Generera användargränssnitt baserade på JSON Scheman

Julius Recep Colliander Celik

14 maj 2018

Sammanfattning

Det saknas en bra lösning för att generera intuitiva inmatningsformulär för datamanipulation. Därför måste användargränssnitt skapas utifrån förutbestämda kända faktorer om datan som användargränssnittet ska baseras på. Det här arbetet använder JSON Schema för att beskriva data som kan manipuleras och använder det för att generera användargränssnitt. Genom att använda samma modell kan användargränssnitt genereras generellt utan att i förväg ha information om datan som användargränssnittet ska baseras på.

Abstract

There does not exist a good solution to generate intuitive input fields for data manipulation. Therefore user interfaces has to be created considering predetermined known factors about the data that the user interface will be based upon. This project uses JSON Schema to describe data that can be manipulated and uses that to generate user interfaces. By using the same model unspecific user interfaces can be generated without prior information about the data that the user interface will be based upon.

Innehåll

Kapitel 1

Introduktion

Examensarbetet handlade om att använda JSON Schema för att annotera manipulerbar data, och sedan sedan automatiskt generera ett användargränssnitt för att interagera med datan. Vad jag känner till var detta det första publikt dokumenterade försöket av att generera ett användargränssnitt från JSON Schema, som var implementerat i något annat programmeringsspråk än HTML, CSS eller JavaScript. Detta var relevant då JSON Scheman utvecklats med hänsyn till JSON och JavaScript, samtidigt som många andra språk och miljöer skulle kunna dra nytta av funktionaliteten som JSON Schema kan erbjuda. Det är viktigt att JSON Schema fungerar bra med andra språk än JavaScript för att det ska vara ett bra och användbart verktyg.

1.1 Bakgrund

Det här kapitlet beskriver bakgrunden till varför och i vilka ämnesområden arbetet utfördes, samt för att ge en förståelse för resten av introduktionen. JSON och JSON Schema utgör en stor del av teknikerna som arbetet undersökte. LS Elektronik AB *(LSE)* är företaget arbetet utfördes hos, och Mimer SoftRadio är systemet som arbetet utvecklades mot.

1.1.1 JSON och JSON Schema

JavaScript Object Notation *(JSON)* är ett textbaserat dataformat för att utbyta data mellan webbtjänster. Till skillnad mot andra alternativ, som exempelvis XML, är det både läsbart för människor och datorer, samtidigt som det är väldigt kompakt, vilket är en anledning till att det är ett av de mest populäraste dataformaten för datautbyte mellan webbtjänster [1]. JSON utvecklades med inspiration till språket ECMAScript *(JavaScript)* men samtidigt programmeringsspråksoberoende, vilket lett till att implementationer för att generera och parsa JSON finns tillgängliga i många olika programmeringsspråk [2]. ECMAScript är ett språk som stöds av alla moderna webbläsare, och har därför blivit en kärnteknik för webben.

Trots att JSON är det populäraste dataformatet för datautbyte mellan webbtjänster saknas det ett väletablerat standardiserat ramverk för metadata-definition [1]. En väldigt lovande formell standard är JSON Schema, vilket är ett ramverk som fortfarande utvecklas av Internet Engineering Task Force *(IETF)*. JSON Schema är ett ramverk för att beskriva och annotera JSON-data [3]. Kapitel 2 beskriver JSON och JSON Schema i mer detalj.

1.1.2 Mimer SoftRadio

Arbetet utfördes hos LSE, som är ett tekniskt företag, som utvecklar och tillverkar elektroniska produkter [4]. LSE erbjuder bland annat ett radiosystem som heter Mimer SoftRadio vilket kan användas för att ansluta ett flertal annars inkompatibla radioenheter i ett och samma system, samt fjärrstyra radioenheterna från en persondator med ett klientprogram. I resten av rapporten kan datorn med klientprogrammet kallas operatörsdator, där användaren kan kallas operatör.

Mimer SoftRadio *Mimer* är ett program med väldigt många möjliga inställningar. I många fall är dessa inställningar för komplexa för de vanliga operatörerna att redigera själva, så därför brukar vissa kunder låsa redigeringsmöjligheterna, och bara tillåta vissa administratörer att ställa in alla inställningar på rätt sätt. Det finns också kundfall där flera operatörer använder samma dator, vid olika tidpunkter. Ett förekommande kundfall är att en operatör jobbade dagtid med att leda och organisera dagsarbete, medan en annan operatör tar över nattskiftet för att övervaka många fler radioenheter.

För att förenkla dessa två kundfall påbörjade LSE utvecklingen av funktionalitet som skulle erbjuda användare att spara uppsättningar av inställningar i olika *profiler*. Det skulle gå att enkelt byta mellan flera förinställda konfigurationer av Mimer. För att konfigurera dessa profiler skapades ett administratörsprogram, som skulle kunna fjärrkonfigurera profilerna hos operatörsdatorerna. Fjärrstyrningen skulle underlätta administratörer att ställa in profiler på flera datorer samtidigt, som sannolikt skulle innehålla liknande inställningar.

1.2 Problemområde

Systemet för att konfigurera profiler (se kapitel 1.1.2) bygger på att alla operatörsdatorer exponerar ett API mot en server, över en TCP-port. Administratörsprogrammet skulle erbjuda ett användargränssnitt för att konfigurera profilinställningar, för att sedan kommunicera ändringarna till operatörsdatorerna. I resten av rapporten kan operatörsdatorerna och administratörsprogrammet kallas server respektive klient. Administrationsprogrammet skrevs som en skrivbordsapplikation till Windows, med språket Delphi.

Kommunikationsprotokollet var ett eget skapat protokoll som byggde på att skicka JSON-objekt via TCP. För en beskrivning av JSON, se kapitel 2.1. För en mer utförlig beskrivning av kommunikationsprotokollet se Appendix A. Problemet som LSE hade inför utvecklandet av användarprofilerna var skapandet av ett användargränssnitt på administratörsprogrammet. Olika operatörsdatorer, hos olika kunder, kunde ha olika versioner av Mimer, med olika funktionalitet tillgänglig, och därmed olika uppsättningar konfigurerbara inställningar.

Det vore orimligt kostsamt för LSE att skapa olika administratörsprogram för varje version av Mimer, då både Mimer ändrades med tiden, samt att olika kunder köpte till extra funktionalitet. Samtidigt behövde användargränssnittet på administratörsprogrammet anpassas så att det skulle vara tydligt vad en administratör kunde konfigurera. Helst skulle ett administratörsprogram fungera bra med framtida versioner av Mimer, utan några eller utan stora justeringar av programmet. LSE ville helt enkelt att servern kommunicerade tillgängliga inställningar till klienten, så att klienten sen skulle kunna anpassa sitt användargränssnitt, och det skulle ske på ett så tillräckligt generellt sätt att administratörsprogrammet var framtidssäkert för framtida version av Mimer.

1.3 Problem

Vilka svårigheter finns det med att använda JSON Schema för att automatisk generera ett användargränssnitt, är det möjligt, samt hur generella JSON Scheman går det att använda sig av?

1.4 Syfte

Syftet med tesen är att systematiskt analysera problemen med att försöka skapa automatiskt genererade användargränssnitt utifrån olika JSON Scheman. Målet är att föreslå både en strukturell modell samt en metod för att lösa detta problem.

Syftet med arbetet är att med hjälp av JSON Scheman skapa ett dynamiskt användargränssnitt som anpassade sig efter syfte. Utan att uppdatera administratörsklienten ska samma administratörsklient kunna konfigurera inställningar hos olika datorer med olika versioner av Mimer, och därmed olika uppsättningar konfigurerbara inställningar. Det skulle inte bara innebära stark kompatibilitet utan också framtidssäkerhet hos LSEs produkter.

1.5 Mål

Målet med arbetet är att kunna skapa en grund för användare av Mimer att enkelt kunna konfigurera inställningar, oavsett version eller uppsättning extra funktionalitet. Det skulle kunna innebära att Mimer blir ett bättre verktyg för många potentiella kunder. Samtidigt finns det ett mål med att utforska samt att metodiskt utvärdera och beskriva hur användargränssnitt för att redigera data, automatiskt kan genereras.

1.6 Samhällsnytta, Etik och Hållbarhet

Ur ett samhällsnyttigt och etiskt perspektiv kan en implementation av fjärrstyrda användarprofiler i Mimer SoftRadio innebära effektivisering av samhällsnyttiga funktioner. Mimer användas bland annat av polis, ambulans, brandkår, kollektivtrafik och internationella flygplatser. Fjärrstyrda användarprofiler skulle hos befintliga kunder i många fall innebära effektivare arbete. Som följd av detta går det att argumentera för att det leder till effektivare kommunikation för organisationer som använder Mimer. Då dessa organisationer arbetar med säkerhet i samhället, räddandet av liv, och upprätthållandet av ett effektivt samhälle, lider hela samhället när dessa organisationer inte kan kommunicera ordentligt. Det är därför väldigt etiskt försvarbart att arbeta med att effektivisera arbetet hos dessa organisationer.

1.7 Risker

En ekonomisk risk är risken som alltid finns vid all hantering av data. Om någon datahantering skulle bli fel, och data skulle försvinna, skrivas över eller bli korrupt, så måste kunder tillägna tid åt att återskapa datan. Därför är det viktigt att implementera ett robust system som är delvis feltolerant. Ingen data som kommer hanteras kommer vara kritisk, och kommer vara relativt enkel att återskapa. Problemet blir att det skulle innebära en kostnad att behöva skapa profiler igen och ställa in inställningar igen, och utöver det så skulle korrupta filer till och med kunna innebära att Mimer SoftRadio inte går att använda alls.

1.8 Metodval

Arbetet som ska utföras är till viss del en fallstudie, men samtidigt ska den utforska något nytt och med det föreslå en ny modell. Designorienterad forskning *(Design science research)* är den metod som passar bäst för den här sortens arbete och därför har den metoden valts. Arbetet följde de följande stegen:

1. **Medvetenhet** En beskrivning av problemet som ska lösas med modellen.
2. **Förslag** Förslag på lösning presenteras.
3. **Utveckling** Modellen utvecklas.
4. **Utvärdering** Modellen utvärderas. Lyckades modellen lösa problemet beskrivet i *Medvetenhet*?
5. **Sammanfattning** Dra slutsatser

För att skapa en medvetenhet över hur andra tidigare löst eller försökt lösa liknande problem utvärderades och testades 30 olika implementationer som erbjuder liknande funktionalitet. Det skapade en bra bild över vad som krävdes för att genomföra arbetet. Efter det konstruerades

systemarkitekturen för systemet, så att resten av arbetet kunde planeras utifrån det. De stora praktiska problemenen som skulle lösas blev:

- Hur ska schemat som formuläret baseras på, genereras?
- Hur ska schemat parsas, tolkas och sedan representeras i språket Delphi?
- Hur ska schemats representation tillsammans med data presenteras i ett användargränssnitt?
- Hur ska användarinteraktion integreras i detta system, så att interaktion faktiskt manipulerar data?

1.9 Avgränsningar

En viktig avgränsning är att rapporten endast väldigt ytligt kommer undersöka olika användargränssnitt, och användbarheten hos dem. Arbetet handlar inte primärt om användbarhet, utan arbetet handlar i större grad om hur JSON Schema kan automatisera skapandet av användbara gränssnitt. Med hjälp av kunskapen som arbetet presenterar kan användbara användargränssnitt enklare skapas.

Ett annat ämne som också är viktigt är säkerhet av systemet som skapas. Säkerheten hos applikationen omfattas inte av arbetet, men det ignoreras samtidigt inte. Systemet som utvecklas för att utbyta JSON Scheman och JSON-data sker över en SSL-krypterad säker uppkoppling. Det här arbetet utvärderar inte säkerheten hos den uppkopplingen.

Att skapa ett användarvänligt användargränssnitt utifrån alla möjliga sorters JSON Scheman med samma verktyg omfattas inte av arbetet. Arbetet kommer utforska olika strategier och metoder för att arbeta med förutbestämda JSON Scheman. Dessutom kommer bara en delmängd av JSON Scheman att hanteras, och en färdig JSON Schema parser skapas och testas ej.

Validering av data är något som webbtjänster ofta måste ta hänsyn till. En klient kan annars skicka otillåten data till en webbserver och därför måste webbservern alltid validera data när den tar emot data, innan data används eller lagras. JSON Scheman fungerar utmärkt för validering av data, men då datan valideras hos klienten, både klienten och servern omfattas av arbetet, samt att användarna inte anses ha uppsåt att förstöra eller falsifiera data, kommer JSON Scheman inte användas för att validera data hos servern.

1.10 Disposition

Kapitel 2 presenterar den teoretiska bakgrunden. Kapitel 3 presenterar och utvärderade kända implementationer som berörde liknande ämnesområden som det här arbetet. Kapitel 4 redovisar hur systemet utvecklades, vilka val som togs, varför de valen togs samt diskuterar hur arbetet förhåller sig till tidigare implementationer av liknande projekt och verktyg. Kapitel 5 diskuterar

hur bra JSON Schema är som verktyg för att annotera manipulerbar data, vilka outforskade intresseområden som skulle kunna behöva fortsatt arbete, samt utvärderar arbetets resultat.

Kapitel 2

JSON och JSON Scheman

Det här kapitlet beskriver vad JSON och JSON Scheman är, samt hur de används. Kapitel 2.1 beskriver vad JSON är. Kapitel 2.2 beskriver hur JSON används för kommunikation mellan webbtjänster. Kapitel 2.3 beskriver JSON Scheman, vad de är och hur de är specificerade. Kapitel 2.4 diskuterar användningsområden av JSON Schema samt listar kända implementationer.

2.1 JSON

JSON erbjuder stöd för några enkla datatyper: textsträngar, siffror, tomt värde samt booleska värden, som presenteras i figur 2.1. JSON erbjuder dessutom stöd för två komplexa datatyper vilket är vektorer *(array)*, en ordnad lista av JSON-värden, samt objekt *(object)*, vilket är en oordnad mängd av namn-värde-par *(properties)*. Exempel på de två komplexa datatyperna visas i figur 2.1. Resten av rapporten kommer utbytbart använda JSON-värde, JSON-data, JSON-fil och JSON-dokument för att förklara en av de sex datatyperna som kan representeras med JSON.

Med hjälp av att rekursivt använda *array* eller *object* går det att representera komplexa datastrukturer med hjälp av JSON. Det finns inga begränsningar i hur komplext datastrukturer kan representeras.

Tabell 2.1: De primitiva datatyperna i JSON

Datatyp	Namn i JSON och JavaScript	Exempel
Textsträng	String	`"hej värld"`
Siffra	Number	`4`
Tomt värde	Null	`null`
Booleskt värde	Boolean	`false`

```
[
  "hej värld",
  4,
  null,
  false
]
```

(a) Exempel på JSON-array

```
{
  "firstName": "Erik",
  "lastName": "Andersson",
  "age": 30,
  "human": true
}
```

(b) Exempel på JSON-object

Figur 2.1: De komplexa datatyperna i JSON

2.2 JSON i webbkommunikation

På grund av att JSON är kompakt, enkelt läsbart och har brett stöd hos många språk och implementationer, har JSON blivit väldigt utbrett bland webbtjänster. Figur 2.2 visar ett exempel på en hypotetiskt transaktion av data på webben. En hypotetisk förfrågan till en webbtjänst skulle kunna se ut som i Figur 2.2a, där en klient förfrågar om de nuvarande väderförhållandena i Stockholm i Sverige. Svaret från webbservern skulle kunna se ut som i Figur 2.2b där webbservern svarar att temperaturen är minus tre grader Celsius och att det snöar. Exemplet visar hur simpelt JSON som dataformat är att förstå, vilket delvis skulle kunna vara en förklaring för populariteten.

```
{
  "country": "Sweden",
  "city": "Stockholm"
}
```

(a) Exempel på förfrågan till webbserver

```
{
  "timestamp": "06/01/2018 10:45:08",
  "country": "Sweden",
  "city": "Stockholm",
  "weather": "Snowing",
  "temperature": -3
}
```

(b) Exempel på svar på förfrågan från webbserver

Figur 2.2: Exempel på datatransaktion mellan webbklient och webbserver

2.3 JSON Schema

JSON Schema är ett ramverk för att förklara hur JSON-värden kan se ut. JSON Schema specificerar regler som kan användas för att antingen bestämma om befintliga JSON värden är giltiga, eller för att i förväg beskriva hur giltiga värden får se ut. Objektet i figur 2.1b skulle kunna valideras enligt JSON Schemat i figur 2.3. Den senaste fastslagna versionen *(Draft 7)* av ramverket bygger på tre dokument: *Core*, *Validation* samt *Hyper-Schema*. [3, 5, 6]

2.3.1 JSON Schema Core

JSON Schema Core täcker grunderna för JSON Schema. Dokumentet fastställer exempelvis mediatypen som borde användas för att skicka JSON Scheman över HTTP, förhållandet mellan flera JSON Scheman, samt hur heltal borde behandlas. Att JSON Scheman själva är JSON-dokument bestäms också. Dokumentet fastställer också att validering och annotering av JSON-värden ska ske enligt dokumentet draft-handrews-json-schema-validation-01 *(Validation)*, samt att draft-handrews-json-schema-hyperschema-01 *(Hyper-Schema)* behandlar reglerna kring att beskriva hypertextstrukturen hos JSON-dokument. [3]

2.3.2 JSON Schema Validation

JSON Schema Validation beskriver tre saker: hur man beskriver ett JSON-dokument, hur man ger tips åt användargränsnitt för att jobba med JSON-dokument samt hur man kan beskriva påståenden om ett dokuments validitet. Förenklat beskriver det här dokumentet strukturen hos ett JSON Schema, med beskrivningar av nästan alla nyckelorden. Utöver att beskriva hur JSON-dokument ska valideras, presenteras annoteringsnyckelord som `"title"` och `"description"`, där `"title"` är en kort förklaring för JSON värdet den validerar, och `"description"` är en längre förklaring. [5]

```
{
  "type": "object",
  "required": ["firstName", "age"],
  "properties": {
    "firstName": { "type": "string" },
    "lastName": { "type": "string" },
    "age": { "type": "integer" },
    "human": { "type": "boolean" }
  }
}
```

Figur 2.3: Exempel på simpelt JSON Schema

2.3.3 JSON Schema Hyper-Schema

JSON Schema utvecklas till stor del för användandet av JSON Scheman i webbtjänster. Därför beskriver det tredje dokumentet, JSON Schema Hyper-Schema, hur resurser kan manipuleras och interageras med över hypermediamiljöer som HTTP. JSON Schema Validation skulle kunna beskriva hur ett API anrop ska hanteras och vad som förväntas från förfrågningar och svar på dem. JSON Schema Hyper-Schema kan då användas för att beskriva ett helt API och hur de olika anropen och resurserna är relaterade till varandra. [6]

2.3.4 Kontroversiella flyttal i JSON Schema

JSON Schema stöder två nyckelord för siffror: *number*, och *integer*, vilket motsvarar siffror respektive heltal [7]. Nyckelordet *number* betyder inte nödvändigtvis flyttal då JSON och JavaScript inte skiljer på heltal och flyttal, vilket en betydligt stor andel programmeringsspråk gör som Python, Ruby, C, C++, C#, Java, Delphi med många fler [2, 8–16]. JSON Schema definerar ett heltal som alla siffror med en bråkdel som är lika med noll [7]. Det skulle betyda att både talet 1 och 1.0 skulle tolkas som ett heltal. Många JSON parsers i många språk tolkar 1.0 som ett flyttal vilket gör det svårt att kontrollera om en siffra är ett heltal. Flera implementationer av JSON Schema parsers som exempelvis Python-baserade jsonschema tolkar 1.0 som ett flyttal, trots att JSON Schema specificerar motsatsen [17].

Det finns andra oklarheter kring flyttal, som att det är svårt att validera om ett flyttal är en multipel av ett annat tal, då få språk erbjuder exakt nogranhet för flyttal [18]. Vissa föreslår att flyttal borde hanteras som textsträngar med nyckelordet *format*, men då krävs en större bredd av valideringstermer för att erbjuda samma funktionalitet som det redan finns till datatyper av typen *number* [19, 20].

2.4 Användningsområden för JSON Schema

Användningsområden för JSON Scheman är bland annat:

1. Validering av data.
2. Annotering av data.
3. Beskrivning av REST APIer.
4. Automatisk generering av kompatibel kod, för att hantera JSON värden beskrivna med JSON Schema.
5. Automatisk generering av API-dokumentation.
6. Automatisk generering av användargränsnitt.

Att använda JSON Schema för användningsområdena 1-3 är trivialt. Det går att utveckla program som kan hantera alla oändligt möjliga permutationer av JSON Schema. Det som däremot inte är trivialt är hur användningsområdena 4-6 skulle kunna generaliseras så pass mycket att ett program eller algoritm skulle kunna hantera vilket giltigt JSON Schema som helst. Användningsområde fyra och fem omfattas inte av den här rapporten, och varför användningsområde sex inte är trivialt diskuteras mer i resultatet.

The Json Schema organisation listar kända implementationer på sin hemsida, och har delat upp dem i följande kategorier [21]:

- Validators
- Hyper-Schema
- Schema generation
- Data parsing
- UI generation
- Editors
- Compatibility
- Documentation generation

Arbetet kommer behöva implementera tre av de listade implementationerna: Schema generation, Data parsing samt UI generation, vilket diskuteras i kapitel 2.4.1, 2.4.2 samt 2.4.3.

2.4.1 Generering av scheman

Schemagenerering som kategori består av tolv implementationer där det går att ytterligare dela upp implementationerna i tre kategorier. Det finns implementationer som utgår från JSON data, och genererar ett JSON Schema för att beskriva datan. Det kan användas om det går att anta att all användning av JSON Schemat kommer att användas på data med exakt likadan struktur. Den andra kategorin av implementationer är implementationer som genererar JSON Scheman utifrån kända datatyper i ett statiskt typat språk. Den tredje kategorin av implementation är implementationer som erbjuder en annan metod att beskriva datan, för att sedan översätta det till ett JSON Schema. [21]

Implementationerna som genererar JSON Scheman från JSON data:

- Schema Guru *(Scala)* [22]
- JSON Schema Generator *(Visual Studio)* [23]
- json-schema-generator *(JavaScript / JSON)* [24]

Implementationerna som genererar JSON Scheman från statiska datatyper inbyggda i språket:

- Json.NET Schema *.NET* [25]
- NJsonSchema for .NET *.NET* [26]
- typescript-json-schema *(TypeScript)* [27]
- Typson *(TypeScript)* [28]

Implementationerna som genererar JSON Scheman från andra liknande beskrivningar:

- Liform *(PHP)* [29]
- JSL *(Python)* [24]
- JSONSchema.net *(Online webbverktyg)* [28]
- Schema Guru Web UI **Obs:** Verktyget hittades ej och kommer därför exkluderas från resten av rapporten.
- APIAddIn *(Sparx Enterprise Architect)* [30]

2.4.2 Parsning av JSON Scheman

En parser tolkar JSON Scheman och representerar schemat med någon annan datastruktur. Ofta är parsning viktigt för att schemat ska kunna representeras med en datastruktur som programmeringsspråket är kompatibelt med. Vissa implementationer använder ett färdigt JSON Schema och genererar kod som är kompatibelt med att hantera JSON som är formaterad utifrån schemats struktur. Andra implementationer kan dynamiskt hantera vilket schema som helst under exekvering, och dynamiskt skapa parsers för JSON formaterad utifrån schemat. De parsers som listas på The Json Schema organisations hemsida är följande:

- DJsonSchema *Delphi* [31]
- jsonCodeGen *Groovy* [32]
- aeson-schema *Haskell* [33]
- AutoParse *Ruby* [34]
- json-schema-codegen *Scala* [35]
- Argus *Scala* [36]
- Bric-à-brac *Swift* [37]
- gojsonschema *Golang* **Obs:** Verktyget saknade information på engelska eller svenska och kommer därför exkluderas från resten av rapporten. [38]
- jsonschema *Golang* [39]

2.4.3 Tidigare försök av generering av användargränsnitt baserade på JSON Scheman

Det finns olika implementationer av att generera ett användargränsnitt utifrån ett JSON Schema. Samtliga kända implementationer är skrivna i språket JavaScript och bemöter därför ingen av svårigheterna med att använda JSON eller JSON Schema med andra språk. Samtliga implementationer är implementationer för att generera hemsidor eller komponenter till hemsidor, vilket skiljer sig mycket mot att generera användargränsnitt åt Windows med Delphi, vilket arbetet gjorde.

Vissa av implementationerna används för att generera ett användargränsnitt för att förklara ett API beskrivet med JSON Schema och andra implementationer används för att generera ett formulär för att manipulera data beskrivet av JSON Schema. Att generera ett formulär för att manipulera data beskrivet av JSON Schema är exakt vad den här rapporten utvärderar. Användargränsnittsgenererarna som listas på The Json Schema organisations hemsida är följande:

- Alpaca Forms [40]
- Angular Schema Form [41]
- Angular2 Schema Form [42]
- JSON Editor [43]
- JSON Form [44]
- json-forms [45]
- JSONForms [46]
- Jsonary **OBS!**
- liform-react [47]
- Metawidget [48]
- pure-form webcomponent **Obs not found**
- React JSON Schema Form [49]
- React Schema Form [50]

Kapitel 3

Förarbetet

För att förstå hur systemet skulle utvecklas, och om det redan fanns liknande lösningar utvärderades alla kända implementationer som berörde liknande ämnesområden som det här arbetet. Implementationerna presenterades i kapitel 2.4 och kommer diskuteras mer djupgående i detta kapitel.

3.1 Generering av scheman från JSON

Att generera JSON Scheman från en JSON-fil perfekt är omöjligt. Att bara observera en JSON fil är inte tillräcklig information för att generera ett JSON Schema för i så fall skulle inte JSON Schema behövas. Observera exemplet i figur 3.1. Att skapa ett JSON Schema som beskriver det objektet skulle kunna se ut som i figur 3.2a. Då har antaganden tagits om att det objektet alltid ska ha en *property* som heter *name* och ska innehålla en textsträng. Det skulle kunna vara så att *name* alltid ska innehålla en textsträng som börjar på stor bokstav, vilket är rimligt när det representerar ett namn, och då skulle schemat se ut som i figur 3.2b. Det skulle annars kunna vara så att *name* inte får vara vilken textsträng som helst utan måste vara en av två textsträngar. Det skulle till och med kunna vara så att *name* också skulle kunna vara det booleska värdet *false*. Då skulle schemat se ut som i figur 3.2c.

Alla de här exemplena har enbart behandlat feltolkning av en *property* och inte diskuterat ännu större missförstånd med objektets struktur. Objektet skulle möjligtvis kunna ha en valfri *property* som representerar efternamn. Då skulle schemat se ut som i figur 3.2d. Alla exempel är giltiga scheman till objektet i det givna exemplet. Det finns ingen möjlighet att förstå vilket schema

```
{
  "name": "Julius"
}
```

Figur 3.1: Exempel på enkelt JSON-objekt

```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    }
  }
}
```

(a) Exempel på genererat schema

```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "pattern": "^[A-Z].*$"
    }
  }
}
```

(b) Exempel på genererat schema med strängmönster

```
{
  "type": "object",
  "properties": {
    "name": {
      "enum": ["Julius", "Erik", false]
    }
  }
}
```

(c) Exempel på genererat schema med *enums*

```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    }
  },
  "required": ["name"]
}
```

(d) Exempel på genererat schema med dolda *properties*

Figur 3.2: Olika JSON Scheman som är kompatibla med JSON-objektet i figur 3.1

som faktiskt beskriver datan, utan att göra många antaganden om datan. Därför ansågs samtliga lösningar som genererar JSON Scheman från JSON-filer vara otilräckliga för projektet.

3.2 Generering av scheman från statiska datatyper i statisk typade programmeringsspråk

Det finns implementationer som utnyttjar att statiskt typade programmeringsspråk redan innehåller beskrivningar av data som ska bearbetas. Både *.Net* och *TypeScript* erbjuder programmeringsfunktioner inbyggda i språket för att beskriva komplexa datastrukturer. Att sedan översätta dem till JSON Schema-filer presterar riktigt bra. Varken *.Net* eller *TypeScript* erbjuder stöd för att exempelvis bestämma att ett tal bara får befinna sig inom ett bestämt intervall, och det saknas fler specificiteter som JSON Schema erbjuder. För att bemöta de bristerna använder samtliga implementationer speciellt formaterade kommentarer som kallas annotationer, vilket möjliggör all funktionalitet som JSON Schema erbjuder. Att TypeScript är ett språk som kompileras till JavaScript, vilket JSON och i sin tur JSON Scheman bygger på innebär att översättningen mellan datatyper i TypeScript och JSON Scheman fungerar väldigt smidigt. [25–28]

3.3 Andra genererare av scheman

Liform och JSL erbjuder metoder och funktioner som underlättar dynamiskt skapande av JSON Scheman under exekvering. De erbjuder datatyper som är inbyggda i språket för att definera och hantera komponenter av JSON Scheman. De kan sedan generera ett JSON Schema utifrån de här instanserna av datatyperna. [24, 29]

APIAddIn är ett plugin åt *Sparx Enterprise Architect*, vilket är ett verktyg för modellering, visualisering och design av system, mjukvara, processer eller arkitekturer. Verktyget baseras på UML vilket är ett generellt språk för modellering av system. Det erbjuder delvis en liknande funktion som JSON Schema erbjuder. Om datan som skulle beskrivas av JSON Scheman redan fanns definerade med UML, skulle det möjligtvis vara ett rimligt alternativ att överväga, men då datan inte fanns definerat med UML ansågs APIAddIn vara ett onödigt verktyg, när schemat lika gärna kan defineras med JSON Schema från första början. [30]

Online-verktyget JSONSchema.net är ett verktyg för att skapa JSON Scheman, med hjälp av ett grafiskt användargränsnitt. Det erbjöd inte mer funktionalitet än att skriva JSON Schemat för hand. Dessutom genererade den inkorrekta JSON Scheman. Oktober 17, 2017 släpptes den sjätte versionen av JSON Schema-specifikationen och en ändring mot den tidigare versionen var att *exclusiveMinimum* och *exclusiveMaximum* tidigare var booleska värden men från den sjätte versionen skulle vara siffror. [7] JSONSchema.net påstår att de genererar JSON Scheman utifrån den sjätte eller sjunde versionen men *exclusiveMinimum* och *exclusiveMaximum* är fortfarande booleska värden i deras genererade scheman. [51]

3.4 Parsning av JSON Scheman

En viktig del i all hantering av JSON Scheman är självklart parsningen, vilket är utförandet av att läsa och tolka JSON Schemat, för att sedan representera innehållet på nytt med en annan representation som är mer användbar för syftet. Vissa parsers tolkar ett JSON Schema, och genererar kod för att hantera JSON-filer som är strukturerade efter det schemat. Det kan också användas för att programmet dynamiskt ska förstå hur en JSON-fil ska läsas. Det kan också handla om att parsa ett JSON Schema för att skapa ett dynamiskt test för att testa om given JSON-data är korrekt formaterad, utifrån det givna schemat.

DJsonSchema är ett verktyg som parsar ett JSON Schema och sedan genererar kod som klarar av att parsa JSON som följer samma struktur som schemat. DJsonSchema är skrivet i Delphi och genererar kod för Delphi, vilket är samma språk som konfigurationssystemet skrevs i. Systemet krävde dynamisk parsning av scheman så därför passar inte DJsonSchema för parsning i detta projekt. Utöver det saknades stöd för version sju av JSON Schema, vilket var den senaste versionen av JSON Schema, samt den version som valdes för systemet. DJsonSchema uppger dessutom att implementationen var ofullständig. [31]

jsonCodeGen hade bristfällig dokumentation. Det framstod att det var ett verktyg som parsar en utökad version av JSON Scheman som inte var kompatibel med den senaste officiella, eller någon tidigare officiell JSON Schema specifikation. Dessutom var verktyget skrivet för Groovy, vilket också gör den kompatibel med Java, men inte Delphi. Utifrån beskrivningen av hur verktyget

skulle användas framstod det som att verktyget genererar statiska filer utifrån JSON Scheman eller JSON-filer, vilket inte är dynamiskt som konfigurationssystemet kräver. Det är också värt att nämna att utvecklarna av jsonCodeGen är samma utvecklare som utvecklade DJsonSchema. [32]

aeson-schema är ett verktyg för att validera ett JSON-värde mot ett schema, eller för att generera en JSON parser åt JSON-filer som är strukturerade efter ett givet Schema. Det garanterar att om ett JSON-värde blivit godkänd i validering, kan samma JSON-värde parsas och användas i ett program skrivet i Haskell. Verktyget implementerar version tre av JSON Schema, vilket inte är version sju som projektet använder. [33]

AutoParse är ett verktyg som inte uppdaterats sedan 26 Mars 2013, vilket betyder att den som bäst kan implementera version tre av JSON Schema. Dessutom länkar projektet till en hemsida som länkar tillbaka till projektet, vilket saknar dokumentation. [34]

json-schema-codegen stöder stora delar av JSON Schema men inte allt. Verktyget parsar ett JSON Schema och genererar sedan kod som kan parsa JSON som är strukturerade efter schemat. [35]

Argus är ett verktyg som erbjuder nästan exakt den funktionalitet arbetet skulle implementera, vilket var att dynamiskt parsa JSON Scheman och sedan erbjuda en JSON parser som parsar JSON-filer som har strukturen som är beskriven av schemat. Parsern som parsar JSON-filerna kunde sedan erbjuda representationer av datan i form av datastrukturer som är kompatibla med programmeringssråket programmet är skrivet i. Argus implementerades i Scala, vilket inte är kompatibelt med Delphi. [36] Bric-à-brac är också ett verktyg som liknade Argus i funktionalitet, fast implementerat i språket Swift [37]. En notering är att dokumentationen är motstridig då den påstår att verktyget erbjuder en oföränderlig *immutable* datastruktur, men den i exempelkoden visar motsatsen [37]. Det skulle kunna vara en indikator för att projektet inte är helt felfritt.

gojsonschema saknar dokumentation och hänvisar enbart till en hemsida på kinesiska. På grund av detta så utvärderades inte detta verktyg. [38]

jsonschema är ett verktyg som dynamiskt parsar JSON Scheman och erbjuder en instans som kan användas för att validera JSON-filer. Projektet är skrivet i Golang, vilket inte är kompatibelt med Delphi. [39]

Många av dessa projekt saknar tillräcklig dokumentation, vilket gör det omöjligt att använda dem i ett projekt. Dessutom är parsningen av JSON Scheman relativt enkelt, då det enbart handlar om att parsa JSON-filer som följer en förutbestämd struktur. Dessutom skiljer sig användningsområdena sig starkt efteråt, vilket kan kräva unik anpasssning av JSON Schema-parsern, vilket är en stor anledning till att det finns så många olika sorters parsers. Ett annat problem att ta hänsyn till med implementationerna är att många av dem bara erbjuder generering av statisk kod i förväg, och klarar inte av att tolka olika JSON Scheman, under exekvering av programmet. Det här arbetet handlar om att dynamiskt tolka olika sorters scheman för att anpassa ett användargränsnitt till servern som klienten kommunicerar mot, vilket innebär att de implementationerna är oanvändbara. Det sista och största problemet med implementationerna är att bara en var kompatibel med Delphi, vilket är språket som systemet skrevs i. Med hänvisning till att det är relativs enkelt att skriva en parser och med problematiken alla implemenationer presenterade, skrevs en egen parser åt det här arbetet.

3.5 Användargränssnitt genererade baserat på JSON Scheman

?? Samtliga implementationer som genererar användargränssnitt utifrån JSON Scheman, gör det för hemsidor *(HTML, CSS och JavaScript)*. Att generera ett användargränsnitt, baserat på JSON Scheman, med ett annat språk än JavaScript kan presentera hinder eller svårigheter. Att använda ett webbaserat användargränssnitt var inte ett alternativ för systemet men samtliga implementationer undersöktes, för att ta lärdom om hur de fungerade. Många lösningar separerar JSON Schemat med datan som JSON Schemat beskriver. Den faktiska datan som användargränsnittet visar, och sedan manipulerar kommer också kallas modell i resten av rapporten.

Att använda JSON Schema för att generera användargränssnitt kan ske på olika sätt, och många implementationer bygger på att ytterligare information tillförs för att kunna specificera hur modellerna ska representeras i det grafiska användargränssnittet. Vissa lösningar är generella för användning på vilken hemsida som helst, medan andra är kopplade till att användas med ett visst ramverk, som React, JQuery eller Angular.

Alpaca Forms är en unik lösning då den är den enda implementationen som inte använder ett JSON Schema som är formaterat med JSON, utan schemat är en instans av ett JavaScript-objekt, vilket möjliggör att schemat inkluderar funktionslogik; En funktionalitet som saknas hos JSON. Utöver funktionslogik, inkluderar schemat en *property* som kallas *options*, som används för att beskriva hur användargränssnittet ska se ut. De delarna av schemat som är korrekt JSON Schema används till stor del enbart för att koppla ihop användargränssnittet med datan. [40]

Andra lösningar som utökade det oficiella JSON Schemat är:

- Angular2 Schema Form [42]
- JSON Editor [43]
- json-forms [45]
- JSONForms [46]
- liform-react [47]
- Metawidget [48]
- React JSON Schema Form [49]

En annan lösning till problemet är att använda ett separat JSON-objekt som fungerar som ett schema för formulärets utseende och beteende. Lösningarna som använder den sortens lösning är:

- Angular Schema Form [41]
- Angular2 Schema Form [42]
- JSON Form [44]

- json-forms [45]
- JSONForms [46]
- liform-react [47]
- React JSON Schema Form [49]
- React Schema Form [50]

Observera att vissa implementationerna både utökar JSON Schemat, och lägger till extra scheman för att beskriva formulärets utseende och beteende. Vissa lösningar utökar Schemat för att sedan lägga till ett extra mellansteg där vissa parametrar i modellen kopplas till extra funktionslogik.

Kapitel 4

Arbetet

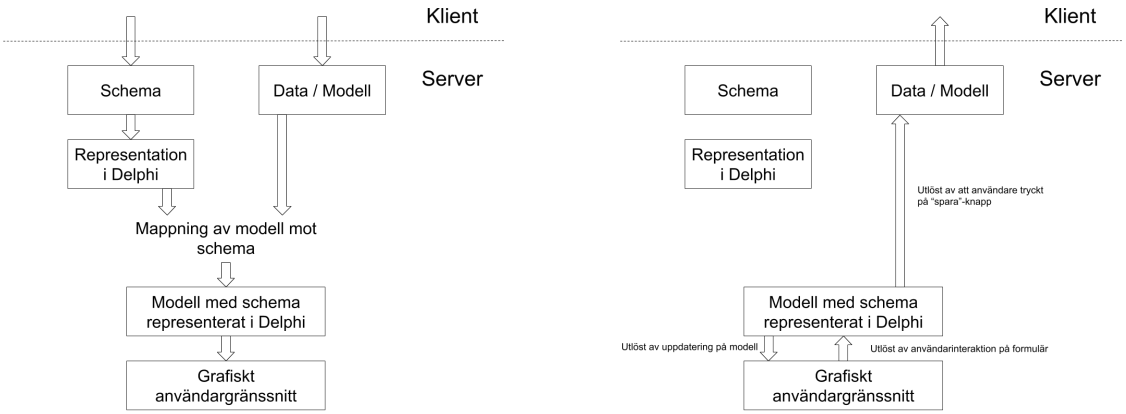
Arbetet kan del

Stort problem är att json och json scheman inte skiljer på heltal och reela tal, medan många språk gör det. Specifikt Delphi som systemet utvecklades i.

4.1 Systemet i helhet

När användargränssnittet ska visas första gången, eller efter att en användare valt att trycka på spara-knappen"(se figur **??**), skickar servern två dokument till klienten. Det dataflödet illustreras i figur 4.1a. Det ena dokumentet är ett JSON Schema som beskriver vilken data användaren med hjälp av klienten ska kunna se och redigera. Det andra dokumentet som skickas till klienten är den faktiska data som är sparad på servern. Först så parsas schemat och representeras i instanser av *records* eller objekt i Delphi. Sedan används datan för att populera objekten med de korrekta värdena, för att tillsammans skapa modellen. Det sista steget är att det grafiska användargränssnittet genereras och presenteras för användaren.

Varje gång användaren interagerar med användargränssnittet och matar in giltig data, uppdateras modellen vilket i sin tur uppdaterar det grafiska användargränssnittet. Det dataflödet illustreras i figur 4.1b Detta sker utan någon kontakt med servern. Om användaren är nöjd med sina ändringar och vill spara dem på servern kan användaren trycka på spara-knappen"(se figur **??**) så uppdateras dataobjektet med de nya värdena från datamodellen, vilket sedan skickas till servern. Efter det börjar systemet igång från början igen.



(a) Flödesschema över systemet (b) Flödesschema över systemet

Figur 4.1: Illustration av dataflöde till och från användargränssnittet

4.2 Generering av JSON Schema i Delphi

Hur JSON Schemat genereras, påverkar inte slutanvändaren, men det påverkar utvecklarna som ska jobba med systemet, och kan hindra vilken funktionalitet som finns tillgänglig i systemet. Ett alternativ är självklart att skriva scheman för hand, men då olika klienter måste ha olika scheman, måste utvecklare se till att rätt schema hamnar på rätt klient vid instalationen av Mimer SoftRadio. Om en kund skulle vilja uppgradera funktionaliteten hos sin installation av Mimer SoftRadio, och därmed behöva tillgång till fler konfigurerbara inställningar i systemet, skulle schemat behöva uppdateras efteråt.

Ett annat alternativ är att basera schemat på inbyggda datatyper i det statiskt typade språket Delphi. Json.NET Schema, NJsonSchema for .NET, typescript-json-schema samt Typson erbjuder exakt den här funktionaliteten i språken .NET respektive TypeScript. För att lägga till extra beskrivningar av datan som språket inte räcker till för, använder .NET-implementationerna *Data Annotation Attributes*, och TypeScript-implementationerna använder experimentella *Decorators*. [25–28] Delphi erbjuder liknande funktionalitet med *Attributes (RTTI)* [52]. Att annotera datastrukturerna som faktiskt sedan kommer användas i ett system kan automatisera väldigt mycket. Det passar jättebra om det handlar om att bygga ett API där JSON Scheman ska användas för att beskriva för klienter hur och vilken data de kan manipulera. Då kan scheman dynamiskt skapas vid exekvering och de är alltid synkroniserade med datan som systemet är konstruerat för att hantera. Problemet med den lösningen är att schemat i det här arbetet inte ska användas för att beskriva vilken data som kan manipuleras på servern, utan det ska användas för att beskriva hur ett användargränsnitt ska se ut. Den data som ska presenteras på användargränsnittet är en delmängd av all tillgänglig data på servern. Det skulle kunna gå att lägga till annoteringar som beskriver vilken data som ska finnas med på schemat, men det finns enklare lösningar.

Alternativet att handskriva JSON Scheman kräver mycket upprätthållning av scheman och är väldigt mottagligt för misstag hos utvecklarna. Det kräver dock inget system för att dynamiskt generera scheman vid exekvering, då de redan skulle vara genererade. Alternativet att konstruera

ett system som använder run-time type information *(RTTI)* är väldigt smidigt för utvecklare under exekvering men kräver relativt mycket utveckling. Ett mellanting skulle vara felsäkert som handskrivna JSON Scheman, men samtidigt dynamiskt genererat vid exekvering för felfria scheman.

JSL är ett exempel på en implementation som erbjuder specifika komplexa datatyper som underlättar genererandet av JSON Scheman. JSL skiljer på datastrukturerna som används för att representera och manipulera data, och själva schemat som används för att beskriva de tidigare nämnda datastrukturerna. [24] Det ansågs vara ett väldigt praktiskt mellanting. Scheman skrivs för hand men de skrivs inte som JSON Scheman. Istället skrivs de som instanser av komplexa datatyper i Delphi, som sedan under exekvering dynamiskt tolkas för att generera ett JSON Schema. Det här tillåter servern att under exekvering utföra logiska bedömningar för att inkludera exakt det som ska inkluderas i schemat, beroende på vilken version av Mimer SoftRadio operatörsdatorn har, och med vilken tillgänglig funktionalitet datorn har. Schemagenerering är frikopplat från databehandling, men är samtidigt dynamiskt genererat utifrån tillgänglig funktionalitet på operatörsdatorn. Ett exempel på ett genererat schema presenteras i figur 4.2.

Att generera scheman från JSON-filer ansågs inte vara praktiskt genomförbart (se kapitel 3.1), och skulle inte kunna upprätthålla kraven på systemet. Det skulle dessutom helt ta bort möjligheten för utvecklare att beskriva användargränssnittet.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "TMimerMainSettings": {
      "title": "Mimer main settings",
      "description": "Mimer main settings",
      "properties": {
        "MyName": {
          "title": "My name",
          "description":
            "This is the name that will be shown. Max length is 10 characters. Name
            ↪  has to be capitalized.",
          "type": "string",
          "maxLength": 10,
          "minLength": 3,
          "pattern": "^[A-Z].*"
        },
        "MyId": {
          "title": "My id",
          "description":
            "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
            ↪  tempor",
          "type": "integer",
          "minimum": 10,
          "maximum": 249
        },
        "FixedSize": {
          "title": "Fixed size",
          "description":
            "If this is turned on, the Mimer SoftRadio window is not resizable",
          "type": "boolean"
        },
        "SendUserInfoIP": {
          "title": "Send user info IP address",
          "description": "Address to send user info to",
          "type": "string",
          "format": "ipv4"
        },
        "FreeMoveEnabled": {
          "title": "Device panels moveable mode",
          "description":
            "This mode decides if the device panels are fixed or moovable.",
          "type": "boolean",
          "enum": [
            false,
            {
              "title": "Fixed",
              "const": true
            }
          ]
        }
      }
    }
  }
}
```

Figur 4.2: Exempel på genererat schema

4.3 Parsningen av JSON Schema i Delphi

Parsningen av JSON Scheman skedde på ett liknande sett som scheman genererades. För att förenkla arbetet skapades en hjälpklass för att hantera scheman i Delphi. Det innehöll olika objekt *(records)* för att representera olika komponenter i JSON Scheman, samt hjälpfunktioner för att generera JSON Scheman utifrån objekten, och parsa JSON Scheman för att sedan få dessa objekt. Ett tillägg till enkla JSON Scheman var att dessa objekt också kunde innehålla värdet av komponenten den innehöll. Utöver hjälpfunktioner för att parsa och generera scheman fanns bland annat också hjälpfunktioner som användes för att bestämma vilken grafisk komponent som skulle användas för att representera datan.

Strukturen av dessa JSON Scheman kunde antas ha en förutsatt struktur, då användningsområdet var väl känt, samt att både klient och server utvecklades med åtanke till varandra och ingen annan tjänst. Alla inställningsfiler som skulle manipuleras av användargränssnittet hade en liknande struktur som figur 4.3. Hela filen är ett objekt, med inställningsgrupper som properties. Figuren har bara en inställningsgrupp men det finns möjlighet för fler. Varje inställningsgrupp är också ett objekt, och den har properties som representerar inställningar. Dessa inställningar är alltid en textsträng, ett heltal, eller ett booleskt värde. Det går därför att utforma parsningen utifrån den här förutbestämda strukturen.

```
{
  "TMimerMainSettings": {
    "MyName": "User1",
    "SendUserInfoIP": "192.168.0.10",
    "MyId": 10,
    "FixedSize": false,
    "FreeMoveEnabled": false
  }
}
```

Figur 4.3: Exempel på genererad inställningsfil

För att både generering och parsning hanterades av samma tjänst ansågs nyckelordet *definitions* och *$ref* vara onödigt. Parsern hade därför inte stöd för de två nyckelorden. Det var också känt att parsern aldrig skulle behöva hantera vektorer *(array)* som datatyp vilket innebar att alla nyckelord relaterade till array kunde ignoreras i implementationen. Utöver det var den grundläggande strukturen väl känd och därför behövdes inte nyckelord som:

- if
- then
- else
- allOf
- anyOf
- oneOf
- not

Det fanns fler nyckelord som inte implementerades då bara en delmängd av JSON Schema. För att flersvarsalternativ skulle kunna erbjudas i användargränssnittet implementerades stöd för nyckelordet *enum*, dock med vissa tillägg vilket diskuteras mer i kapitel 4.4.

Figur 4.1a beskriver systemet i helhet och där är parsningen en viktig del av dataflödet. Först parsas schemat för att skapa en representation av schemat i Delphi, med de tidigare nämnda objekten. Objekten innehåller också värdet av datan de beskriver men i det första steget är alla värden tomma. Nästa steg är att traversera JSON-filen som innehåller den riktiga datan, och sedan populera objekten med korrekta värden. Först då kan användargränssnittet genereras.

4.4 Representation av data i användargänssnittet

Många implementationer diskuterade i **??** använder ett extra JSON-dokument för att beskriva hur schemat, och modellen schemat representerar ska presenteras i användargränssnittet. Det ansågs vara överflödigt att behöva skicka två dokument som båda ska användas vid exakt samma steg i systemet. Det andra alternativet för att utöka funktionalitet hos JSON Schema är att använda egna nyckelord i schemat som inte finns standardiserade i JSON Schema specifikationerna. Det här arbetet krävde nästan ingen utökad funktionalitet hos JSON Scheman, förutom förbättrad funktionalitet hos *enum* vilket diskuteras mer i kapitel 4.4.6. Det gick att generalisera användargränssnittet tillräckligt för att hantera alla datatyper som systemet använder sig av, och presentera ett relevant användargränssnitt av dem. Nyckelordet *format* kunde erbjuda väldigt hög kontroll av användargränssnittet.

4.4.1 Det generella användargränssnittet

Det enklaste valet av gränssnitt skulle vara att erbjuda en generell JSON editor. Det finns många exempel på användargränssnitt som erbjuder ett grundläggande gränssnitt för att redigera

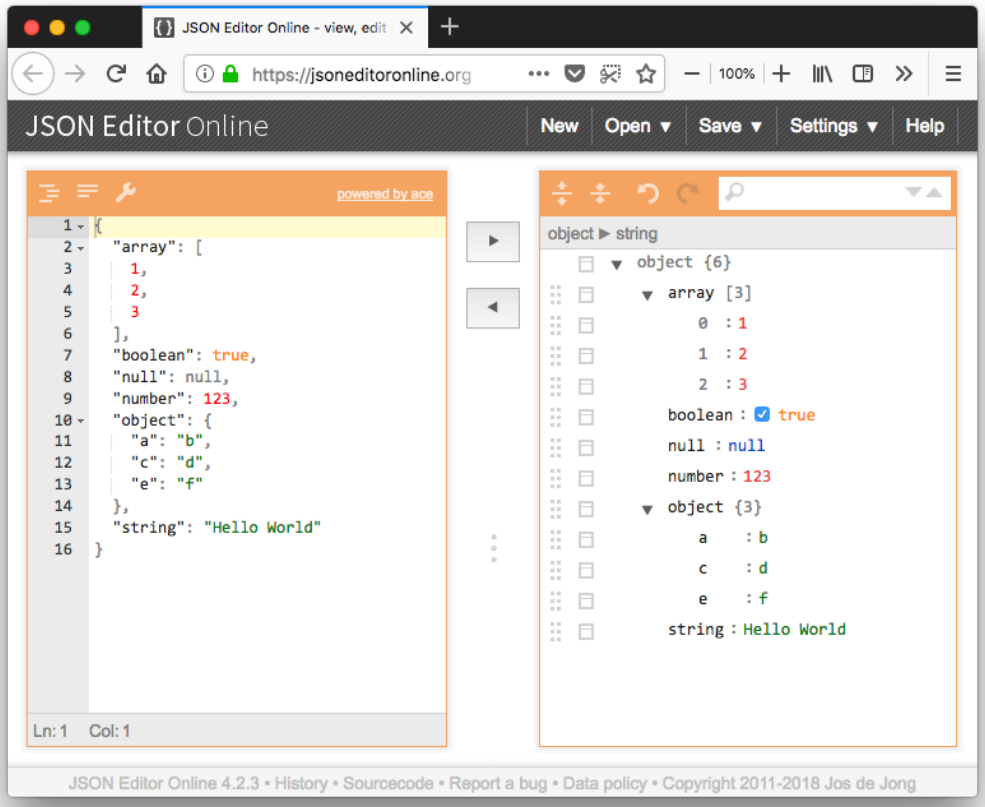
generell JSON-data. De skiljer sig inte mycket mot att skriva JSON manuellt, men kan erbjuda enkel hjälp med nyckelord och formatering. Figur 4.4 visar JSON Editor Online, vilket är ett exempel på ett sådant användargränssnitt [53]. Det skulle behöva begränsa vad användaren skulle kunna mata in med hjälp av JSON Schemat men ett sådant formulär har möjlighet att hantera JSON Data som följer all möjlig sorts struktur. Det vore en stor fördel att kunna erbjuda ett formulär som kan hantera alla möjliga sorters scheman. En nackdel med att använda ett sådant användargränssnitt är att det inte är särskillt anpassat för ändamålet, och gränssnittet blir inte särskillt frikopplat mot datan som ska manipuleras. Det blir dessutom svårt att presentera information till användaren som kan hjälpa användaren förstå hur datan kommer användas.

Ett annat alternativ är att göra som React JSON Schema Form (se figur 4.5). React JSON Schema Form genererar ett webbformulär utifrån ett JSON Schema och ett extra JSON-dokument som kallas UISchema. [49] Användargränssnittet blir mycket mer anpassat till syftet, och lättare att använda. Dessutom utnyttjas annoteringsnyckelorden *title* och *description* från JSON Schemat, för att förklara för användaren vad datan betyder. Det som kan bli svårt med en design som är anpassad efter syftet som ett formulär är hur komplexa JSON-strukturer ska hanteras. Hur skulle exempelvis en vektor innehållande objekt med två properties, varav ena propertyn är ett objekt med två olika vektorer, presenteras på ett användarvänligt sätt? Att även presentera annoteringar som *title* och *description* till alla datanoder i det formuläret vore ännu en till utmaning. En vanlig JSON-Editor klarar av att presentera djupt komplexa datatyper, medan det kan vara väldigt svårt att göra samma sak på ett generellt sätt med formulär. Många lösningar som React JSON Schema Form erbjuder generella formulär som klarar av alla JSON Scheman, med alternativet att specificera mer hur formuläret ska se ut, med hjälp av ett extra JSON-dokument. [49] Det är en riktigt bra kompromiss om lösningen måste kunna klara av att presentera djupt komplexa datatyper, som inte följer en förutbestämd struktur.

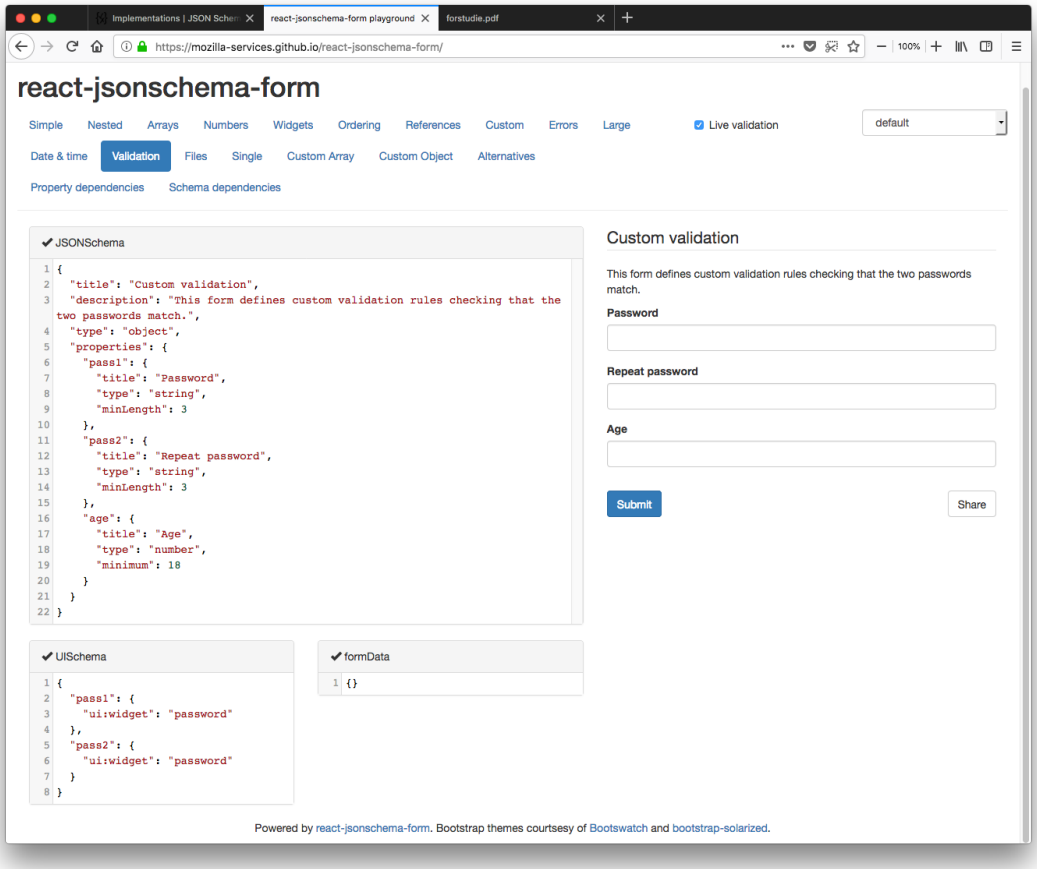
Datan som skulle presenteras på användargränssnittet i arbetet var varken komplex eller totalt generell. Datan bestod av ett objekt med flera objekt som properties, där de objekten i sin tur var antingen en textsträng, ett heltal eller ett booleskt värde. Datan var som mest tre lager djup. Det gick dessutom att dela in datanoderna i inställningsgrupper och inställningsnoder. Därför kunde både en JSON-editor användas, eller ett formulär. Datan som skulle presenteras ansågs av LSE vara för komplex för många användare att förstå utan en ordentlig förklaring av varje inställningsnod. Därför ansågs ett formulär vara det bästa användargränssnittet. Användargränssnittet presenteras i figur 4.6. Då användargränssnittet skulle användas både för data med delvis förutbestämd struktur och inte djupt komplex data ansågs ett extra JSON-dokument för att lägga till extra specificering av användargränsnittet överflödigt och onödigt komplext.

Användargränssnittet som utvecklades blev tvådelat, där ena halvan av gränssnittet användes för att presentera tillgängliga inställningar och inställningsgrupper, samt låta användaren välja en inställning att manipulera. Andra halvan av användargränssnittet användes för att presentera en inställning, med beskrivning av inställningen samt möjligheter att ändra värdet på den valda inställningen. Beslutet grundades på att det skulle vara enkelt för användaren att få en överblick över vilka inställningar som fanns tillgängliga, samtidigt som varje inställning skulle vara enkel att förstå och ändra. Värdena på inställningarna kan alltid synas utan att en inställning väljs men en inställning måste väljas för att kunna bli manipulerad. Namnet på inställningsgrupperna och inställningarna är frikopplade från deras riktiga namn data-filerna och modellen med hjälp av nyckelordet *title* i JSON Schemat.

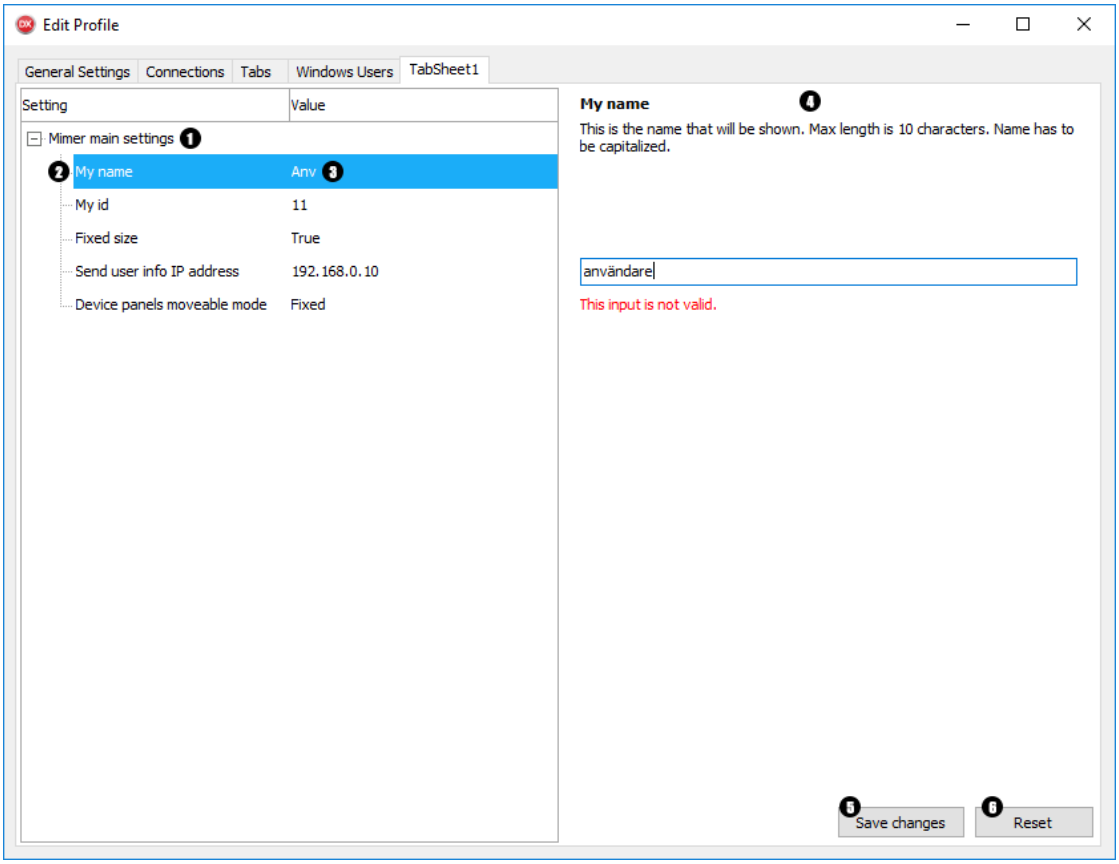
För att förenkla för användaren erbjuds en omstarts-knapp som raderar alla osparade ändringar



Figur 4.4: JSON Editor Online [53]



Figur 4.5: React JSON Schema Form [49]



Figur 4.6: Användargränssnittet av administratörsprogrammet **(TODO! ta ny bild!!!)**

1. Inställningsgrupp
2. Namn på inställning
3. Värde på inställning
4. Formulär för att ställa in inställning
5. Spara-knapp
6. Omstarts-knapp

och startar om formuläret så att formuläret motsvarar inställningarna på servern. Vid varje användarinteraktion, där en användare manipulerar ett värde på formuläret, propageras ändringen till datamodellen vilket i sin tur uppdaterar hela användargränsnittet, men ändringen skickas inte till servern. Användaren har erbjudits en spara-knapp och först då förmedlas alla ändringar till servern.

4.4.2 Hur formuläret förhindrar fel

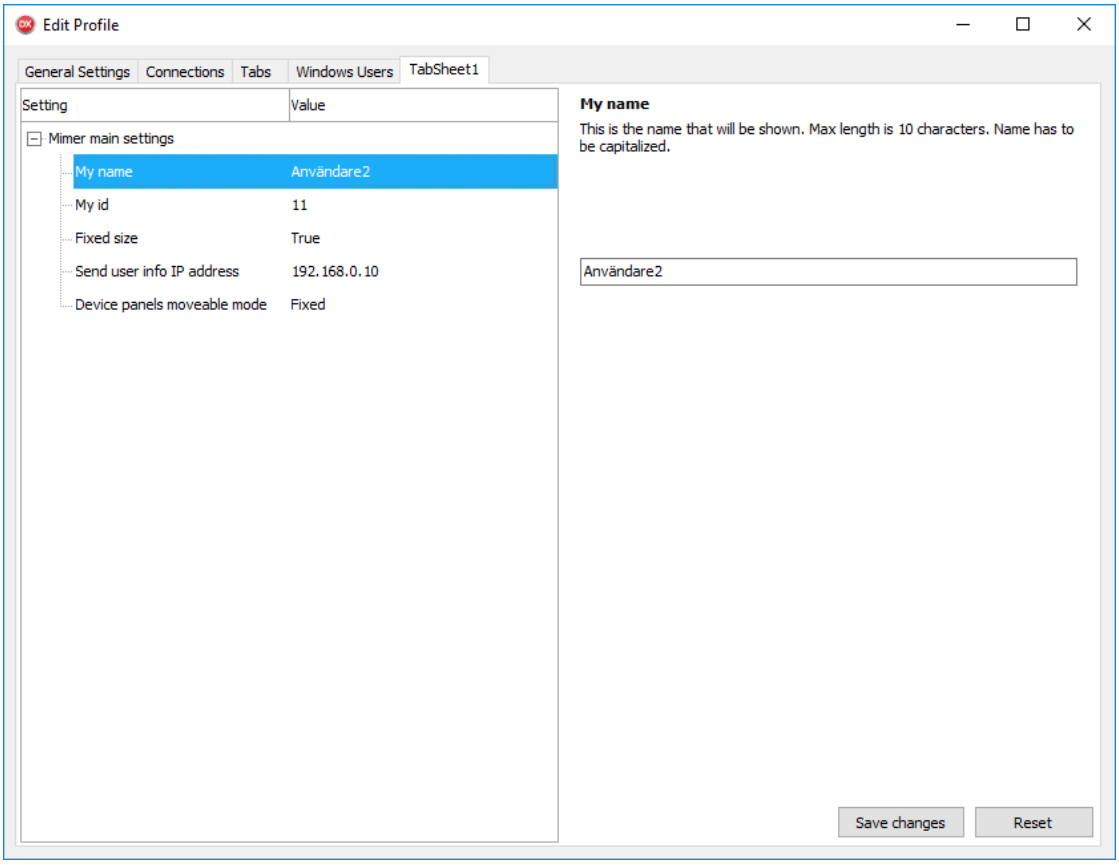
Syftet med JSON-scheman är till stor del att förhindra att inkompatibel data sparas och används i applikationer. Det kan handla om att ett användargränssnitt inte är byggt för att presentera viss sorts data, eller att ett helt system upphör att fungera för att systemet inte kunde hantera den sparade datan. Användningen av JSON Scheman i det här arbetet handlar också om att säkerställa att enbart kompatibel data skickas mellan klient och server. En kritisk frågeställning vid utvecklandet av ett intuitivt användargränsnitt är hur felaktiga användarinteraktioner ska hanteras i användargränssnittet.

Den enklaste hanteringen av inkompatibel data, är att när en användare har givit inkompatibel data, kan ett generiskt felmeddelande presenteras, samt att användaren förhindras från att spara datan före användaren givit kompatibel data. Det förlitar sig på att varje inställning har en tillräckligt utförlig förklaring för hur datan ska formateras. Ett annat alternativ är att förhindra att användaren kan mata in felaktig data. Det kan exempelvis handla om att användaren bara kan mata in siffror till formuläret, om datan ska formateras som en siffra. Om användaren försöker mata in en bokstav så ignoreras den användarinteraktionen. Det kan kompleteras med förklaringar av korrekt formatering i inställningsbeskrivningen, men att användaren inte ens kan mata in inkompatibel data är väldigt intuitivt. Det perfekta målet vore att när en användare matar in inkompatibel data, ändrar formuläret datan till den närmsta kompatibla datan. Om en användare matar in siffran 300 i ett formulär som bara tillåter siffror upp till siffran 255 så kan formuläret automatiskt ändra värdet till 255. Det kan dock vara svårt att lösa för mer komplexa situationer.

För att erbjuda en grundlägande användbarhet av formuläret presenterades olika sorters inmatningsfält beroende på om typen av inställning var en textsträng, en siffra eller ett booleskt värde. Detta förhindrade att användare kunde mata in data som var av fel typ. Det fanns fler fall där ännu fler olika inmatningsfällt kunde presenteras beroende på mer formateringsinformation av datan. När det gick så skapades formuläret så att den aldrig tillät användaren att mata in inkompatibel data, men i vissa fall visades bara ett felmeddelande. Hur olika formateringar specificerat i schemat upprätthölls beskrivs i resten av kapitlet.

4.4.3 Textsträngar och *format* i användargränssnittet

När en vald inställning ska vara formaterad som en sträng, presenteras ett inmatningsfält för strängar, som i figur 4.7. JSON Schema erbjuder tre valideringsnyckelord enbart för textsträngar: *maxLength*, *minLength* samt *pattern*. Nyckelorden *maxLength* och *minLength* används för att bestämma längden av textsträngen. Nyckelordet *pattern* bestämmer att textsträngen måste följa ett mönster, som specificeras som ett regular expression enligt ECMA 262 regular expression

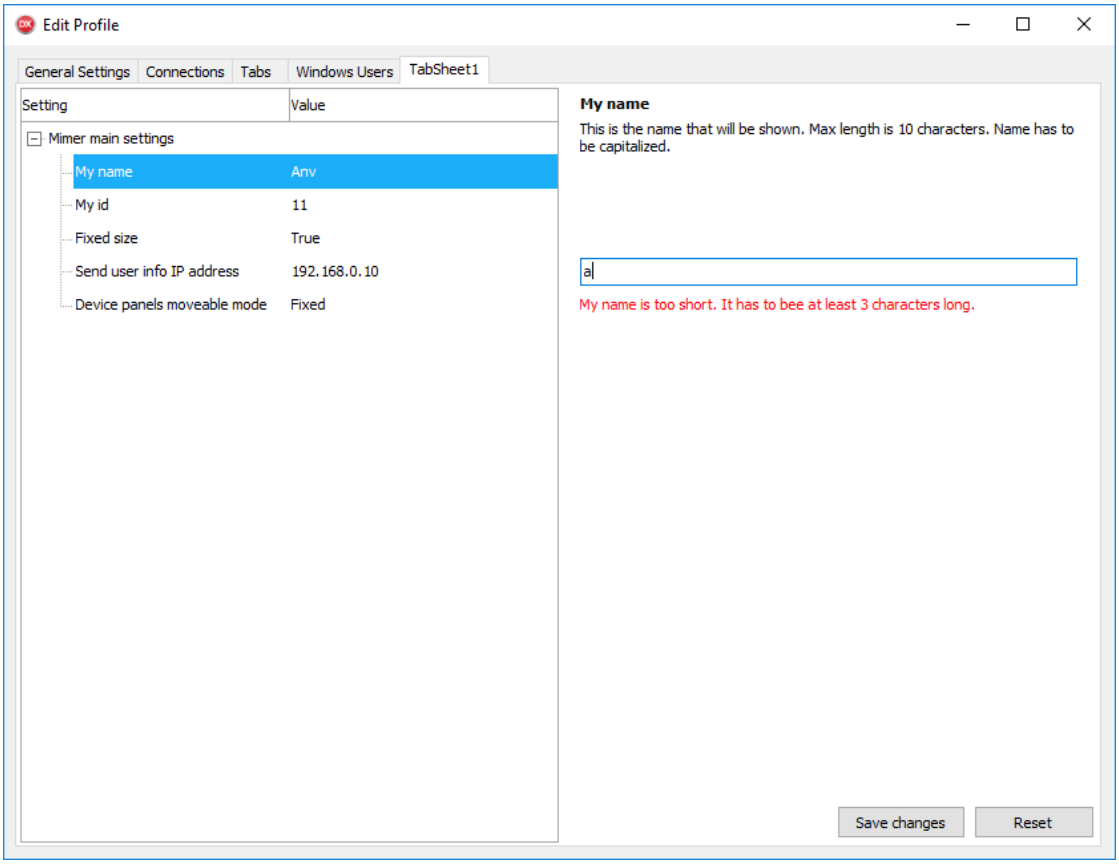


Figur 4.7: Inmatningsfält för textsträngar

dialect [16]. Utöver de tre nyckelorden finns det ett fjärde nyckelord som är applicerbart på textsträngar, vilket är *format*. I de senaste JSON Schema specifikationerna är samtliga åtta definerade format enbart applicerbara på textsträngar. Det går att applicera nyckelordet *format* på vilken datatyp som helst, vilket kort kommer diskuteras senare i rapporten. [7]

Om en användare försöker mata in en karaktär i inmatningsfältet, och textsträngen redan är lika lång som den maximala tillåtna längden, ignoreras användarinteraktionen. På det sättet går det inte ens för en användare att mata in inkompatibel data, och försöka förstå hur korrekt formaterad data ska se ut. Detta bör kompleteras med en förklaring i beskrivningen av inställningen, så att användaren förstår varför formuläret hanterar användarinteraktioner på det sättet.

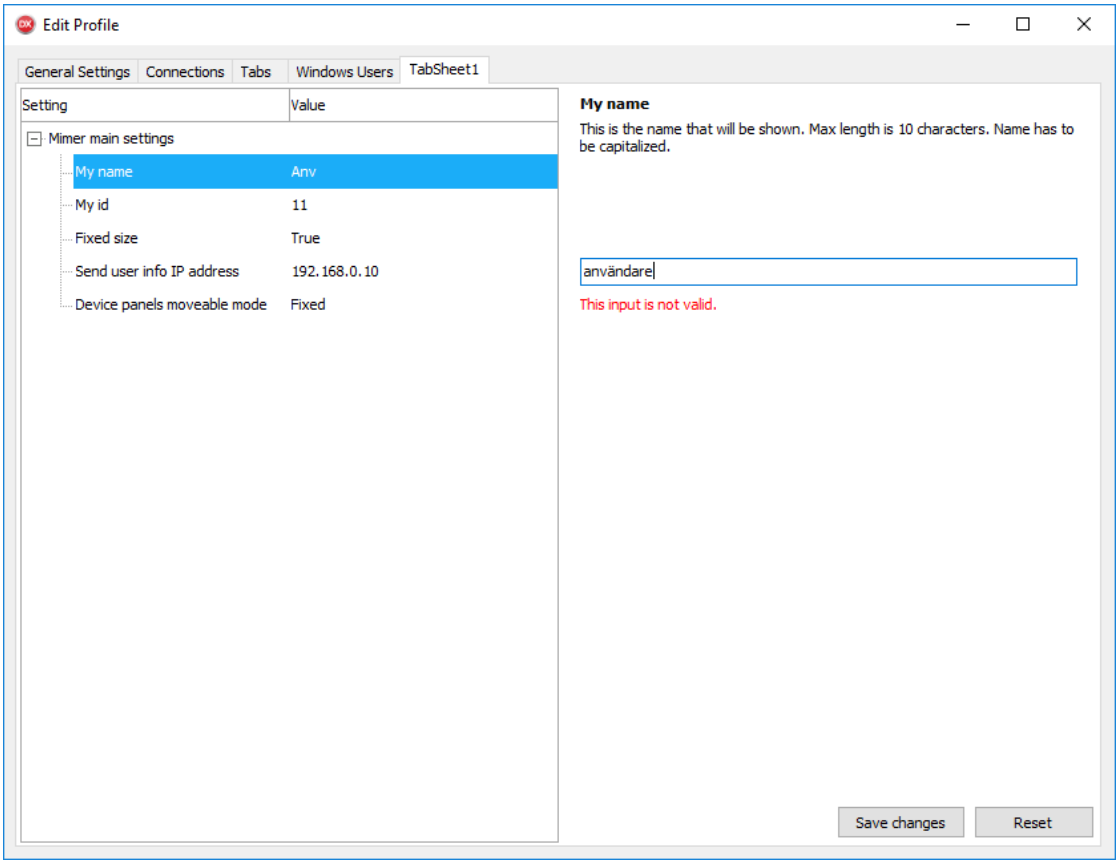
Om en användare försöker ta bort så många karraktärer så att textsträngens längd är kortare än den minsta tillåtna längden, presenteras ett felmeddelande och den senaste tillåtna datan sparas i modellen (se figur 4.8). Det skulle kunna gå att förhindra användaren från att ta bort så många karraktärer, men det skulle kunna upplevas som omständigt av användaren, då det kan finnas fall då användaren vill ta bort hela textsträngen för att byta ut textsträngen mot en helt ny. Om användaren skulle förhindras från att ta bort för många karraktärer, och användaren skulle vilja byta ut hela textsträngen mot en helt annan, skulle användaren behöva mata in den nya samtidigt som den gamla delvis var kvar, vilket inte ansågs vara användarvänligt. Att kombinera



Figur 4.8: Felmeddelande vid för kort textsträng

det med en maximal längd på textsträngar skulle tvinga användaren att hela tiden förhålla sig inom ett intervall av textsträngslängd.

Om formatet av textsträngen är mer komplext än längd kan *pattern* användas för att specificera väldigt komplexa mönster som en textsträng måste upprätthålla. Det kan exempelvis användas för att specificera att en textsträng ska inledas med versal som i figur 4.9. Då regular expression-mönster kan vara väldigt komplexa och oförutsägbara, är det svårt att hitta ett närmsta kompatiblet värde. Därför presenteras ett felmeddelande när textsträngen inte följer mönstret. Då det inte går att veta hur mönstret fungerar på ett användarvänligt sätt, används ett generiskt felmeddelande, vilket innebär att det är viktigt att inställningsbeskrivningen beskriver formatet som textsträngen ska följa.



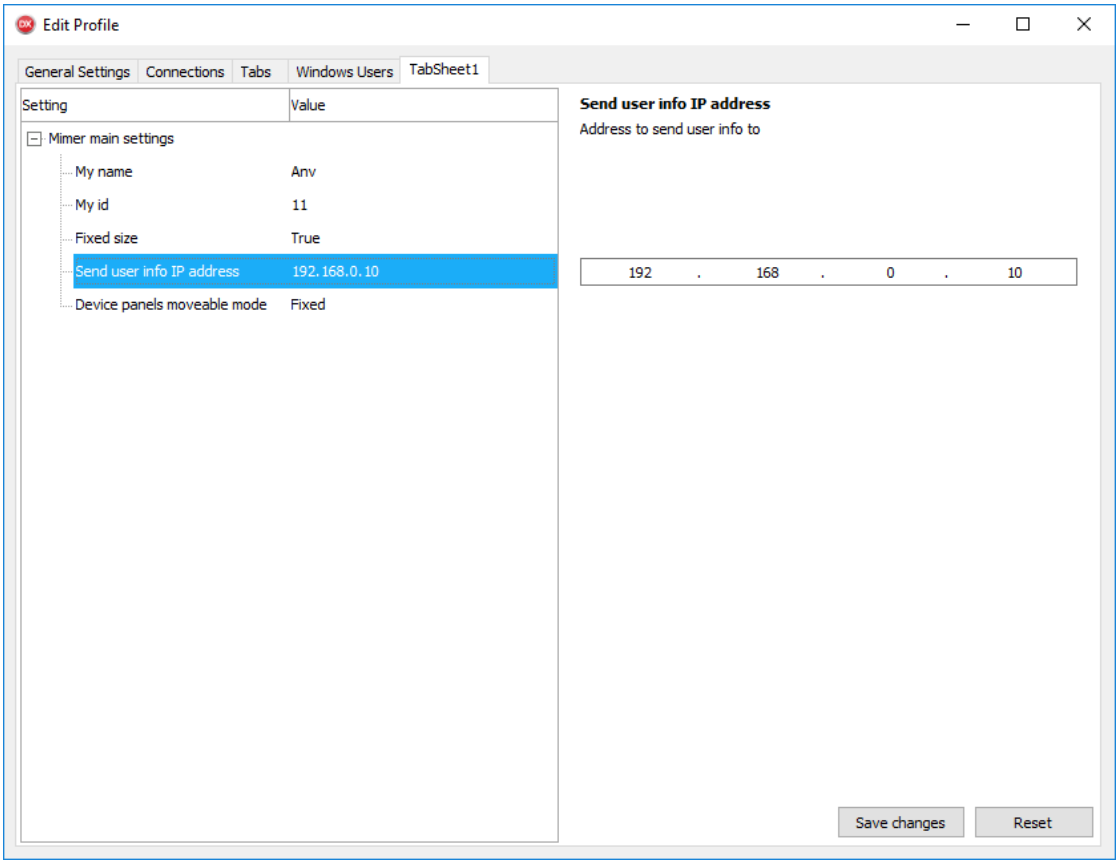
Figur 4.9: Felmeddelande vid för textsträng som inte följer regular expression-mönstret

4.4.4 Nyckelordet *format*

Nyckelordet *pattern* är jättebra för att säkerställa att textsträngar följer ett mer komplext format än enbart längden på textsträngen, men *pattern* kan inte användas för att bestämma hur inmatningsfältet ska se ut. Ibland skulle det passa bättre att presentera ett inmatningsfält som är mer lämpat för en specifik textsträng. Det kan exempelvis passa med ett litet inmatningsfält för korta textsträngar som bara ska vara ett eller några ord, men vid längre texter kanske ett större inmatingsfält behövs. Nyckelordet *format* används både för annotering och validering enligt JSON Schema specifikationerna och får användas för att beskriva egna format på JSON-värden, som inte nödvändigtvis måste följa en standard upprättad av IETF, men som måste vara förutbestämd av alla parter i samma system [5].

För validering kan *format* användas när nyckelordet *pattern* inte räcker, eller blir komplext att använda, och för annotering kan det användas för att förklara formatet av ett JSON-värde. Det kan också användas för att annotera för ett användargränssnitt, att ett JSON-värde borde presenteras på ett specifikt sätt. Figur 4.10 visar hur *format* används för att beskriva att en textsträng ska vara formaterad som en IPv4-adress. I det fallet kan ett specifikt inmatningsfält erbjudas, som bara tillåter användaren att mata in en IPv4-formaterad textsträng.

Nyckelordet *format* kan användas för andra typer än textsträngar och systemet som utvecklas har stöd både på klienten och servern, för att använda *format* till siffror och booleska värden. När *format* används till textsträngar är det viktigt att poängtera att det borde kompleteras med *pattern* om möjligt. Alla delar av systemet ska vara bakåtkompatibla med tidigare versioner av systemet, för en betydligt lång tidsperiod, och om nya sorters *format* läggs till, kan vissa äldre instanser av klienter vara inkompatibla med den sortens *format*.



Figur 4.10: Textsträng formaterad som en IPv4-adress

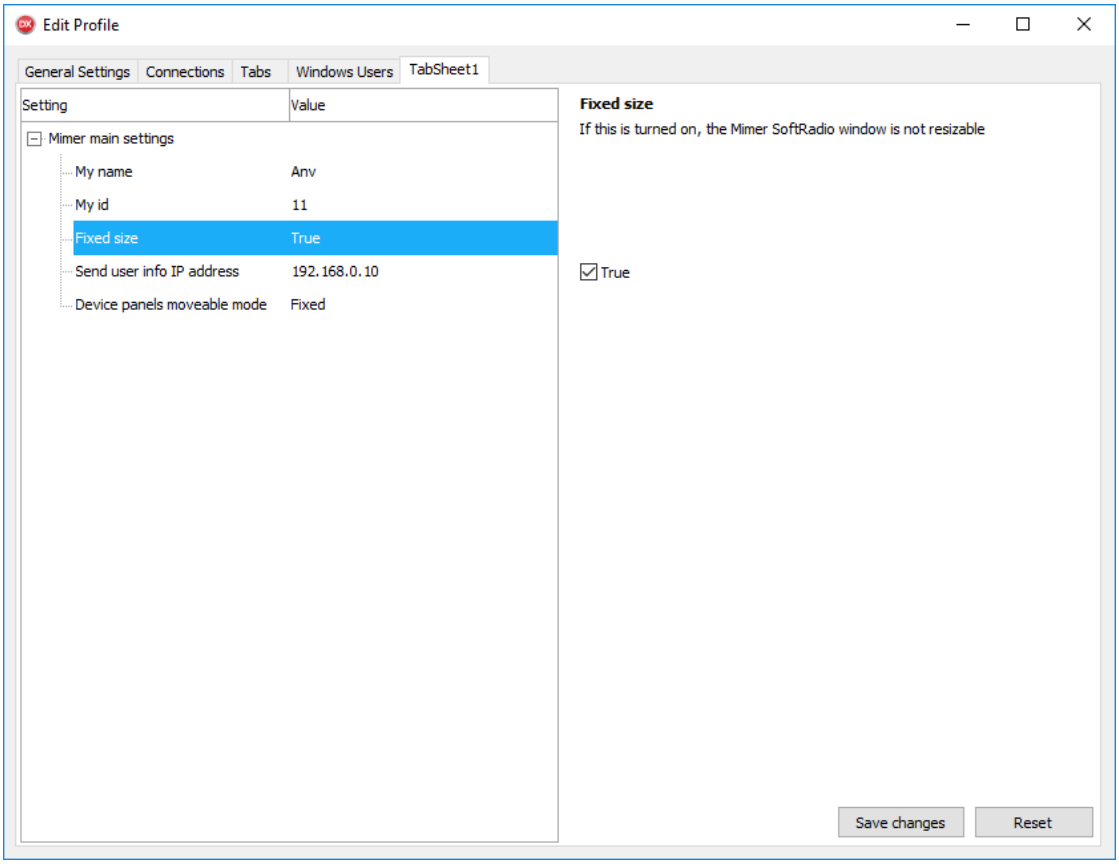
4.4.5 Booleska värden och heltal i användargränssnittet

Booleska värden kan bara vara ett av två värden. Inmatningsfältet som valdes för att presentera booleska värden blev en kryssruta, som visas i figur 4.11. Det finns ingen möjlighet för användaren att mata in ett annat värde än de två tillåtna värdena. Om användargränssnittet ska presentera något annat värde än *true* eller *false* kan nyckelordet *enum* användas, vilket diskuteras mer senare.

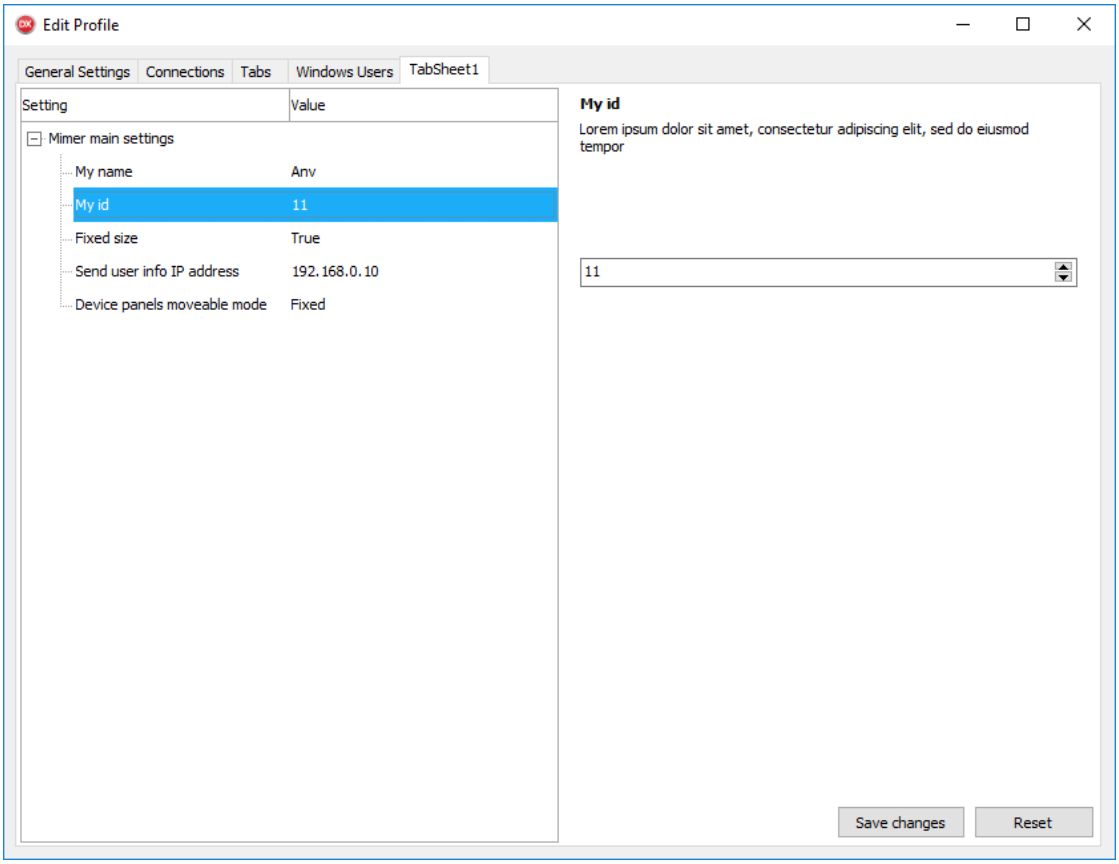
Utöver booleska värden och textsträngar skulle systemet klara av att hantera heltal. Då hanteringen av flyttal fortfarande ansågs vara experimentel och ofärdig, samt att det saknades behov av flyttal så implementerades bara stöd för heltal. Då både klient och server utvecklades med hänsyn till varandra, i samma språk (Delphi), samt att systemet inte var byggt för att hantera flyttal, kunde alla siffror antas vara heltal, och inga oklarheter kring vad som räknas som heltal (se kapitel 2.3.4) behövde hanteras. Om något tal skulle skickas som ett flyttal skulle det talet avrundas och hanteras som ett heltal vid parsningen av JSON-filen.

Heltal presenterades som i figur 4.12. Inmatningsfältet hanterar endast siffror och har dessutom två hjälpknappar för att höja och sänka värdet i steg. Det finns fem valideringsnyckelord enbart för siffror: *multipleOf*, *maximum*, *exclusiveMaximum*, *minimum* samt *exclusiveMinimum*. Nyckelordet *multipleOf* kräver att siffran som valideras ska vara jämnt delbart med värdet beskrivet av *multipleOf*. Användningsområdet för *multipleOf* ansågs inte vara applicerbart för systemet och därför ignorerades. De resterande fyra nyckelorden specificerar ett interval som siffran måste befinna inom. Om en användare försökte mata in ett värde som var utanför intervallet så ändrades värdet till det närmaste kompatibla värdet inom intervallet. [5]

Om stöd för flyttal skulle behövas i framtiden, föreslås att nyckelordet *format* används för att specificera det. Det skulle kunna hanteras på ett sätt som är bakåtkompatibelt med klienter som bara kan hantera heltal, genom att de klienterna hanterar flyttalen som heltal, samtidigt som nyare klienter kan hantera flyttalen som flyttal.



Figur 4.11: Kryssruta för ett booleskt värde



Figur 4.12: Inmatningsfält för heltal

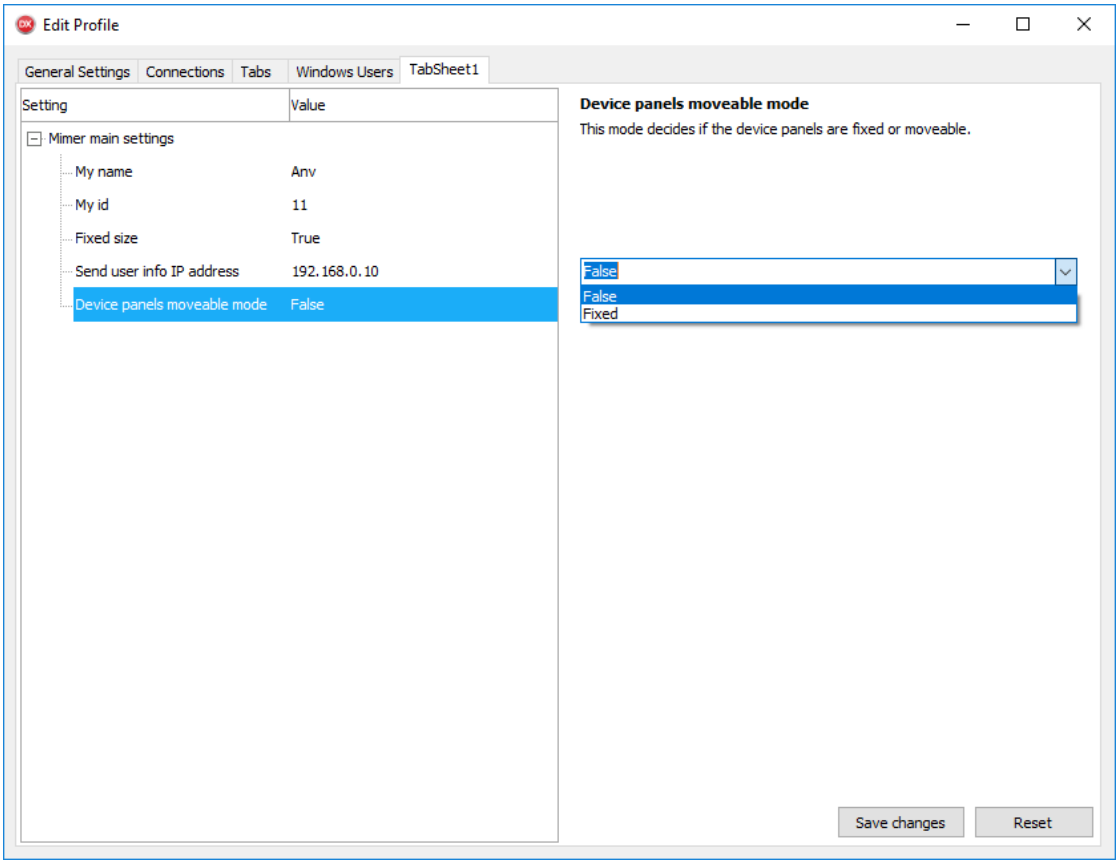
4.4.6 Flervalsalternativ och *enum* i användargränssnittet

Vissa inställningar hade så pass begränsade tillåtna alternativ att bara ett fåtal alternativ på värden var tillåtna. Då användes nyckelordet *enum*. Nyckelordet *enum* används för att specificera en lista med tillåtna värden. Inmatningsfältet som valdes var en lista med alternativ som användaren kan välja, som presenteras i figur 4.13. Det fanns aldrig fall då olika alternativ skulle vara av olika typer, trots att JSON Schema tillåter det [5].

Enum tillåter inga alternativ för att annotera de förutsatta värdena med metadatainformation som *title* eller *description*. Det innebär att värdena som *enum* specificerar, inte går att koppla till någon annan representation än det faktiska värdet. Ett problem skulle exempelvis kunna uppstå om ett värde bara få ha värdena *1*, *2* eller *3* i en JSON-fil, men användarna ska presenteras med *one*, *two* och *three* som svarsalternativ. React JSON Schema Form bemöter det problemet med att utöka JSON Schema och lägga till nyckelordet *enumNames* som i figur 4.14a [49]. Användandet av det nyckelordet är inkompatibelt med JSON Schema [5]. Dessutom finns det massor av oklarheter över hur fall skulle hanteras om *enumNames* finns men *enum* saknas, eller om *enumNames* och *enum* innehåller olika antal element.

Ett annat förslag som React JSON Schema Form föreslår är användandet av nyckelordet *anyOf*, vilket är helt kompatibelt med JSON Schema specifikationerna men är väldigt verbost och använder *anyOf* på ett sätt som det inte ursprungligen är menat för att användas på [5, 49]. Dessutom kräver deras implementation att nyckelordet *enum* ska användas för att definera listor innehållande ett konstant värde, vilket inte är syftet med *enum* [5]. Deras förslag visas i figur 4.14b. Jag föreslår användandet av nyckelordet *const* istället, vilket visas i figur 4.14c. Nyckelordet är till för att specificera att en definition bara får ha ett enda konstant värde och är mindre än ett år gammalt, vilket kan vara en anledning till att React JSON Schema Form inte använt *const* i sin implementation [5].

Lösningen att använda *anyOf* är helt kompatibelt med JSON Schema men kan introducera många risker för fel, då *anyOf* kan användas för att definera flera olika generella definitioner, vilket är mycket mer komplext än att använda *enum* för förbestämda värden. Jag föreslår ett fjärde alternativ vilket varken introducerar fler nyckelord, eller förlitar sig på att olika nyckelord förhåller sig korrekt till varandra, men som inte är helt kompatibelt med de nuvarande JSON Schema specifikationerna. Jag föreslår att *enum* används som tidigare för att specificera förutbestämda värden, men att ett värde också kan vara ett objekt med vissa bestämda nyckelord som används för annotering. Figur 4.15a är ett exempel på det formatet. Nyckelorden *title*, *description* och *const* skulle kunna reserveras. Ett problem vore hur en parser skulle kunna skilja på ett objekt som faktiskt innehåller en *property* med namnet *title*, men det skulle kunna vara bestämt att om ett element i listan i *enum* innehåller en *const-property* så kommer eventuella *title* och *description* användas som metadata för värdet i *const*. Om ett objekt med en *property* med namnet *title* ska specificeras i *enum* så räcker det med att placera det objektet i ett objekt under dess *const-property* som i figur 4.15b. Självklart går det dessutom att blanda element som bara är värdet de representerar med annoterade objekt. Arbetet använde den här metoden för att hantera *enums* och flervalsalternativ.



Figur 4.13: Inmatningsfält för *enum*

```
{
  "type": "number",
  "enum": [1, 2, 3],
  "enumNames": ["one", "two", "three"]
}
```

(a) Hur React JSON Schema Form föreslår *enumNames* [49]

```
{
  "type": "number",
  "anyOf": [
    {
      "type": "number",
      "title": "one",
      "enum": [1]
    },
    {
      "type": "number",
      "title": "two",
      "enum": [2]
    },
    {
      "type": "number",
      "title": "three",
      "enum": [3]
    }
  ]
}
```

(b) React JSON Schema Forms JSON Schema-kompatibla alternativ till *enumNames* [49]

```
{
  "type": "number",
  "anyOf": [
    {
      "type": "number",
      "title": "one",
      "const": 1
    },
    {
      "type": "number",
      "title": "two",
      "const": 2
    },
    {
      "type": "number",
      "title": "three",
      "const": 3
    }
  ]
}
```

(c) Ett modernare förslag av React JSON Schema Forms JSON Schema-kompatibla förslag med *const*

Figur 4.14: Olika alternativ av annotering av *enum*

```
{
  "type": "number",
  "enum": [
    {
      "title": "one",
      "const": 1
    },
    {
      "title": "two",
      "const": 2
    },
    {
      "title": "three",
      "const": 3
    }
  ]
}
```

(a) Enkelt förslag på utökning av enum

```
{
  "type": "number",
  "enum": [
    {
      "age": 18,
      "name": "Erik"
    },
    {
      "const": {
        "title": "Mr.",
        "name": "Julius"
      }
    }
  ]
}
```

(b) Förslag på utökning av enum med objekt som använder *title*

Figur 4.15: Förslag på utökning av *enums* i JSON Scheman

Kapitel 5

Diskussion, slutsats och fortsatt arbete

5.1 Att använda JSON Schema för att generera användargränssnitt

JSON Schema är huvudsakligen ämnat för att validera data, trots att det erbjuder annoteringsfunktionalitet. Att generera ett användargränssnitt från ett JSON Schema är därför inte trivialt. Att validera data kräver bara en uppsättning regler och en kontroll för att utvärdera om datan följer alla regler. Att generera ett interaktivt användargränssnitt kräver att det finns ett tydligt samanhang kring reglerna. Användargränsnittet måste förstå hur data skapas för att passa in i reglerna, och ibland räcker inte regler för att förklara hur data ska skapas. Ett exempel på det är nyckelordet *pattern* som används för att testa en textsträng mot ett ibland komplext mönster. Mönstret ger ingen enkel förklaring till hur en korrekt textsträng ser ut, utan ger bara en komplex upsättning regler.

Nyckelorden *allOf*, *anyOf*, *oneOf* och *not* är ytterligare exempel på nyckelord som passar för att ställa upp en komplex upsättning valideringsregler, men som försvårar för ett användargränssnitt att försöka presentera samanhanget av datan, och vilken data som är korrekt. Det är inte helt trivialt att presentera ett inmatningsfält för en datapunkt som får innehålla en *(oneOf)* textsträng eller ett booleskt värde, men inte *(not)* strängen *"hello world"*.

Nyckelordet *enum* saknar annoteringsmöjligheter för de förutbestämda alternativen i enumvektorn. Att introducera en till vektor som är frikopplad från *enum* med annoteringar som ska tilhöra elementen i enumvektorn är ett alternativ som inte borde rekommenderas. Det går att skapa ett JSON Schema som är felaktigt på sättet att det felaktigt beskriver datan det ska beskriva, men att skriva ett schema som inte går att tolka, eller som kan tolkas på olika sätt ska inte vara möjligt. Pezoa, Reutter, Suarez, Ugarte och Vrgoč utvärderar vissa JSON Scheman som kan tolkas olika av olika validerare. Ett av deras tester testade hur validerare tolkar cykliska referenser, vilket är tillåtet enligt JSON Schema men som är otolkbart av en validerare [1]. Utö-

ver det exemplet känner jag inte till några andra sätt att definera ett korrekt JSON Schema som inte går att tolka. Om en frikopplad vektor lades till där varje element skulle tillhöra ett element i enumvektorn, skulle ytterligare ett sätt att skapa felaktiga JSON Scheman introduceras, om inte strikta specifikationer kring antalet element skulle specificeras.

Alternativet som jag rekomenderar och som projektet introducerar är att elementen i enumvektorn utökades med annoteringsfunktionalitet. Vilka annoteringar som hör till vilka enumvektorelemnt skulle inte kunna tolkas på mer än ett sätt, och det skulle inte begränsa nuvarande funktionalitet. Att kunna annotera enumvektorelementen är viktigt för att kunna generera bra användargränssnitt, där användargränssnittet är frikopplat från datamodellen.

5.2 Fortsatt arbete

Ett intressant område för fortsatt arbete skulle vara att försöka generalisera lösningen som det här arbetet presenterade. Den vänstra trädstrukturen skulle kunna vara mer komplex med utökad funktionalitet, för att hantera objekt och vektorer, medan den högra sidan skulle kunna hantera textsträngar, siffror och booleska värden. Noderna i den vänstra trädstrukturen skulle kunna tydligare delas upp i objektnoder för att innehålla flera fält, och inställningsnoder eller lövnoder för att innehålla en enda enkel datastruktur. Dessutom skulel vissa noder kunna inehålla vektorer, med möjligheten att lägga till, ta bort och byta plats på element i vektorn. En svårighet skulle kunna vara hur ett element som får vara en av flera datatyper skulle hanteras.

Ett annat område som borde arbetas på är att försöka erbjuda all funktionalitet som tidigare implementationer av användargränssnittgenerering erbjuder, utan att använda fler beskrivande dokument än ett JSON Schema eller utöka schemat med extra nyckelord. Nyckelordet *format* skulle kunna användas mycket mer. Det skulle kanske kunna gå att lägga till ett generellt nyckelord som kan erbjuda valfri extra information som kompletering till *format*. Det nyckelordet skulle inte behöva hantera validering, utan skulle bara användas för att lägga till information som *format* inte täcker. Ett annat alternativ vore att tillåta *format* att innehålla ett objekt istället för bara en textsträng.

Vissa tidigare implementationer av genererade användargränssnitt, som exempelvis React JSON Schema Form [49], erbjuder stöd för *conditional schema dependencies* som implementerats på olika sätt men oftast inkompatibelt med JSON Schema specifikationerna. Det betyder att vissa delar av användargränssnittet bara visas om någon eller några datapunkter upfyller vissa krav. Kraven brukar vara strängare än valideringskraven för hela formuläret. Det kan exempelvis handla om att ett inmatningsfält representerar ett booleskt värde som ställer frågan *"Har du någonsin köpt en telefon"*, och om användaren svarar ja så presenteras frågan *"Hur många telefoner har du köpt?"* vilket är en fråga som bara är relevant om användaren svarade ja på första frågan. Nyckelorden *if*, *then* och *else* lades nyss till JSON Schema specifikationerna på den senaste verisonen, vilket kan förklara att ingen av implementatoinerna har implementerat de än [5]. Det saknas arbete som utvärderar användbarheten hos de tre nyckelorden, vilket skulle kunna utvärdera om det finns brister med att använda dem till att generera användargränssnitt eller om de är färdiga tillägg till JSON Schema.

5.3 Slutsats

Arbetet lyckades med att generera användargränssnitt på en skrivbordsapplikation, för att låta en användare manipulera data, som beskrevs med ett JSON Schema. Arbetet hanterade en förutbestämd struktur på JSON Scheman och JSON-dokument men har också föreslagit hur implementationen skulle kunna generaliseras. LSE kan enkelt utveckla Mimer SoftRadio till sina operatörsdatorer utan att behöva bry sig mycket om att synkronisera uppdateringar mellan administratörsprogrammen och operatörsdatorerna hos alla sina kunder. Administratörerna har erbjudits ett användargränssnitt som är enkelt att använda, som tydligt förklarar alla möjliga inställningar och som förklarar hur inställningarna ska ställas in för att vara kompatibla med systemet.

JSON Schema fungerar utmärkt för validering men har brister när det gäller annotering. Det är förväntat i och med att JSON Schema inte är en färdig specifikation. De största bidragande faktorerna vore om *format* utökades med att bidra mer generell annoteringsinformation än en sträng, helst med *format* som objekt, och utökade annoteringsmöjligheter av enumvektorelement

Litteratur

[1] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte och D. Vrgoč, "Foundations of JSON Schema", i *Proceedings of the 25th International Conference on World Wide Web - WWW '16*, New York, New York, USA: ACM Press, 2016, s. 263–273, ISBN: 9781450341431. DOI: 10.1145/2872427.2883029. URL: http://dl.acm.org/citation.cfm?doid=2872427.2883029.

[2] I. ECMA, "The JSON Data Interchange Format", *ECMA International*, årg. 1st Editio, nr October, s. 8, 2013, ISSN: 2070-1721. DOI: 10.17487/rfc7158. arXiv: arXiv:1011.1669v3. URL: http://www.ecma-international.org/publications/standards/Ecma-404.htmhttp://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[3] H. Andrews, A. Wright och Internet Engineering Task Force, "draft-handrews-json-schema-01", tekn. rapport, 2018. URL: https://tools.ietf.org/html/draft-handrews-json-schema-01.

[4] R. Ehne, *LS Elektronik About - LS Elektronik*. URL: http://www.lse.se/about/ (hämtad 2018-04-06).

[5] H. Andrews, A. Wright och Internet Engineering Task Force, "draft-handrews-json-schema-validation-01", tekn. rapport, 2018. URL: https://tools.ietf.org/html/draft-handrews-json-schema-validation-01.

[6] ——, "draft-handrews-json-schema-hyperschema-01", tekn. rapport, 2018. URL: https://tools.ietf.org/html/draft-handrews-json-schema-hyperschema-01.

[7] ——, "draft-wright-json-schema-validation-01", tekn. rapport, 2017. URL: https://tools.ietf.org/html/draft-wright-json-schema-validation-01.

[8] Embarcadero, *Internal Data Formats (Delphi) - RAD Studio*. URL: http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Internal{_}Data{_}Formats{_}(Delphi){\#}Enumerated{_}Types (hämtad 2018-05-07).

[9] Oracle, *Primitive Data Types (The Java™ Tutorials > Learning the Java Language > Language Basics)*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (hämtad 2018-05-07).

[10] Microsoft, *C# Types and Variables - A tour of the C# language | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables (hämtad 2018-05-07).

[11] GNU och Free Software Foundation, *The GNU C++ Library - Chapter 4. Support*. URL: https://gcc.gnu.org/onlinedocs/libstdc++/manual/support.html{\#}std.support.types (hämtad 2018-05-07).

[12] ——, *The GNU C Reference Manual*. URL: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html{\#}Primitive-Types (hämtad 2018-05-07).

[13] J. Britt och Neurogami, *Class: Float (Ruby 2.5.0)*. URL: http://ruby-doc.org/core-2.5.0/Float.html (hämtad 2018-05-07).

[14] ——, *Class: Integer (Ruby 2.5.0)*. URL: http://ruby-doc.org/core-2.5.0/Integer.html (hämtad 2018-05-07).

[15] Python Software Foundation, *2. Lexical analysis — Python 3.6.5 documentation*, 2018. URL: https://docs.python.org/3.6/reference/lexical{_}analysis.html{\#}identifiers (hämtad 2018-05-07).

[16] Ecma International, "ECMAScript ® 2017 Language Specification", 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[17] Space Telescope Science Institute, *Numeric types — Understanding JSON Schema 1.0 documentation*, 2016. URL: https://spacetelescope.github.io/understanding-json-schema/reference/numeric.html (hämtad 2018-05-07).

[18] P. Cederqvist, *multipleOf and floating point rounding errors · Issue #312 · json-schema-org/json-schema-spec*, 2017. URL: https://github.com/json-schema-org/json-schema-spec/issues/312 (hämtad 2018-05-07).

[19] E. Poberezkin, *validation: format[Exclusive](Minimum|Maximum) · Issue #116 · json-schema-org/json-schema-spec*. URL: https://github.com/json-schema-org/json-schema-spec/issues/116 (hämtad 2018-05-07).

[20] M. Faassen, *better support for decimals encoded as strings · Issue #361 · json-schema-org/json-schema-spec*. URL: https://github.com/json-schema-org/json-schema-spec/issues/361 (hämtad 2018-05-07).

[21] The JSON Schema organisation, *Implementations | JSON Schema*. URL: http://json-schema.org/implementations (hämtad 2018-04-18).

[22] Snowplow, *Schema Guru*. URL: https://github.com/snowplow/schema-guru (hämtad 2018-04-20).

[23] Mads Kristensen, *JSON Schema Generator - Visual Studio Marketplace*. URL: https://marketplace.visualstudio.com/items?itemName=MadsKristensen.JSONSchemaGenerator (hämtad 2018-04-20).

[24] A. Romanovich, *JSL*. URL: https://github.com/aromanovich/jsl (hämtad 2018-04-20).

[25] Newtonsoft, *Json.NET Schema - Newtonsoft*. URL: https://www.newtonsoft.com/jsonschema (hämtad 2018-04-20).

[26] R. Suter, *NJsonSchema for .NET*. URL: https://github.com/RSuter/NJsonSchema (hämtad 2018-04-20).

[27] Y. El-Dardiry, *typescript-json-schema*. URL: https://github.com/YousefED/typescript-json-schema (hämtad 2018-04-20).

[28] L. Bovet och Swisspush, *Typson*. URL: https://github.com/lbovet/typson (hämtad 2018-04-20).

[29] Limenius, *Liform*. URL: https://github.com/Limenius/liform (hämtad 2018-04-20).

[30] P. Tomlinson, *APIAddIn*. URL: https://github.com/bayeslife/api-add-in (hämtad 2018-04-20).

[31] Schlothauer & Wauer GmbH, *DJsonSchema*. URL: https://github.com/schlothauer-wauer/DJsonSchema (hämtad 2018-04-20).

[32] ——, *jsonCodeGen*. URL: https://github.com/schlothauer-wauer/jsoncodegen (hämtad 2018-04-20).

[33] M. Kowalczyk och T. Baumann, *aeson-schema*. URL: https://github.com/Fuuzetsu/aeson-schema (hämtad 2018-04-20).

[34] Google, *AutoParse*. URL: https://github.com/google/autoparse (hämtad 2018-04-20).

[35] Tundra, *json-schema-codegen*. URL: https://github.com/VoxSupplyChain/json-schema-codegen (hämtad 2018-04-20).

[36] A. Fenton, *Argus*. URL: https://github.com/aishfenton/argus (hämtad 2018-04-20).

[37] Glimpse I/O Inc., *Bric-à-brac*. URL: https://github.com/glimpseio/BricBrac (hämtad 2018-04-20).

[38] andy Zhangtao, *gojsonschema*. URL: https://github.com/andy-zhangtao/gojsonschema (hämtad 2018-04-20).

[39] Q. inc., *jsonschema*. URL: https://github.com/qri-io/jsonschema (hämtad 2018-04-20).

[40] Gitana Software Inc., *Alpaca Forms - Easy Forms for jQuery*. URL: http://www.alpacajs.org/ (hämtad 2018-04-24).

[41] Textalk, *Angular Schema Form*. URL: http://schemaform.io/ (hämtad 2018-04-24).

[42] Makina Corpus, *Angular2 Schema Form*. URL: https://github.com/makinacorpus/angular2-schema-form (hämtad 2018-04-24).

[43] Jeremy Dorn, *JSON Editor*. URL: https://github.com/json-editor/json-editor (hämtad 2018-04-24).

[44] Joshfire, *JSON Form*. URL: https://github.com/joshfire/jsonform (hämtad 2018-04-24).

[45] Brutusin.org, *Json Forms*. URL: https://github.com/brutusin/json-forms (hämtad 2018-04-24).

[46] EclipseSource, *JSON Forms*. URL: https://jsonforms.io/ (hämtad 2018-04-24).

[47] Nacho Martín, *liform-react*. URL: https://github.com/Limenius/liform-react (hämtad 2018-04-24).

[48] Metawidget, *Metawidget*. URL: http://metawidget.org/ (hämtad 2018-04-24).

[49] Mozilla Services, *react-jsonschema-form*. URL: https://github.com/mozilla-services/react-jsonschema-form (hämtad 2018-04-24).

[50] Network New Technologies Inc., *React Schema Form*. URL: https://github.com/networknt/react-schema-form (hämtad 2018-04-24).

[51] Jackwootton, *JSON Schema Tool*. URL: https://www.jsonschema.net/ (hämtad 2018-04-25).

[52] Embarcadero, *Attributes (RTTI) - RAD Studio*, 2016. URL: http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Attributes{_}Index (hämtad 2018-05-02).

[53] J. de Jong, *JSON Editor online*, 2018. URL: https://jsoneditoronline.org/ (hämtad 2018-05-04).

Bilaga A

API-beskrivning

TODO!