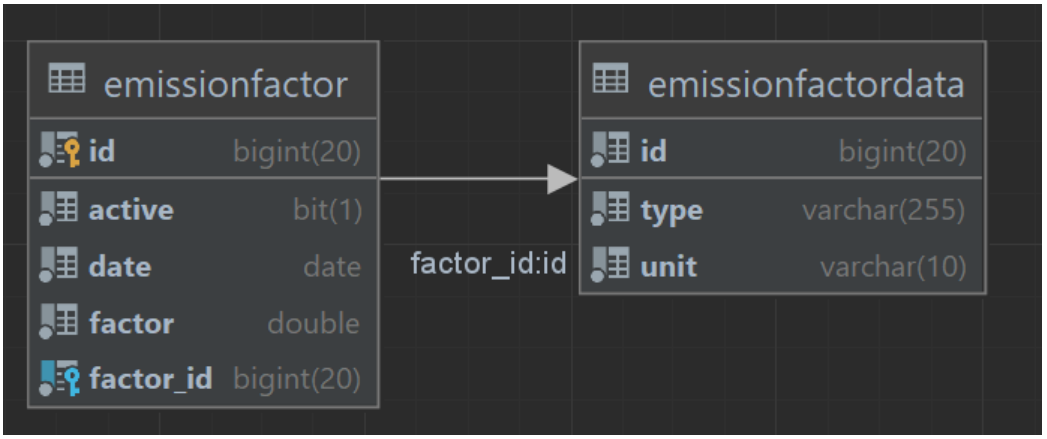


Fig 48, Tabellbeskrivning av emissionfactor och emissionfactordata

Vi korrigerade några attributnamn i en del av tabellerna för att få samma struktur på dem, detta så att backend inte behöver ha specialfall för vilken tabell som efterfrågas. Till exempel ändrades attributnamnet i tabellen “michaelsars” från “used_fuel” till “consumption” för att matcha de övriga kategorierna.

Vi lade även till en ny tabell för attestering(billables) så att man mycket enklare skall kunna redigera, lägga till eller ta bort attesteringsmål istället för att ha dem hårdkodade i både backend och i frontend. En koppling mellan billables tabellen och travelspec tabellen gjordes, se fig 49.

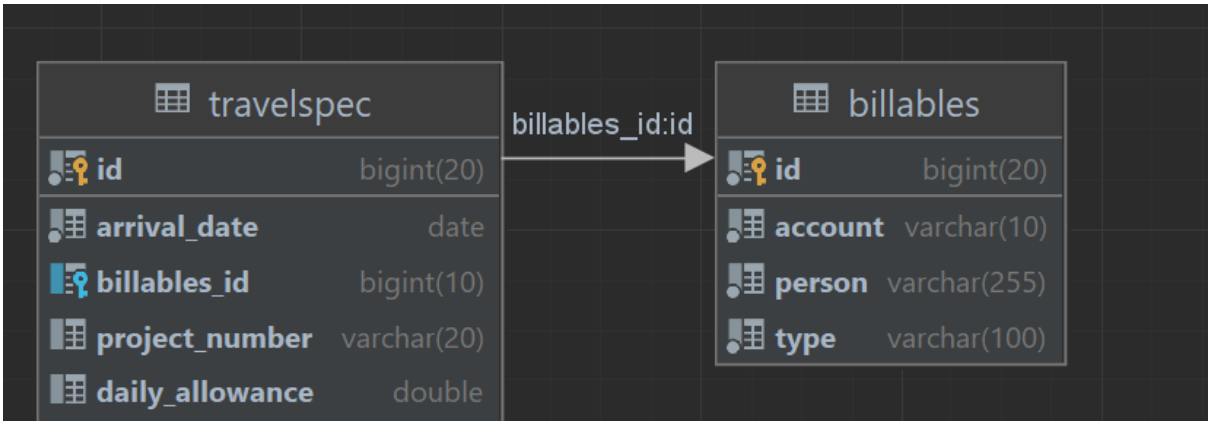


Fig 49, Tabellbeskrivning av ny tabell billables och dess koppling till travelspec

7.4 Automatik av förbrukningar

Av alla de kategorier som det just nu redovisas koldioxidutsläpp för är det få av dem som kan ske automatiskt. De måste skrivas in manuellt då det helt enkelt inte går att göra det på något annat sätt. Men det finns två av dessa som läses av hela tiden, el och fjärrvärme. I vårt fall läses det av konstant av en tredje part(elbolaget) som sammanställer detta data och kan sedan ses via elbolagets portal. P.g.a. detta så tänkte vi att det inte skulle vara svårt att få till en automatisk inläsning till vår portal bara vi fick det som existerade att koppla mot portalen.

Men det var inte fullt så enkelt som vi trodde utan vi fick göra det på helt andra sätt än vad vi tänkt och i slutändan blev det inte så automatiskt som vi tänkt oss. I norra byggnaden var det helt fel sorts mätare för att vi skulle kunna läsa något av det. På södra fanns det rätt sorts mätare men de är däremot så gamla att de inte går att använda i alla fall. Att byta ut dessa

mätare gick inte heller pga komponentbrist, så det gick helt enkelt inte att läsa av dem direkt och vi fick fundera ut något annat.

Ett API skulle gjort detta väldigt enkelt och något vi blev lite förvånade över att inte finns redan. Allt data finns och samlas redan på en portal så steget att göra detta återanvändbart m.h.a. ett API t.ex. så att man kan använda detta till vad som helst borde inte vara så stort. Men det finns alltså inte i dagsläget och det alternativet var då också uteslutet.

Det vi istället fick använda var det Excelblad man kan få ut m.h.a. inloggning på elbolagets portal. Det är ett ark med en tidsstämpel i första kolumnen, datum först och sedan resten av dygnets timmar. Den andra kolumnen är eldata för de respektive timmarna och även fjärrvärme med samma koncept, se fig 50.

1		El	Fjärrvärme
2		kWh	MWh
3	1.3.2022	0,7	0,007
4	01	0,64	
5	02	0,63	0,003
6	03	0,59	0,004
7	04	0,57	0,007
8	05	0,54	
9	06	0,57	0,008
10	07	0,73	
11	08	0,78	0,004
12	09	0,52	0,01
13	10	0,01	0,003
14	11	0	0,003
15	12	0	0,002
16	13	0	0,002
17	14	0,03	
18	15	0,07	0,004
19	16	0,54	0,006
20	17	1,34	0,004
21	18	0,76	0,003
22	19	1,02	
23	20	1,03	0,01
24	21	2,33	0,003
25	22	0,89	0,003
26	23	1,2	0,003

Fig 50, Excelarket som fås från leverantören

Denna Excelfil(.xlsx) laddas alltså upp via vårt frontend, på adminfliken. Förutom filen finns det en *dropdown* för att välja vilken byggnad datat tillhör. Då detta inte står i filen måste användaren välja det. Denna fil bearbetar vi sedan i frontenddelen m.h.a. biblioteket xlsx/SheetJS, se fig 51.

```
file:File;
arrayBuffer:any;
excelFile: any;

fileReader.readAsArrayBuffer(this.file);
fileReader.onload = (e) => {
      this.arrayBuffer = fileReader.result;
      let data = new Uint8Array(this.arrayBuffer);
      let arr = new Array();
      for(let i = 0; i != data.length; ++i) arr[i] =
String.fromCharCode(data[i]);
      let bstr = arr.join("");
      let workbook = XLSX.read(bstr, {type:"binary"});
      let first_sheet_name = workbook.SheetNames[0];
      let worksheet = workbook.Sheets[first_sheet_name];
      this.excelFile =
XLSX.utils.sheet_to_json(worksheet,{raw:true});
      this.excelFile.splice(0,1);
}
```

Fig 51, Hur excel filen läses in i frontend

När vi sedan har hela filen inläst går vi igenom den rad för rad och plockar ut data. Alla dagar ser likadana ut så det var enkelt att bygga upp en funktion som plockade ut detta. Den första raden innehåller datumet så det plockas där men måste sedan skrivas om då det är i fel format mot vår databas. Sedan räknas alla värden för dygnet ihop, el och värme skilt. Värme fås i fel enhet från filen mot vad vi har i vår databas så där måste vi räkna om det för att få det i rätt enhet. Efter dygnets alla timmar skapas ett objekt med datumet, byggnad och värdet som räknats ihop. Detta sätts sedan in i en räcka. När alla rader är lästa så skickas det till backend där räckan loopas igenom och varje position sätts in i databasen. Detta sker på samma sätt för både el och fjärrvärme. Se fig 52.

```
 let heatData: any = [];
this.excelFile.forEach((element,idx) => {
     if(idx % 24 === 0 && idx !== 0){
         let formDataHeat: any = {};
         formDataHeat["date"] = this.dateCreator(date);
         formDataHeat["property"] =
this.readElHeat.value.property;
         formDataHeat["consumption"] = heatSum;
         heatData.push(formDataHeat);
         heatSum = 0;
    }
    if(!isNaN(element.Fjärrvärme)){
       heatSum += element.Fjärrvärme*1000;
    }
    if(idx % 24 === 0){
       date = element.__EMPTY;
    }

private dateCreator(date:String) {
    let tmp = date.split(".");
    tmp[1] = tmp[1].padStart(2,'0');
    tmp[0] = tmp[0].padStart(2,'0');
    return tmp[2].trimEnd() + "-" + tmp[1] + "-" + tmp[0];
}
```

Fig 52, Hur excel filen tolkas

7.5 Språkval

För att andra skolor och övriga Finland ska kunna ta del av högskolans portal var det viktigt att få portalen översatt till åtminstone engelska men helst finska också. För att göra denna internationalisering, beskriven närmare i kapitel 3.2.8, använde vi oss av ngx-translate.

Ngx-translate är ett bibliotek som hjälper till med internationaliseringen för Angular. Den låter en definiera ens översättningar för olika språk för att sedan väldigt enkelt byta mellan

dem. Den ger dig även tillgång till en översättningsservice och en pipe för att hantera dynamisk eller statiskt data[22].

För att få detta att funka behöver man sätta till TranslateModule i rot modulen och sedan definiera standardspråket, se fig 53. Sedan kan man importera översättningsservicen i de komponenterna man behöver.

```
translate.setDefaultLang('sve');
```

Fig 53, Standardspråket defineras

Vi skapade då tre stycken JSON-filer i *assetsmappen*, en fil för varje språk som ses i fig 54. Sedan började vi med att få allt flyttat till den svenska filen och sätta till alla pipes så att översättningen skulle ske automatiskt. För att göra de andra filerna så använde vi som tidigare nämnt Babel som är beskrivet närmare i kapitel 4.1 och går att läsa mer om där.

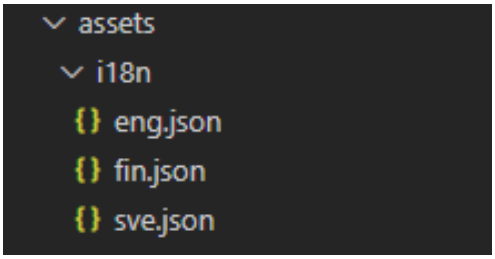


Fig 54, Hur språkfilerna ser ut i assetsmappen

För att byta språk så använder vi oss av knappar som föreställer flaggor, som de flesta andra webbplatserna. Här nedan, fig 55, ett exempel på den brittiska flaggan som då byter språket till engelska.

```
<span class="flag-icon flag-icon-gb"
(click)="useLanguage('eng')"></span>
```

Fig 55, Hur en språkbytarknapp ser ut

Den knappen utlöser sedan funktionen i fig 56 som byter språket, inga konstigheter.

[22] http://www.ngx-translate.com/

```
useLanguage(language: string): void {
    this.translate.use(language);
  }
```
Fig 56, Funktion som byter språk

Det finns olika sätt att hämta ut texten från översättningsfilerna och vi gör det nästan överallt enligt exemplet i fig 57. Vi har valt att dela in våra översättningsfiler enligt de komponenter vi har. I figuren är "calculator.title" ID för själva texten som sedan visas i komponenten. Kollar man figuren 57 och sedan i fig 58 så kan man se att det på svenska kommer stå: Resekalkylator.

```
<h3>{{ 'calculator.title' | translate }}</h3>
```
Fig 57, Rubrik med automatisk översättning

```
"calculator":{
        "title": "Resekalkylator",
        "text1": "Vill du veta hur mycket koldioxid ditt färdsätt
genererar? Det kan du kolla med vår kalkylator här nedan!",
        "text2": "Distans i km",
        "text3": "Färdsätt",
        "text4": "Räkna",
        "text5": "Ditt valda färdsätt genererar ",
        "text6": "kg ",
        "text7": "koldioxid"
    },
```
Fig 58, Hur kalkylatorn är uppbyggd i språkfilen med ID

Innan hade vi *dropdowns* i *list-helpers* för att slippa hårdkoda det många gånger men nu när det skulle översättas så var vi tvungna att komma med en annan lösning. I HTML-filen hämtar vi nu datat direkt från översättningsfilen (fig 59) och får det då på rätt språk direkt. I fig 60 syns vad det är som hämtas.

```
<select class="mx-2" formControlName="position" required
id="position">
   <option *ngFor="let pos of ('positions.data' | translate)"
[ngValue]="pos.dbname" >
       {{ pos.name }}
  </option>
</select>
```
Fig 59, Hur en array som finns i översättningsfilen används i HTML

```
"positions":{
        "title": "positioner",
        "data":[
            {
                "name": "Studerande",
                "dbname": "POSITION_STUDENT"
            },
            {
                "name": "Extern",
                "dbname": "POSITION_EXTERNAL"
            },
            {
                "name": "Personal",
                "dbname": "POSITION_STAFF"
            }
        ]
    },
```
Fig 60, Hur en list helper/array är uppbyggd i översättningsfilen med ID.

Det går även att hämta översättningar från filerna enligt figurerna nedan. Via *translate service* (fig 61) finns en GET-funktion som tar emot ett ID och returnerar strängen som tillhör ID:et (fig 62). Vi använder oss av detta i bl.a. de generella filerna vi skapat för att kunna skriva ut vilken kategori man valt. Typen skickas då till servicen och sparas i en variabel som sedan skrivs ut i HTML-filen.

```
private translate: TranslateService
```
Fig 61, Definiering av translate service i en komponent

```
this.translate.get(this.type).subscribe(translations =>{
     this.translatedTitle = translations;
   });
```
Fig 62, Translate service används för att hämta översättningar genom ID

7.6 Skapa API-nycklar via frontend

En annan efterfrågad sak av kunden var att kunna skapa och ta bort API-nycklar via gränssnittet. En API-nyckel fungerar som en autentiserare. För rapportering av CO2 data finns formulär via gränssnittet men det finns även möjlighet att rapportera direkt till backend via det öppna API:et vi implementerat i POP-kursen. Om man ska rapportera direkt till backend behövs denna API-nyckel för att visa att man har behörighet att göra det. Möjligheten att skapa nycklarna sattes in i adminvyn då vi ansåg att vem som helst inte ska kunna skapa dem eller dela ut dem. Bild på hur segmentet blev att se ut i portalen i fig 63.

Det som går att göra är att visa alla existerande nycklar, generera en ny nyckel och ta bort en existerande nyckel. När en nyckel genereras så ska användningsområde skrivas med också, detta för att enklare hålla koll på varför de skapas. Själva nyckeln slumpas sedan fram och sätts in tillsammans med användningen i databasen.

Fig 63, Hur segmentet ser ut på den riktiga portalen

I backend blev det totalt tre nya mappings:

- get
- post
- delete

Get metoden returnerar en lista på alla API-nycklar som finns. Postmetoden tar emot identifiern/användningsområdet och slumpar nyckeln m.h.a. UUID, se fig 64. Deletemetoden tar emot nyckeln och tar bort den om den existerar. Ingen komplicerad kod utan lätta korta funktioner.

```
String key = UUID.randomUUID().toString();
```

Fig 64, Hur en API-nyckel slumpas fram

Databastabellen fanns redan, det vi skapade var identifiern eller användningsområdet som vi kallar det för i gränssnittet, fig 65.

Column	Type
id	bigint
apikey	varchar(255)
identifier	varchar(200)

Fig 65, Hur databastabellen för API nycklar är skapad

7.7 Lägga till bilagor i reseräkning

Vi började med att skapa en ny databastabell. Där hade vi filnamn, filtyp, själva datat/ innehållet i filen samt ett ID till vilken resa filen hör till. Vi valde att lägga det som en egen tabell då vi tyckte det skulle bli rörigt att lägga in det i någon av de befintliga tabellerna, fig 66.

Column	Type
id	bigint
fileName	varchar(150)
fileType	varchar(45)
data	mediumblob
travelspec_id	bigint

Fig 66, Hur databastabellen för bilagor är skapad

Efter detta började vi med backendbiten. Vi lagade en ny entitet som matchade databas tabellen, en ny service samt *repository*. De olika mappningarna lade vi in i *travel controller* för att ha allt i på samma ställe, men servicen och *repository* tyckte vi blev bäst att ha i egna filer.

Första mappingen som gjordes var att ladda upp en fil, en HTTP-post. Det skickades in i funktionen som en MultipartFile tillsammans med resans ID. Från den kunde vi sen plocka ut namn genom *Stringutils.cleanpath()* och där att anropa *getOriginalFilename()* på själva filen. Filtypen fick vi genom att anropa *getContentType()* på filen som skickades in. Datat fick vi genom *getBytes()*. Se fig 67 för exempel. Detta testades sedan med Postman, beskrivet i kapitel 4.2, för att se att det funkade. När vi justerat några detaljer så funkade det och vi kunde gå vidare till att hämta filer istället.

```
public TravelAttachment uploadFile(MultipartFile file, Long specId){
     try {
           TravelAttachment dbFile = new TravelAttachment();
           String fileName = StringUtils.cleanPath(file.getOriginalFilename());
           dbFile.setFilename(fileName);
           dbFile.setFiletype(file.getContentType());
           dbFile.setData(file.getBytes());
           dbFile.setTravelspec(travelSpecService.findById(specId).get());
           travelAttachmentService.save(dbFile);
          return dbFile;
     }catch(Exception e) {
          return null;
     }
    }
```

Fig 67, Hur en fil sätts in i databasen från backend

Att hämta filerna från backend var inte svårt. Vi skickade in id på resan till mappningen och fick tillbaka en lista med filerna som tillhör den, fig 68. Detta testades också med Postman för att kontrollera att det funkade, vilket det också gjorde.

```
List<TravelAttachment> ta = this.travelAttachmentService.getFilesBySpecId(id);
```

Fig 68, Hur filerna hämtas i frontend från databasen via backend

Efter detta satte vi till stöd för att ladda upp flera filer samtidigt istället för bara en. Detta skedde genom att flytta mappningen till en annan funktion som tog emot en *array* av Multipart files istället för bara en. Vi loopade igenom räckan och skickade in en fil i taget till den gamla funktionen som laddar upp bara en fil.

Eftersom backendbiten gick väldigt fort och vi fick allt att fungera utan problem så trodde vi att även frontendbiten skulle gå lika lätt men där hade vi stora problem på många olika ställen. Det vi först fick problem med var att man inte kan använda vanliga inputtaggar med *type=”file”* i formulär strukturen vi använder. Så vi fick börja söka på andra sätt att ta in filerna, till sist hittade vi ngx-mat-file-input. På den kunde vi enkelt sätta på vilka filtyper som stöds(accept) och om man kan skicka in flera(multiple) filer, se fig 69.

```
<strong> Bilagor/kvitton: </strong>
  <mat-form-field>
    <mat-label>Endast PDF & PNG filer </mat-label>
    <strong
      style="color:red"
      *ngIf="errorText == true">
          Filtypen inte tillåten!
    </strong>
    <ngx-mat-file-input
      formControlName="files"
      [accept]="'.pdf, .png'"
       multiple>
    </ngx-mat-file-input>
    <button> Välj fil</button>
  </mat-form-field>
```

Fig 69, Hur formulärfältet för att ladda upp bilagor ser ut

Efter detta kom de största problemet och det var hur vi skulle få filen och dess innehåll till backend. Vi visste ju att backendfunktionerna funkade eftersom vi testat med Postman men vad vi än gjorde så fick vi inte filen från frontend till backend. Hur vi till sist löste det vara att först kolla filtypen och att du inte laddat upp något annat än PDF eller PNG, fig 70. Ifall användaren matat in något annat visas ett felmeddelande.

```
const file_form: FileInput = this.fourthFormGroup.value.files;
file = file_form.files;
allOk = true;
file.forEach((item) => {
  if(item.type !== "application/pdf" && item.type !== "image/png")
  {
    this.errorText= true;
    allOk = false;
   }
})
```

Fig 70, Kontroll av filformatet

Ifall filtypen/filtyperna stämmer så kommer man till följande loop. Där skapade vi en ny formdatavariabel, som används specifikt för filhantering i Angular. På denna variabel lade vi två nycklar, files och properties. Files nyckeln innehåller hela filen och namnet medan properties innehåller en blob, se fig 71. Detta skickas sedan till servicen för resor och vidare till backend för att lägga till det i databasen, se fig 72.

```
const formData: FormData = new FormData();
file.forEach((f)=>{
  formData.append("files",f,f.name);
});
  formData.append("properties",
    new Blob([JSON.stringify(file)],{type:"application/json"}));
  this.travelService.upload(formData,
    this.specId).subscribe((file)=>{},(err)=>{});
```

Fig 71, En formdata variabel skapas och skickas till servicen

```
upload(files:any, specId: number):Observable<any>{
    let queryParams = new HttpParams();
    queryParams = queryParams.append("specId", specId);
    const url = `api/upload`;
    return this.http.post(url, files, {params:queryParams}).pipe(
      catchError(err => throwError(err))
    );
}
```

Fig 72, Funktionen i service filen som skickar bilagorna till backend för att laddas upp till databasen

Blanketten/formulären är uppdelade i flera olika steg och när användaren kommer till sista steget så ska hela färdiga blanketten synas. Vi valde här att lägga till PDF och bilder som länkar sist i blanketten. Även här stötte vi på problem, men denna gång att vissa funktioner fungerar bara i vissa webbläsare men inte i andra. Därför fick vi ta den lösning som funderade i flest webbläsare, i detta fall att ha en länk som laddar ner PDF:en eller bilden när man trycker på den. För att hämta detta från databasen och skriva ut det som en länk så krävdes en del försök.

Först var vi tvungna att behandla datat olika beroende på om det är en bild eller en PDF. Namnet på filen skulle plockas ut för att kunna användas som länknamn. För att sedan få en URL som kunde användas för att visa innehållet och ladda ner det krävde det att vi tog fildatat och skickade det till funktionen *sanitizer.bypassSecurityTrustUrl()*. Se fig 73.

```
this.travelService.getBlobs(this.specId).subscribe((b)=>{
      b.forEach((item)=> {
        if(item.filetype === "image/png") {
          let objectURL = ''+item.data;

this.imgSrc.push(this.sanitizer.bypassSecurityTrustUrl(objectURL))
;
          let imgIdx = this.imgSrc.length-1;
          this.imgSrc[imgIdx].name = item.filename;
        } else if(item.filetype === "application/pdf") {

this.origPdfSrc.push({src:"data:application/pdf;base64,"+item.data
});
          let pdfIdx = this.origPdfSrc.length-1;
          this.origPdfSrc[pdfIdx].name = item.filename;

this.pdfSrc.push(this.sanitizer.bypassSecurityTrustUrl(this.origPd
fSrc[pdfIdx].src));
          this.pdfSrc[pdfIdx].name = item.filename;
        }
      });
    })
```

Fig 73, Hur filerna hämtas från databasen via backend och sedan behandlas för att kunna användas

Sedan kunde vi skriva ut det som en länk i slutet av blanketten, fig 74.

```
<h5 class="mv-4">Bilagor</h5>
<div *ngFor="let pdf of pdfSrc">
   <p>_____</p>
   <a [href]="pdf" target="_blank" download>{{pdf.name}}</a>
</div>
<div *ngFor="let img of imgSrc">
   <p>_____</p>
   <a [href]="img" alt="testImg" download>{{img.name}}</a>
</div>
```

Fig 74, Bilagan skrivs ut som länk i den färdiga blanketten

7.8 Hantering av SSL-certifikat

Hanteringen av SSL-certifikatet i backend har hittills hanterats på ett sådant sätt att det fungerar men inte på ett såpass bra sätt att det är lätt att underhålla och byta ut. SSL-certifikatet har varit inbäddat i serverns JAR-fil vilket medför att varje gång man vill eller behöver göra någon ändring med certifikatet behöver man bygga om backendservern.

SSL-certifikatet visar viktig information för att verifiera ägaren av en webbplats och kryptering av webbtrafik med SSL/TLS[23], inklusive den offentliga nyckeln (public key), den som utfärdat certifikatet samt de tillhörande underdomänerna.

En lösning på detta problem är att antingen specificera var certifikatet finns med en parameter i körningen av JAR-filen eller genom att specificera en absolut eller en relativ sökväg för certifikatet i serverns application.properties fil. Eftersom att parameter metoden inte fungerade så värst bra för oss valde vi istället att ta den andra metoden med en relativ sökväg vilket man kan se i fig 75.

```
# relative or absolute path to keystore outside jar file
server.ssl.key-store=./co2keystore.p12
```

Fig 75, Inställning för certifikatets sökväg

[23] Transport Layer Security

7.9 Diagramutbyte

Vi hade fungerande grafer redan innan vi började detta arbete men vi har ändå valt att byta ut alla grafer. Vi fick nämligen under den tidigare kursen tips om Apache ECharts[24] från kollegor på tidigare praktikplatser. Utbudet av grafer är mycket större och dessutom finns det mer automatiska funktioner och val för att designa dessa. Därför tyckte vi nu att vi skulle byta ut dom alla nu när vi en gång gjorde om allting. De tidigare graferna var gjorda med ngx-charts[25], som är ett bibliotek utvecklat specifikt för Angular.

Kollar man mängden kod i frontenddelen så blev det inte mindre utan snarare mer kod. Däremot så blev det rejält mycket mindre i backend delen så slår man ihop det så har det blivit mindre. Som tidigare nämnt så finns det mera val för dessa grafer, bl.a. så finns det en s.k. *toolbox* där man kan sätta till exempelvis en datavy, olika diagramtyper man kan byta mellan(*magicType*), en sparfunktion som sparar grafen till en png, mm.

Dessutom sker storleksändringen automatiskt. Tidigare räknades det ut hur stor grafen skulle vara beroende på hur stort fönstret är. Teckenförklaringen(*legend*) som beskriver innehållet i grafen ingår också i denna storleksändring på de nya graferna. I de gamla så inaktiverades den vid en viss storlek då den inte längre fick plats.

Vi valde att hålla ungefär samma design som tidigare då de var tydliga och bra. Färgerna följer även här högskolans grafiska profil. Det finns möjligheter att ha ett mörkt läge, men eftersom vår sida är ljus i övrigt så var inte detta ett alternativ just nu. Det finns även ett *decal pattern* för staplar och delar i pajdiagram. Vi tyckte inte att det passade vår i övrigt enkla och stilrena design så vi skippade även detta.

Det finns alltså mycket fördelar funktionsmässigt med Apache ECharts som vi tyckte vägde tungt. Det finns dessutom en mängd funktioner till, som vi inte valt att aktivera för de diagram vi har. Finns det i framtiden behov av fler grafer eller andra sorter och varianter på

[24] https://echarts.apache.org/en/index.html
[25] https://swimlane.gitbook.io/ngx-charts/

dem så finns det mycket att välja mellan. Konfigurationerna för dessa är väldigt lika, vilket är en stor fördel, så det är nästan bara att kopiera en existerande graf och anpassa den lite.

Följande konfiguration, fig 76, är för en bargraf. Denna kan vi använda i alla olika kategorier för att visa CO_2 för ett års tid. Det enda som skiljer dessa åt är *data* i *legend* och *series* som man ser längst ner i denna konfig.

```
return {
  tooltip: {
    trigger: 'axis',
    axisPointer: {
      type: 'shadow'
    }
  },
  color: ['#3c8a2e', '#44697d', '#d2492a', '#fed100'],
  legend: {
    data: tmp
  },
  toolbox: {
    show: true,
    orient: 'vertical',
    left: 'right',
    top: 'center',
    feature: {
      mark: { show: true },
      dataView: { show: true, readOnly: false },
      saveAsImage: { show: true }
    }
  },
  xAxis: [
    {
      type: 'category',
      axisTick: { show: false },
      data: ["Jan", "Feb", "Mar", "Apr", "Maj", "Jun", "Jul", "Aug", "Sep", "Okt", "Nov", "Dec"]
    }
  ],
  yAxis: [
    {
      type: 'value'
    }
  ],
  series: arr
};
}
```

Fig 76, Konfiguration för en graf

Datat som kommer från backend ska sedan bearbetas som kommer som en *Map*. Denna bearbetning ser lite olika ut beroende på vilken kategori det är men för de flesta ser det ut som följande, fig 77. Vi plockar ut nycklarna från mappen till tmp och detta är namnen i legenden. Sedan tar vi ut rätt data från värdena i mappen mha nyckeln.

```
let tmp = Object.keys(data);
let arr: any = [];

tmp.forEach((item, idx)=> {
    arr.push(
    {
        name: item,
        data: data[item],
        legend: {
            data: tmp
        },
        type: 'bar',
        emphasis: {
              focus: 'series'
        },
        barGap: 0
     }
     );
});
```

Fig 77, Datat som kommer från backend bearbetas för att kunna sättas i grafen

I fig 78 och fig 79 ses skillnaden på sökdiagrammet före och efter ändringen. Det som skiljer sig och är en stor förbättring med det nya diagrammet är att det är scrollbart. Det går alltså att zooma in och ut på både x- och y-axel. Innan syntes det enligt datum men det anpassades inte alls efter hur långt intervall man sökt på och vid stora intervall syntes bara slutet av perioden. Längst till höger i det nya diagrammet syns även toolboxen.

I vår toolbox har vi först en datavy. Datat ändrar till en tabell och det blir som ett kalkylblad som är enkelt att kopiera för redovisning. Det finns även en nedladdningsknapp. Diagrammet blir då sparat som en bild i PNG-format

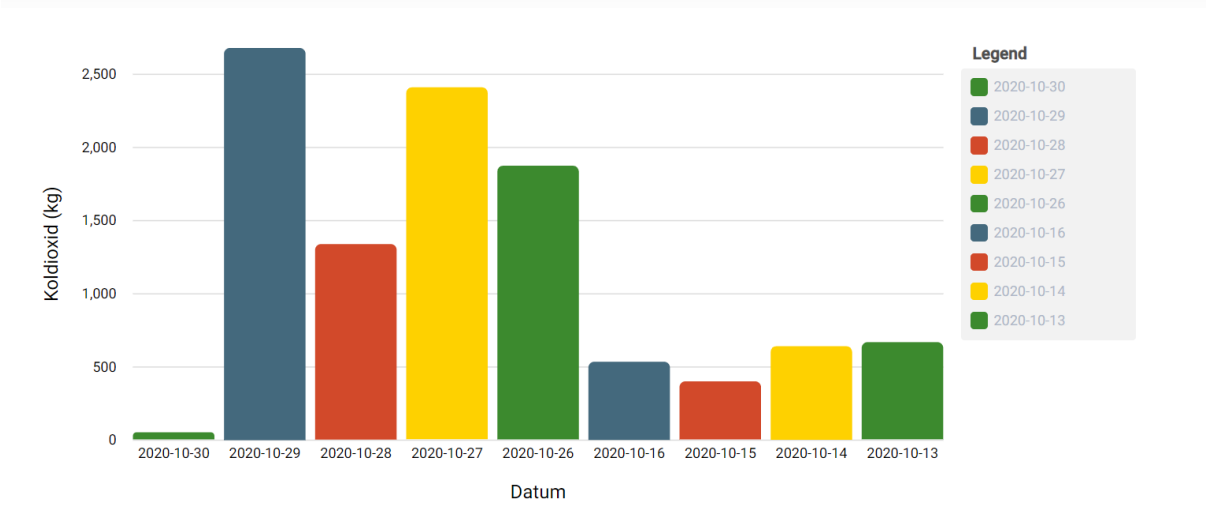


Fig 78, Den gamla grafen för en sökning

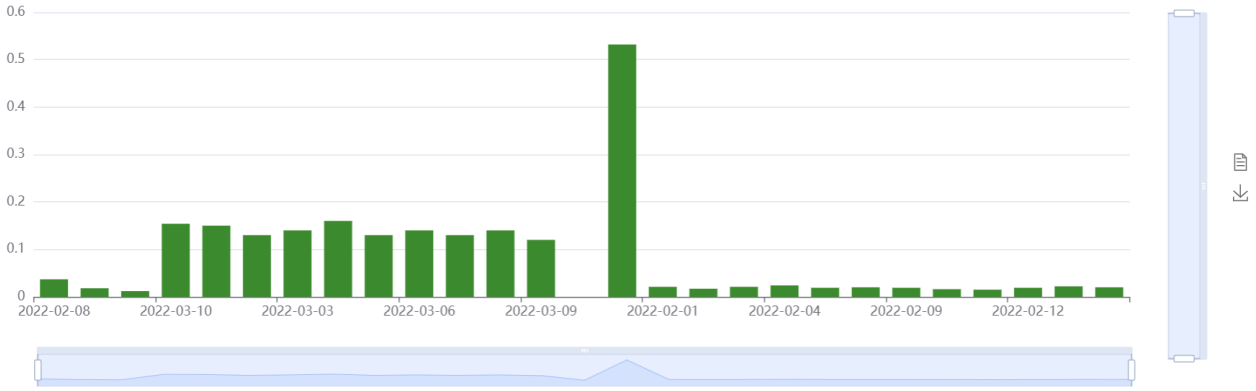


Fig 79, Den nya grafen för en sökning

För visningen av en kategori har det också blivit lite skillnad, figur 80 och 81. Utseendemässigt är det ganska lika men funktionerna är annorlunda. Bl.a. så ser man även i detta nya diagram toolboxen längst till höger. Sedan går det att inaktivera vissa kategorier om man inte vill se dem. Klickar man på “högskolan västra” i teckenförklaringen så kommer alla gröna staplar inaktiveras.

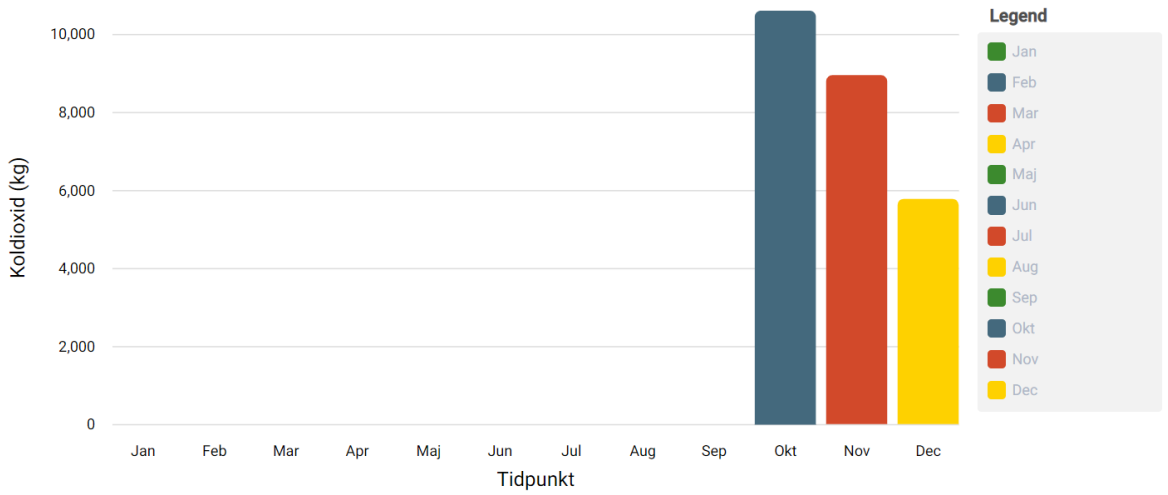


Fig 80, Gamla diagrammet för kategorivisning

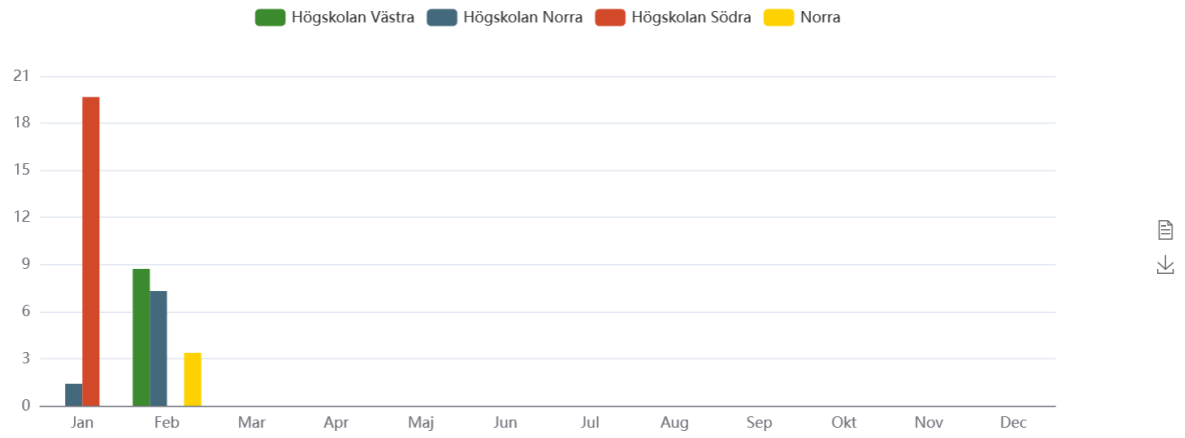


Fig 81, Nya diagrammet för kategorivisning

För att driva de nya graferna behövde vi nu också ändra på hur backend skickar datat till frontend. Före denna ändring skickades datat i listform med egna datatyper som innehåll. Se fig 82 och fig 83 för exempel på hur dessa typer såg ut och hur de användes.

```
@Getter @Setter
public class GraphDataSingle {
    // Ngx charts needs data to be in a specific format;
    // name (label) and value (amount)
    Date name;
    Double value;
}
```

Fig 82, Egen datatyp för grafer

```
public List<GraphDataGroupedList> convertElectricityToGraphMonthList(Integer year){
    List<GraphDataGroupedList> converted = new ArrayList<GraphDataGroupedList>();
    String[] months = {"Jan", "Feb", "Mar", "Apr", "Maj", "Jun", "Jul", "Aug", "Sep",
"Okt", "Nov", "Dec"};
    //Get total co2 data for each month
    for (int i=1; i <= 12; i++) {
        List<GraphDataGrouped> monthData = new ArrayList<GraphDataGrouped>();
        GraphDataGroupedList graphDataGroupedList = new GraphDataGroupedList();
        List<String> properties = this.elRepository.findDistinctProperties();
        for (String property : properties) {
            List<Electricity> el = this.getAllElectricitybyMonthAndYear(i, year,
property);
            double totalCo2 = this.calculateTotalCo2ForList(el);
            GraphDataGrouped graphsingle = new GraphDataGrouped();
            graphsingle.setName(property);
            graphsingle.setValue(totalCo2);
            monthData.add(graphsingle);
        }
        graphDataGroupedList.setName(months[i-1]);
        graphDataGroupedList.setSeries(monthData);
        converted.add(graphDataGroupedList);
    }
    return converted;
}
```

Fig 83, Grafmetod som använder de egna datatyperna

Efter ändringen skickar vi nu en map med strängar som nycklar och listor med flyttal med dubbel precision som dess värden till frontend, denna mappning översätter väldigt bra till en json liknande struktur vilket underlättar hanteringen av datat i frontend. I fig 84 kan man se hur förändringen tas i bruk. Denna ändring medför också en markant reducering i kodlängd.

```
public Map<String, List<Double>> convertDataTest(Integer year) {
    List<String> properties =
this.repository.findDistinctPropertiesByYear(year);
    Map<String,List<Double>> map = new HashMap<>();
    for(String property: properties) {
        List<Double> monthData = new ArrayList<>();
        for (int i = 1; i <=12; i++) {
            List<T> el = this.getAllDataByMonthAndYear(i, year, property);
            double totalCo2 = this.calculateTotalCo2ForList(el);
            monthData.add(totalCo2);
        }
        map.put(property,monthData);
    }
    return map;
}
```

Fig 84, Ny grafmetod som använder en map som lagring

I tabell 3 beskrivs statistiken över antalet rader kod före refaktoreringen, efter refaktoreringen, efter graf bytet samt differensen mellan refaktorering och grafbytet.

Tabell 3 Statistik över grafbytet

	Före refakt	***Efter refakt***	***Efter graf byte***	***Differens***
Rader kod	618	278	98	180
Unika metoder	26	12	7	5

7.10 Parameterdrivna infografer

För att kunna visa CO_2-data på skärmar och kunna anpassa grafen efter vad man vill visa så har vi gjort en komponent för parameterdrivna infografer. Rutten definieras som alla andra rutter utan några extra parametrar eller någonting, fig 85. Parametrar läggs på direkt i sökvägen. När denna sökväg anropas inaktiveras menyraden, för att det ska bli tydligare när det visas på en skärm eller liknande.

```
{ path: 'graph', component:GraphGraphComponent}
```

Fig 85, Sökvägen för grafen defineras

Sedan i typescriptfilen kollas de parametrar vi valt att ha med, man kan skriva in annat än dessa också men dess innehåll kommer inte att kollas. För att få innehållet i dessa parametrar anropas *activatedroute.queryParamMap()*. I subscribemetoden för detta anrop kör vi en GET på parametrarna m.h.a. en nyckel för att variablerna ska få rätt innehåll. Om parametern inte finns med i sökvägen så sätts den till ett default värde, se fig 86 för exempel. Efter detta skapas en titel för diagrammet mha parametrarna och vad de har för värde, som översätts till det språk man angett.

```
this.Activatedroute.queryParamMap
        .subscribe(params => {
          this.category= params.get('category')||"all";
          this.year= params.get('year')|| this.year;
          this.fromDate= params.get('from')||"0";
          this.toDate= params.get('to')||"0";
          this.chartType= params.get('chart')||"bar";
          this.building= params.get('building')||"all";
          this.language = params.get('language')||"sve";
          this.help = params.get('help')||"false";
});
```

Fig 86, Parametrarna plockas ut för att kunna användas i data hämtningen

Det finns även en hjälpparameter. Sätts denna till true fås en vy med lite information. Där anges vilka parametrar som finns, deras default värden samt vilka värden de kan ha. Exempelvis finns endast fyra byggnader just nu och då är det de fyra man kan ange, annars kommer något resultat inte kunna visas.

Parametrarna skickas sedan till en service som i sin tur skickar det till en mapping i backend. *Endpoint* i backend tar emot de sex parametrar som skickas med anropet till denna *endpoint* (se fig 87) och skickar vidare alla parametrar direkt till en metod i en service som behandlar dessa parametrar.

```
@GetMapping("/show/paramgraph")
public ResponseEntity<JSONObject> getParamGraph(
@RequestParam String category, @RequestParam Integer year,
@RequestParam String fromDate, @RequestParam String toDate,
@RequestParam String chartType, @RequestParam String building) {
    try {
        return new ResponseEntity<>(emissionSourceAllService.getParamGraph(
                    category, year, fromDate, toDate, chartType, building),
                    HttpStatus.OK);
    } catch (Exception e) {
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Fig 87, Endpoint för den parameterdrivna grafen

Denna service metod är ansvarig för att hämta rätt data baserat på de parametrar som kommit in, där den mest överordnade förgreningen är ifall “fromDate” eller “toDate” har angetts vid anropet eller inte. Om inget datum har angetts blir grafens data baserat på det år som kommit in med anropet. Om både “fromDate” och “toDate” är giltiga datum baseras grafen på det intervallet, men om bara ett av datumen är giltiga kommer datat att enbart reflektera ett dygns värden.

Nästa nivå är ifall en kategori har angetts eller inte. Ifall ingen kategori har angetts visas data för alla kategorier. Den sista nivån är ifall en byggnad har angetts eller inte. Om ingen har har angetts visas data för alla byggnader eller all data där byggnad är irrelevant.

Baserat på vilken graftyp som angetts skickas det hämtade datat till respektive grafbyggarmetod (se fig 88) som bygger upp ett JSON-objekt som kan skickas tillbaka i retur för anropet.

```
private JSONObject buildChart(String chartType, Map map) {
    return switch (chartType) {
        case "pie" -> GraphData.pieBuilder(map);
        case "line" -> GraphData.lineBuilder(map, false);
        case "bar" -> GraphData.barBuilder(map, false);
        case "stackedbar" -> GraphData.barBuilder(map, true);
        case "stackedline" -> GraphData.lineBuilder(map, true);
        default -> null;
    };
}
```

Fig 88, Metod som anropar grafbyggarmetoder

När det sedan kommer tillbaka från backend är all konfiguration klar. Det enda vi gör mer är att vi översätter teckenförklaringen och x-axelns data till rätt språk.

8. SLUTSATS

8.1 Resultat

Målet med arbetet var att förbättra portalen. Detta gjorde vi genom att vidareutveckla prototypen genom bl.a. refaktorering och ny funktionalitet, vilket också uppfyllts. Alla krav vi hade på vår kravspecifikation har vi lyckats implementera inom tidsramen för arbetet.

Vi har nått målet tack vare att vi systematiskt jobbat oss igenom kraven från det med högst prioritet till det med lägst. Vi har funderat och diskuterat varje krav tillsammans för att hitta en så bra lösning som möjligt.

Vi lyckades väldigt bra med vår uppgift. En del väldigt användbar funktionalitet har tillkommit. Vi fick även ner mängden kodduplicering och filer markant. Det är nu lättare att förstå projektet och det kommer gå snabbare för de som i framtiden ska utveckla denna applikation att sätta sig in i det.

8.2 Reflektioner

Detta har varit ett givande projekt att jobba med och vi har återigen lärt oss nya saker. Det vi kan börja med att konstatera är att grupparbetet funkat bra. Vi har samma ambitionsnivå och mål så det finns inga problem med motivationen att jobba. Uppdelningen har också fungerat bra både när det kommer till implementationen och skrivningen.

När det sedan kommer till implementationen och resultatet av det finns det några saker vi funderat på i efterhand. Dels så kan vi insett att tester borde finnas. Det fanns inte några tester när vi tog över projektet och vi skrev inget nytt för de delar vi implementerade under kursen “programkonstruktion och projekthantering”. Men efter ha gått kursen “testdriven programutveckling” parallellt med detta arbete insåg vi att det borde finnas i vårt arbete också. Men eftersom det inte fanns test från början så valde vi också bort det för de nya delar som impelemterades och det är svårt att skriva test när arbetet är påbörjat/klart eller som vi nu gjort bara refaktorerat koden.

När det kommer till API-dokumentationen så är det samma sak. Det finns en sak som borde gjorts från början och inte när det är klart/den ska refaktoreras. Det är att skriva dokumentationen först, *contract first* som det kallas. Det hade förmodligen gjort dokumentationen lättare att förstå och alla mappningar hade blivit mer genomtänkta när det kommer till namn och uppdelning. Vårt fokus nu har varit att få ner koden så mycket som möjligt men i.o.m det så har en del av förståelsen också försvunnit. Ser man koden så är det tydligt, men om man inte är insatt och bara läser dokumentationen så kommer det sannolikt bli lite otydligt. Så det är också en sak vi tar med oss. Ibland behövs mer kod för att göra det tydligare och lättare att förstå.

Det sista som vi redan nämnt i refaktoreringen av frontend, kapitel 7.1, är användningen av variabeltypen *any*. Den ska helst inte användas då man borde vara så strikt som möjligt med variabeltyper. Men som vi tidigare beskrivit så tyckte vi att det var värt att använda den och få ner kod duplicering i detta fall.

8.3 Framtida utveckling

Portalen kommer nu att tas i bruk och börja användas av främst hållbarhetsgruppen men även övrig personal. I.o.m det kommer det komma fram saker som behöver förtydligas eller göras om lite, men även funktionalitet som saknas och borde läggas till. Högskolan kommer ta fram en förvaltningsplan där det bör ingå hur korrigerings- och utvecklingsförslag samlas in och vem som ansvarar för portalen men också för vidareutveckling.

Vi har under kursen “programkonstruktion och projekthantering” samt detta arbete implementerat det vi tycker är viktigast. Dels kunde det vara en idé att adminanvändare skall kunna godkänna reseförordnanden och reseräkningar via portalen. Det kräver godkännanden från högre instanser och diskussioner hur exakt det godkännandet ska ske men det kunde vara en idé.

En annan idé är att en användare skall kunna spara halvfärdiga blanketter/formulär. Det kanske inte går att fylla i hela på en gång, om det saknas detaljer eller liknande. Då skulle man slippa göra om allt igen utan kan fortsätta där man slutade.

Det sista som vi vet att kommer behöva läggas till/ändras är inläsningen av el/fjärrvärme. Vi har fått reda på att det kommer komma ett API för att läsa detta data och då skulle man kunna byta ut Excelinläsningen och då istället göra det helautomatiskt som det egentligen var tänkt. Detta API beräknas vara klart 2023 och då kan denna ändring ske.

KÄLLFÖRTECKNING

76. Embedded Web Servers. (n.d.). Retrieved March 25, 2022, from https://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/html/howto-embedded-web-servers.html

418 I'm a teapot. (n.d.). Retrieved March 25, 2022, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418

Angular. (n.d.-a). Retrieved March 25, 2022, from https://angular.io/guide/typescript-configuration

Angular. (n.d.-b). Retrieved March 25, 2022, from https://angular.io/guide/what-is-angular

Angular. (n.d.-c). Retrieved March 25, 2022, from https://angular.io/guide/router

Angular. (n.d.-d). Retrieved March 25, 2022, from https://angular.io/cli

Angular. (n.d.-e). Retrieved March 25, 2022, from https://angular.io/guide/dependency-injection

Angular. (n.d.-f). Retrieved March 25, 2022, from https://angular.io/guide/forms-overview

Angular. (n.d.-g). Retrieved March 25, 2022, from https://angular.io/api/forms/FormBuilder

Angular. (n.d.-h). Retrieved March 25, 2022, from https://angular.io/guide/i18n-overview

BabelEdit user interface. (n.d.). Retrieved April 10, 2022, from https://www.codeandweb.com/babeledit/documentation

Bors, M. L. (2018, March 28). *An overview of Angular - Mátyás Lancelot Bors*. Medium. https://medium.com/@mlbors/an-overview-of-angular-3ccd2950648e

Brian. (2022, January 28). *How refactoring works in Java*. CodeGym. https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

Build software better, together. (n.d.). Github. Retrieved May 10, 2022, from https://github.com

Crusoveanu, L. (2016, December 28). *Inversion of Control and Dependency Injection with spring*. Baeldung. https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Documentation - The Basics. (n.d.). Retrieved March 25, 2022, from https://www.typescriptlang.org/docs/handbook/2/basic-types.html

Fain, Y., & Moiseev, A. (2020). *TypeScript Quickly*. Manning.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Generics, inheritance, and subtypes. (n.d.). Retrieved March 9, 2022, from https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html

Hållbar utveckling. (2022, January 14). Högskolan på Åland. https://www.ha.ax/om-hogskolan/hallbar-utveckling/

How to Design a REST API. (2018, April 1). REST API Tutorial. https://restfulapi.net/rest-api-design-tutorial-with-example/

IBM Cloud Education. (n.d.). *What is Java Spring Boot?* Retrieved February 11, 2022, from https://www.ibm.com/cloud/learn/java-spring-boot

JavaScript Tutorial. (n.d.). Retrieved March 25, 2022, from https://www.w3schools.com/js/default.asp

Rådet för yrkeshögskolornas rektorer Arene rf. (2018, May 29). Arene. https://www.arene.fi/radet-for-yrkeshogskolornas-rektorer-arene-rf/

REST API. (n.d.). Retrieved March 29, 2022, from https://www.seobility.net/en/wiki/REST_API

REST Architectural Constraints. (2018, May 8). REST API Tutorial. https://restfulapi.net/rest-architectural-constraints/

Spring Framework - Overview. (n.d.). Retrieved March 9, 2022, from https://www.tutorialspoint.com/spring/spring_overview.htm

Utveckling av CO2-portal för Högskolan på Åland. (2021, April 12). Högskolan på Åland. https://www.ha.ax/forskning-samverkan/projekt/utveckling-av-co2-portal-for-hogskolan-pa-aland/

What is REST. (2018, May 29). REST API Tutorial. https://restfulapi.net/

Why use generics? (n.d.). Retrieved March 9, 2022, from https://docs.oracle.com/javase/tutorial/java/generics/why.html

Wikipedia contributors. (n.d.-a). *Javascript*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Javascript&oldid=50285538

Wikipedia contributors. (n.d.-b). *Lista över HTTP-statuskoder*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Lista_%C3%B6ver_HTTP-statuskoder&oldid=49196026

Wikipedia contributors. (2022a, January 24). *Code refactoring*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Code_refactoring&oldid=1067541688

Wikipedia contributors. (2022b, February 7). *Spring Framework*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Spring_Framework&oldid=1070434162

Wikipedia contributors. (2022c, February 9). *TypeScript*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=TypeScript&oldid=1070886749

Wikipedia contributors. (2022d, February 16). *Unit testing*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Unit_testing&oldid=1072241278

Wildcards and subtyping. (n.d.). Retrieved March 9, 2022, from https://docs.oracle.com/javase/tutorial/java/generics/subtyping.html

BILAGOR

Bilaga 1 - Användarguide

Användarguide CO2 portal Högskolan på Åland

co2.ha.ax

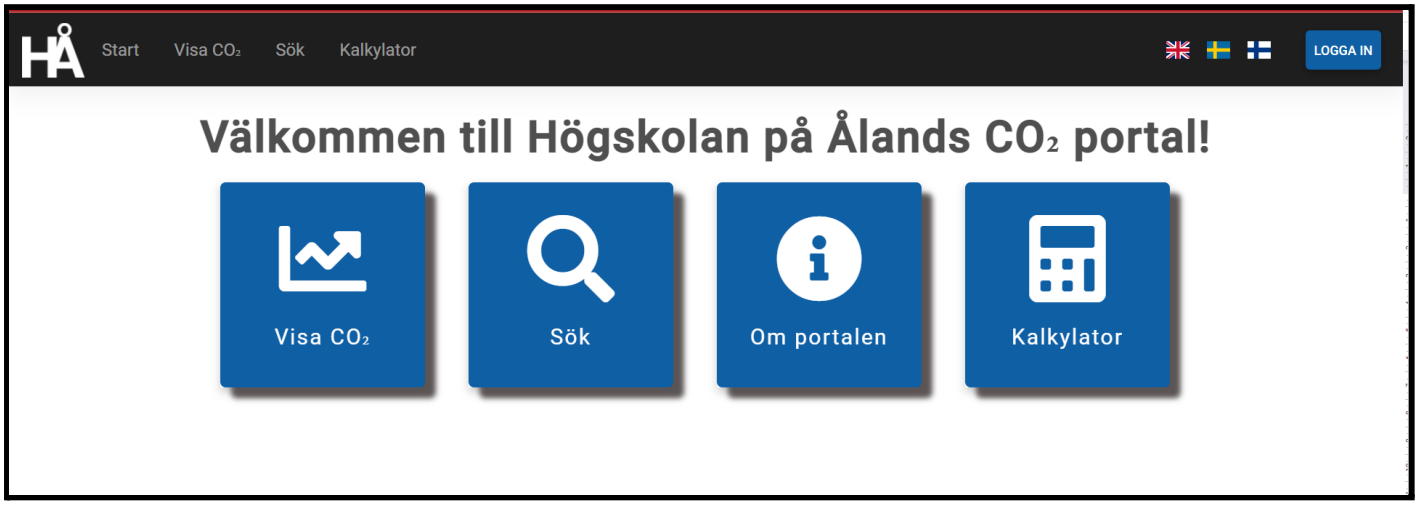
Sofia Södergårdh
Christian Syväluoma

INNEHÅLLSFÖRTECKNING

1. ÖPPET FÖR ALLA

1.1 Logga in

För att logga in på portalen och komma åt de lösenordsskyddade funktionerna så trycker du på den blåa logga in knappen uppe i högra hörnet. Där loggar du in med ditt @ha.ax konto efter att ha tryckt på “sign in with google”.

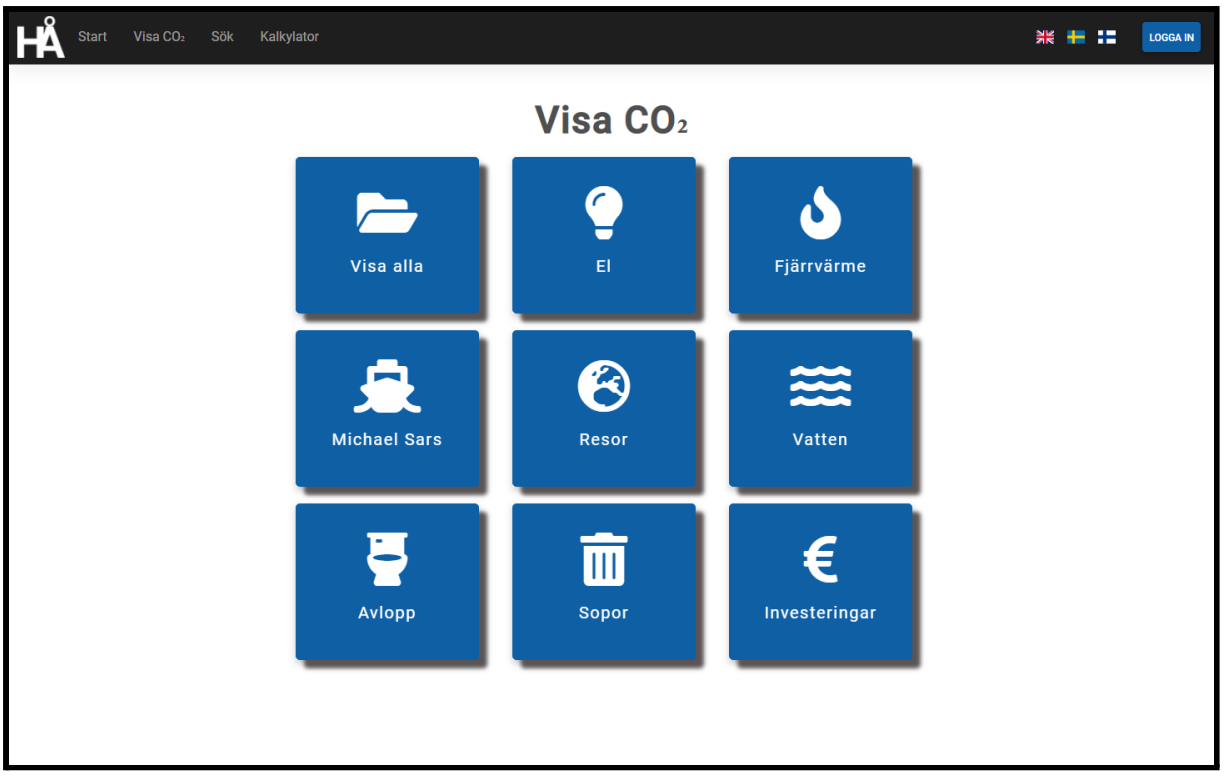


1.2 Visa CO_2-data

1.2.1 Efter år

För att visa koldioxiddata för ett visst år så klickar på på “Visa CO_2” antingen på startsidan eller uppe i menyraden. Du kommer då till en sida med olika kategorier, där kan du antingen välja visa alla kategorier eller välja en specifik kategori.

Du trycker i alla fall på någon av lådorna och du kommer då till en ny sida där du kan välja vilket år du vill visa data från. Sedan trycker du på “visa” och finns det någon information insatt för det året kommer det visas en graf och en tabell.



1.2.2 Eget intervall

För att få fram data för en viss kategori för ett visst intervall så kan man använda sidan “sök”, den hittar man i menyraden och på startsidan. På denna sida kan man välja först vilken kategori/typ man vill söka på, sedan mellan vilka två datum och till sist trycka på “sök”. Den information som finns för den kategorin mellan de datumen kommer då att visas i form av en graf och en tabell.

1.2.3 Parameterdriven graf

För att helt anpassa hur grafen ska se ut så finns det ett alternativ för det också. Dessa grafer styrs av parametrar som du själv bestämmer värdet på. De parametrar man på denna portal kan välja mellan är:

- Kategori
- Språk
- Byggnad
- Graftyp
- Tidsperiod
    - år
    - tidsintervall

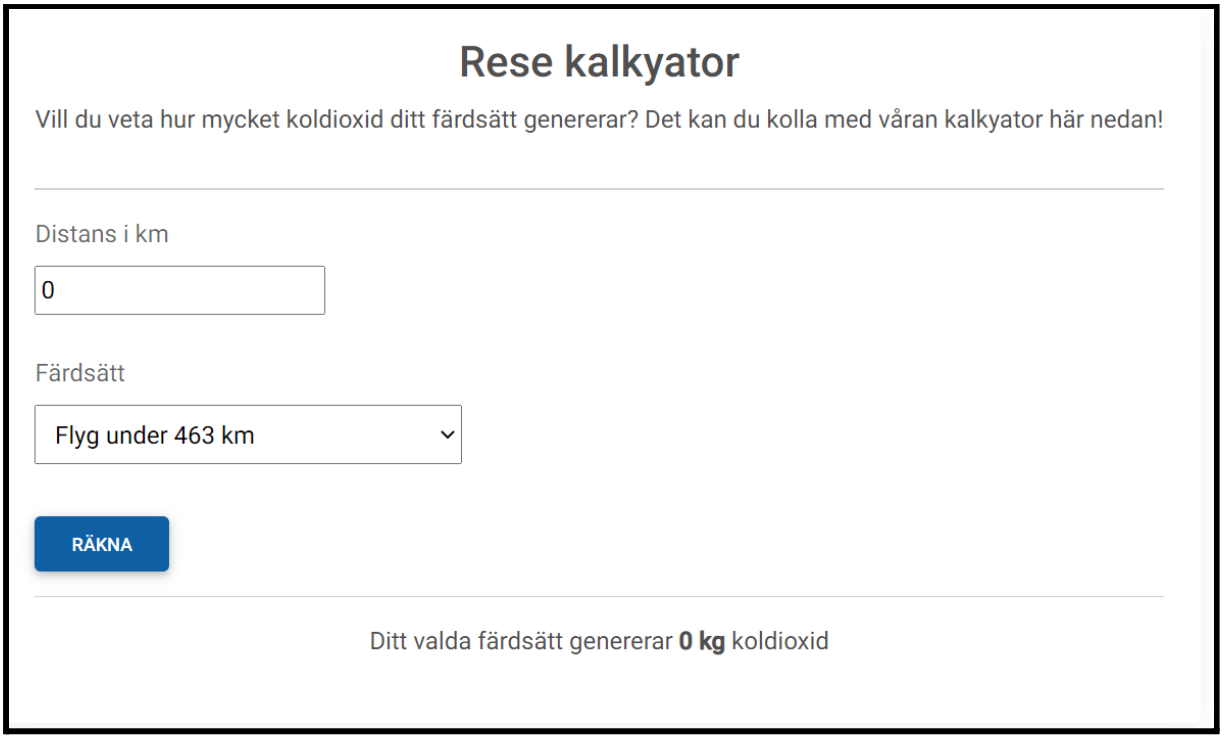
Genom att kombinera dessa kan du få fram vad som helst i princip från datat. Vill man se standardvärdena på dessa samt vilka alternativ det finns för tex kategori samt lite exempel på hur dessa länkar kan se ut så skriver man in:

co2.ha.ax/graph?help=true

Då får man upp en hjälpsida med denna infon.

1.3 Resekalkylator

Har du en resa inplanerad och vill kolla på förhand det miljövänligaste alternativet så kan man använda rese kalkylatorn. Den kommer du åt från både menyraden och startsidan under namnet “kalkylator”. I den kan du välja vilken färdsätt du tänkt använda och hur många km det handlar om. Den räknar då ut hur många kg koldioxid det kommer generera. Du kan då se vilket alternativ som är mest miljövänligt innan du åker.



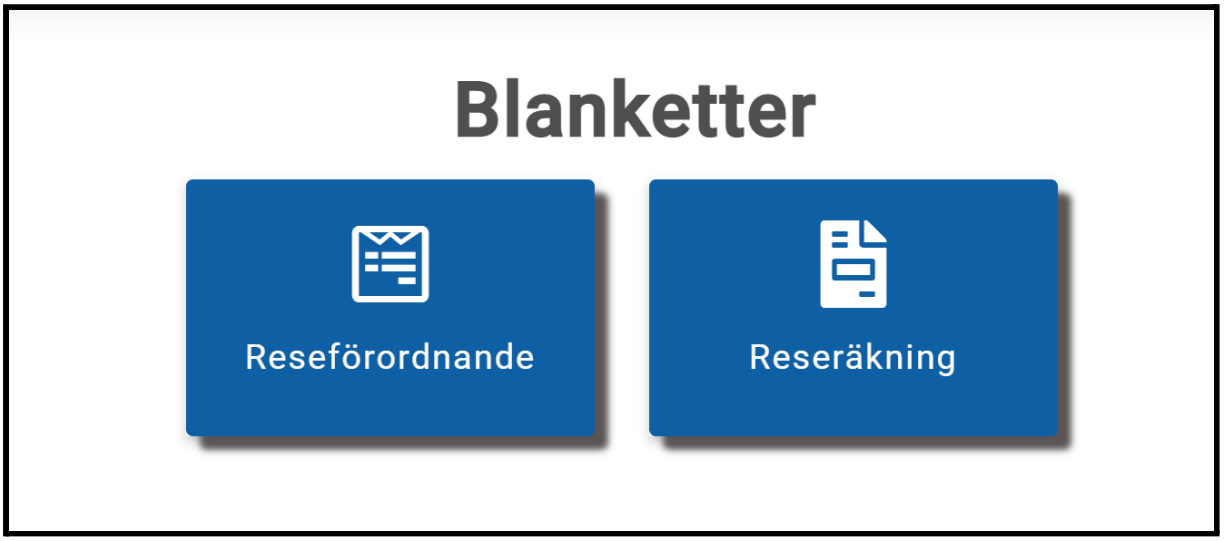
1.4 Byta språk

Portalen är översatt till tre olika språk, svenska är default språk men sedan finns den också på engelska och finska. För att byta språk så trycker man bara på flaggorna uppe i menyraden.

2. LÖSENORDSSKYDDAT

2.1 Blanketter

På sidan finns två olika blanketter, reseförordnande och reseräkning. Reseförordnande görs innan resan och lämnas in för att få godkännande. Reseräkningen görs efter och använder samma info som i reseförordnande. Båda blanketter på sidan är godkända av XXX. Se bilaga A för att se hur många kilometer de vanligaste båtresorna är.



2.1.1 Reseförordnande

Reseförordnandet görs i fyra steg och hittas under fliken “blanketter” på startsidan eller i menyraden.

2.1.1.1 Steg 1

I första steget fyller du i bas infon om resan: anledningen till din resa, avresedatum, destination, uppskattade färdkostnader, uppskattade dagtraktamenten för resan, uppskattade logikostnader.

2.1.1.2 Steg 2

I steg två fylls översiktlig data i. Avresedatumet fylls automatiskt i från föregående steg, ankomstdatum och resekategori måste fyllas i. Efter det ska resans olika delar

fyllas i, du kan lägga till hur många delar du vill och kan även ta bort om det behövs. För en delresa ska avresedatum, avreseort, destinationsort och restyp fyllas i. Väljer du hotellkostnader så fyller du i kostnaden i euro, annars ska antal kilometer fyllas i.

2.1.1.3 Steg 3

I näst sista steget så är det din personliga information som ska fyllas i, den infon hämtas från din profil så har du uppdaterat infon där så kommer den vara rätt i detta steg. Men skulle den vara fel så kan du ändra ditt namn eller din befattning.

2.1.1.4 Steg 4

Sedan på fjärde steget så finns ditt färdiga förordnande. Kontrollera att infon stämmer och tryck på knappen “skriv ut/ladda ner PDF” och välj om du vill skriva ut den eller spara den digitalt. Viktigt är när du får upp utskrifts/pdf fönstret är att gå in under “alternativ” och se till att “sidhuvuden och sidfötter” INTE är ikryssad.

2.1.2 Reseräkning

Reseräkningen görs i fem steg.

2.1.2.1 Steg 1

Första steget är din personliga information, här hämtas din information du angett i din profil men skulle den vara felaktig så kan du ändra/fylla i den i detta steg.

2.1.2.2 Steg 2

I nästa steg så ska du välja resan du vill göra en räkning på. Det du får är alltså en lista på alla olika förordnande du gjort, du ser datum tillsammans med destination och koldioxidutsläpp. Har du inte gjort ett förordnande måste du alltså göra det innan.

2.1.2.3 Steg 3

När du sedan valt resa så kommer du till steget “ändra resa”. Här hämtas infon från ditt reseförordnande, du kan däremot ändra all information och de olika del resorna.

Det du måste fylla i är vem den attesteras av, ett eventuellt projektnummer samt avfärdstid och ankomsttid för de olika del resorna. Du måste även ange om du vill ha betalt för de olika del resorna längst ner på varje del.

2.1.2.4 Steg 4

Näst sista steget är “ersättningskrav”. Information om ersättningsgrunder hittar du på landskapsregeringens hemsida. Här fyller du alltså i ersättnings faktor för privat bil, dagtraktamenten, övriga resekostnader samt erhållna förskott. Här kan du även ladda upp bilagor, tex kvitton från resan. Dessa kommer sedan finnas som länkar längst ner på den färdiga blanketten.

2.1.2.5 Steg 5

Sedan på fjärde steget så finns din färdiga räkning. Kontrollera att infon stämmer och tryck på knappen “skriv ut/ladda ner PDF” och välj om du vill skriva ut den eller spara den digitalt. Viktigt är när du får upp utskrifts/pdf fönstret är att gå in under “alternativ” och se till att “sidhuvuden och sidfötter” INTE är ikryssad.

2.1.3 Ladda ner en blankett igen

Ifall du av någon anledning skulle vilja ladda ner en blankett igen så finns alla dina räkningar som en lista i din profil. Så om du klickar på “profil” via menyraden eller startsidan så hittar du lite längre ner på sidan “mina reseräkningar” där kan du trycka på “välj” för att se den igen och där även ladda ner/skriva ut den.

2.2 Redigera profil

Din personliga info används när du ska göra en blankett. För att du inte ska behöva skriva in denna info varje gång så rekommenderas att uppdatera din profil. När du sedan gör en blankett kommer infon hämtas därifrån.

Din profil ser du under fliken “profil” i menyraden till höger, den ses även på startsidan. Där fyller du i dina uppgifter i de olika fälten, som har tydliga namn för vad

de ska innehålla. Kom ihåg att trycka på “spara” när du är klar. Denna info kan ändras när som helst.

3. ADMIN SKYDDAT

3.1 Registrera co2 data

Som admin användare så kan du lägga till ny data. Detta gör du genom att gå in på “rapportera” genom menyraden eller via startsidan. Där väljer du sedan den kategori som du vill rapportera. De olika kategoriernas rapporter ser sedan lite olika ut med vad man ska fylla i, men alla har ett visst antal fält som måste va ifyllda. Alla fält är uppmärkta med namn så det är bara att läsa och fylla i.

3.2 Uppdatera koefficienter

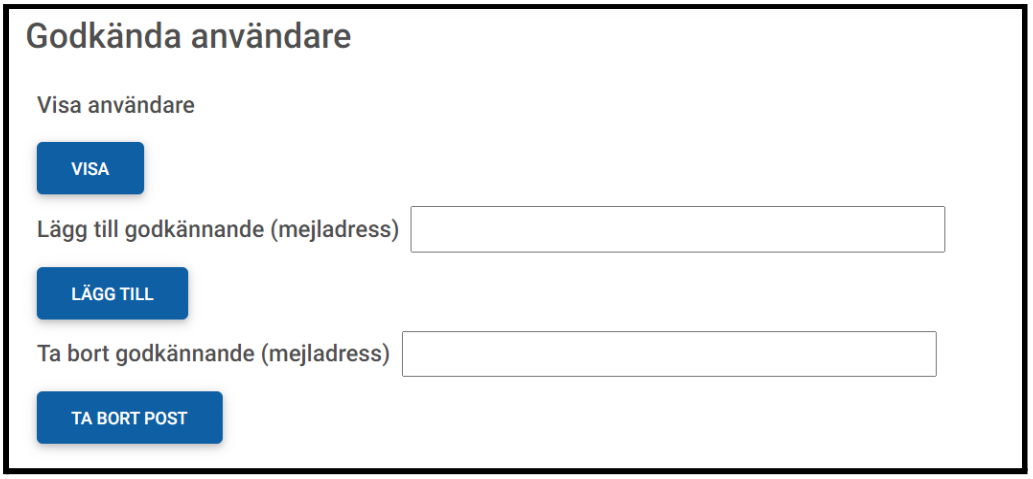
Koefficienterna är det som används när koldioxidmängden räknas ut. Dessa är från början de siffror som leverantörerna för de olika kategorierna till högskolan uppgett.

Man går då in på fliken “administration” som hittas till höger i menyraden. Där kan man välja den kategori man vill uppdatera och sedan skriva in den nya koefficienten. Det blir då den som koldioxidmängden för den kategorin som kommer användas i fortsättningen. VÄLDIGT NOGA att komma ihåg vilken enhet det ska rapporteras i.

3.3 Lägg till/ta bort användare

Behöver någon som inte har ett @ha.ax konto tillgång till portalen eller behöver denne persons rättigheter tas bort? Då kan man lägga till/ta bort de personerna via “administration” fliken, som finns i menyraden till höger.

Där finns överst på sidan två separata fält, den ena för att ta bort och den andra för att lägga till. Skriv in personens mailadress i rutan och tryck sedan på knappen så kommer det verkställas omedelbart.

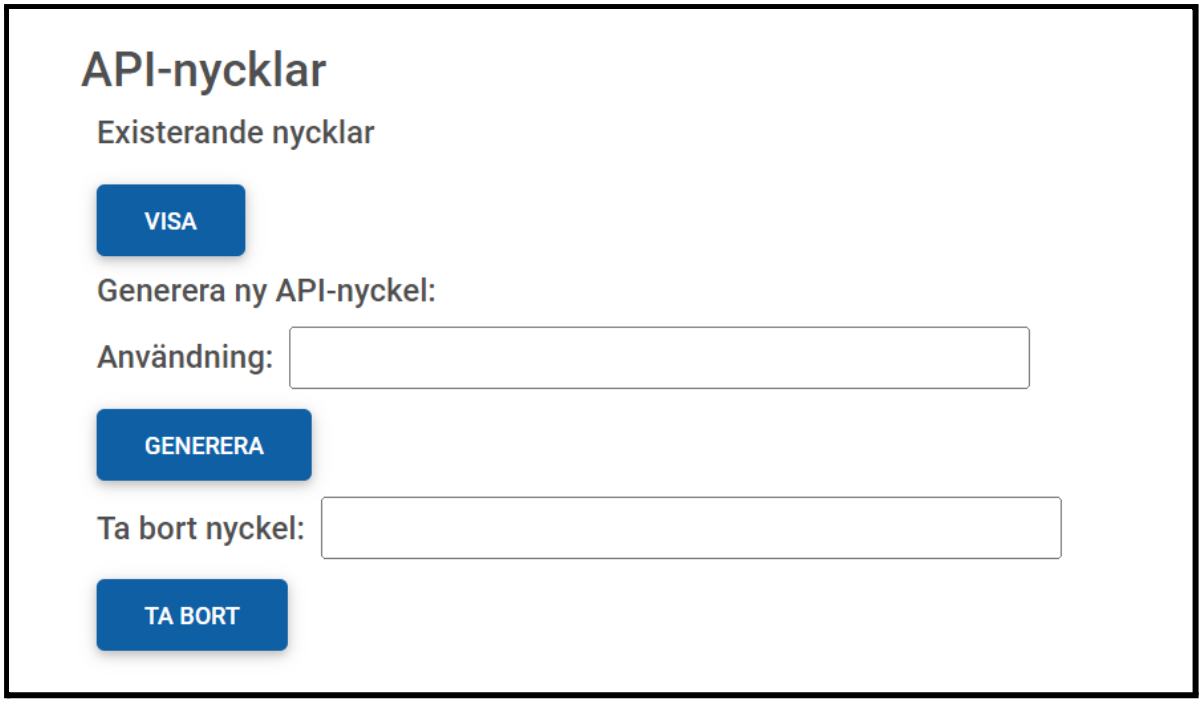


3.4 Skapa/ta bort API nycklar

API nycklar går att hantera genom admin fliken under rubriken “API-nycklar”. Dels så kan man visa de existerande nycklarna men man kan också lägga till nya/ta bort.

Behövs det läggas till en nyckel så skrivs först ett användningsområde in, detta för att det ska vara enklare att se tex vem som har tillgång till den eller till vad den används till ifall problem senare skulle uppstå. När det är inskrivet är det bara att trycka på “generera” så kommer det skapas en random nyckel som sedan synas direkt under knappen.

Vill man ta bort en nyckel så skrivs den in i rutan “ta bort nyckel” så tas den bort, men se till att den inte behövs mer och att ingen använder den innan den tas bort.



3.5 Registrera förbrukning genom excel fil

För el och fjärrvärme kan man istället för att skriva in manuella värden ladda upp en excel fil som läses automatiskt. Denna fil får dock inte se ut hursomhelst utan hämtas från Mariehamns energis portal “min energi”. Denna fil laddas sedan upp i admin fliken längst ner under rubriken “El/värme data:”. I drop downen väljer man vilken byggnad datat tillhör. tryck sedan på “läs fil” så laddad detta upp till databasen indelat på dag.

Bilaga A - Båtresor- kilometer

- Stockholm - Mariehamn ca 140 km
- Kapellskär - Mariehamn : 67,4 km
- Grisslehamn - Berghamn: 44,5 km
- Åbo - Marieham: 195 km
- Långnäs - Åbo: 126 km
- Helsingfors - Mariehamn: fågelvägen 278 km

Bilaga 2 - Instruktioner för produktionssättning

Produktionssättning

co2.ha.ax

Sofia Södergårdh

Christian Syväluoma

Frontend:
kontrollera google login id i app.module.ts

frontend körs via nginx
portar: 443 och 80, 80 redirectar till 443

ssl cert sparas i /etc/ssl/co2

bygg frontend med kommandot: “ng build --configuration production” vilket hamnar i dist mappen
kopiera co2 mappen från dist till servern /home/pop1/CO2/
kopiera sedan co2 mappen till rätt ställe på servern med kommandot:
“sudo cp -r ~/CO2/co2 /var/www/html”
kör sedan “sudo chown -R www-data /var/www/html/co2/”

NGINX - configs:
slå på dessa i /etc/nginx/nginx.conf

```
# Basic settings:
server_tokens off;
# Gzip settings:
gzip on;
gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript;
```

/etc/nginx/sites-available/default

```
server {
	listen 80;
	server_name co2.ha.ax;
	return 301 https://co2.ha.ax$request_uri;
}

server {
	listen 443 ssl;

	root /var/www/html/co2;

	index index.html

	server_name co2.ha.ax;
	ssl_certificate /etc/ssl/co2/wildcard.ha.ax.pem;
```

```
        ssl_certificate_key /etc/ssl/co2/wildcard.ha.ax.key;

        location / {
                try_files $uri $uri/ /index.html =404;
                expires 86400;
                add_header Cache-Control "public, no-transform";
        }

        location /api {
                try_files $uri $uri/ @phpproxy;
                expires 86400;
                add_header Cache-Control "private, no-transform";
        }
        location @phpproxy {
                proxy_pass https://localhost:8080;
                proxy_set_header Host $host;
                proxy_set_header x-Forwarded-For $remote_addr;
                proxy_intercept_errors off;
                recursive_error_pages on;
                error_page 404 = @rewrite_proxy;
        }
        location @rewrite_proxy {
                rewrite ^/(.*)$ /index.html?$1 break;
                proxy_pass https://co2.ha.ax;
                proxy_set_header Host $host;
                proxy_set_header X-Fowarded-For $remote_addr;
        }
}
```

Backend:
i application.properties

- kontrollera att “app.cors.accept.1” pekar på rätt address
- kontrollera databas namn och lösenord samt att den är i state: validate
- se till att raderna under “Server Stuff” inte är utkommenterade
  server.port kan anges men vi använder default 8080
- server logging kan slås på här också
- api dokumentation och swagger ui kan slås på/av under “api docs and swagger ui config”

kör mvn clean package för att skapa en .jar fil i target mappen
kopiera jar filen till /home/pop1/CO2
jar filen körs av en service: “/etc/systemd/system/co2.service”

```
[Unit]
Description=co2
After=syslog.target

[Service]
User=pop1
Group=pop1
ExecStart=/home/pop1/CO2/co2-0.0.1-SNAPSHOT.jar

[Install]
WantedBy=multi-user.target
```

om förändringar görs i servicefilen behöver man starta om demonen med “sudo systemctl daemon-reload”
är det en ny service behöver man köra “sudo systemctl enable co2.service”
vid uppladdning av ny version av jar filen behöver man köra “sudo systemctl restart co2.service”

Scripts:

databasdump

```
#!/bin/bash
#####################################
#
# Backup database dumps.
#
#####################################

# What to backup.
backup_files="/home/pop1/CO2/db_dump"

# Where to backup to.
dest="/home/pop1/CO2/backup"

# Create archive filename.
day=$(date +%A)
hostname=$(hostname -s)
archive_file="$hostname-$day.tgz"

# Print start status message.
echo "Backing up $backup_files to $dest/$archive_file"
date
echo

# Backup the files using tar.
tar czf $dest/$archive_file $backup_files

# Print end status message.
echo
echo "Backup finished"
date

# Long listing of files in $dest to check file sizes.
ls -lh $dest
```

7dgrs roterande backup

```
#!/bin/bash


targetFolder="/home/pop1/CO2/db_dump/"

sqlFile="co2_db_dump.sql"
dataBase="co2"
```

```
user="" #omitted
pw="" #omitted


mysqldump --opt --user=${user} --password=${pw} ${dataBase} >
${targetFolder}${sqlFile}
```

restart av backend

```
#!/bin/bash
systemctl stop co2.service
systemctl start co2.service
```

crontab:

```
0 2 * * 6 /home/pop1/CO2/scripts/restart-co2-service.sh
0 1 * * * /home/pop1/CO2/scripts/dump-co2-db.sh
15 1 * * * /home/pop1/CO2/scripts/backup.sh
```